Universidad Nacional Autónoma de México

Unidad Académica de los Ciclos Profesional y de Posgrado del Colegio de Ciencias y Humanidades

AMBIENTE DE PROGRAMACION PARA C INTERPRETE Y DEPURADOR SIMBOLICO

T E S I S

Que para obtener el grado de Maestro en Ciencias de la Computación

p r e s e n t a

Jennie María Becerra Bertram

Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas

México, D. F. **1989**

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE

1. INTRODUCCION

En esta tesis se presenta un ambiente de programación interactivo, integrado y uniforme para el desarrollo de programas en C, que provee al usuario de métodos y herramientas que lo auxilian en el proceso de transformar especificaciones dadas en programas que funcionan. El sistema está formado por cuatro módulos: un editor de pantalla, un compilador a código intermedio, un intérprete y un depurador simbólico. La interface entre el usuario y estos módulos se realiza a través de un procesador de comandos.

La implantación del ambiente surgió ante la necesidad de contar con sistemas que agilicen y faciliten la generación y depuración de programas. El programador, al pasar de especificaciones dadas a programas que funcionan, sigue un ciclo de programación, empleando en cada etapa sistemas que son independientes entre sí, presentan diferentes interfaces y tienen acceso a representaciones distintas del mismo programa, lo que dificulta la tarea del programador. En APC: Ambiente de Programación para C, se han integrado varias herramientas en un solo sistema interactivo, que presenta una interface uniforme para el usuario y en el que se comparte la información generada por las diversas herramientas que lo forman.

El lenguaje de programación C tiene hoy en día una gran importancia y su uso ha sido muy difundido. Desgraciadamente, algunos factores a los cuales debe su popularidad han sido causa de problemas para los programadores. Debido a la importancia del

lenguaje y a los problemas que presenta, este ambiente de programación ha sido creado específicamente para programas que se desarrollen en C.

Por lo complejo del sistema se procedió a dividirlo en dos partes: una formada por el editor y la parte anterior del compilador o "front end", cuya implantación fue realizada por Rodrigo Sigüenza Vega y se reporta en [SIGU89], la otra parte, que se presenta en este trabajo, corresponde a la implantación del intérprete y del depurador.

APC fue desarrollado bajo el sistema operativo UNIX. Para su diseño y construcción se hizo hincapié en la utilización de herramientas ya existentes. En particular, la implantación del compilador, intérprete y depurador se hizo empleando dos programas que forman parte de las utilerías de UNIX, que son Yacc y Lex, las cuales están basadas en la teoría formal de lenguajes.

La utilización de Yacc para la generación del analizador sintáctico, requiere que el lenguaje de programación se exprese por medio de una gramática LALR(1). A la iniciación de este proyecto no fue posible encontrar una gramática LALR(1) de C. El libro The C Programming Language, escrito por los autores del lenguaje, Kernighan y Ritchie [KERN78], que ha sido la fuente tradicional de información, no cuenta con una gramática detallada. Por este motivo, para contar con un marco de referencia, y poder así mismo emplear Yacc para la implantación del sistema, se canalizó una parte importante del esfuerzo en la definición de la gramática que se incluye en el trabajo. Posteriormente, surgió otra escrita por Harbison y Steele que aparece en el C Reference Manual [HARB84].

El Capítulo 2 plantea los problemas que trata de resolver este ambiente de programación. La interface con el usuario y una idea general para la utilización de APC se presentan en el Capítulo 3. En el Capítulo 4 se describe el diseño global del sistema, se mencionan los preprocesos que, para el intérprete y depurador, realizan el editor

y compilador, así como las estructuras de datos que generan. El Capítulo 5 contiene una descripción general de Lex y Yacc, y de la especificación que cada herramienta requiere. El Capítulo 6 describe el diseño e implantación del intérprete. El Capítulo 7 está dedicado a las expresiones y su evaluación. Las capacidades de depuración, que no forman parte integral del intérprete, se presentan en el Capítulo 8. Al final del trabajo se incluye la gramática que se utilizó como especificación del lenguaje.

2. DESCRIPCION DEL PROBLEMA

En el problema que se ha tratado de resolver intervienen dos aspectos: por un lado se tienen las dificultades que, como veremos, presenta el lenguaje de programación C, y por otro lado, la diversidad de herramientas, ajenas unas a otras, que se utilizan en el ciclo de programación.

LENGUAJE DE PROGRAMACION C

C forma parte de los lenguajes de programación algebraica. Fue diseñado por Dennis Richie en los Laboratorios Bell, alrededor de 1972. Aunque, C [KERN78] es un lenguaje de programación de propósito general, se ha usado, tradicionalmente, para programación de sistemas, en particular el sistema operativo UNIX [RITC78] fue escrito en C.

La popularidad de C se debe a varios factores: en primer lugar, provee un conjunto de facilidades para manejar una amplia variedad de aplicaciones. Tiene todos los tipos de datos útiles, incluyendo apuntadores y cuerdas, existe un conjunto amplio de operadores y estructuras modernas de control. C cuenta también con una librería estándar de entrada/salida que flexible para las clases más comunes de entrada y salida a archivos y terminales.

En segundo lugar, los programas en C son eficientes. C es un lenguaje pequeño y sus tipos de datos y operadores están muy relacionados con las operaciones provistas directamente por la mayor parte de las computadoras. Existen varias operaciones cuyo significado se refleja en operaciones implantadas en el hardware de la computadora.

Por todo lo anterior, el número de programadores y programas en C va en constante aumento, es especialmente popular para aplicaciones que deben ser portadas a diferentes máquinas. Muchos de los paquetes de software que exiten en la actualidad, como bases de datos, manejadores de ventanas y sistemas de graficación proveen una interface con C.

Desgraciadamente, algunas de las características de C que contribuyen a su popularidad también plantean problemas a los programadores. Por ejemplo, lo pequeño del lenguaje se debe, en gran parte, a la ausencia de reglas que constriñen en libertad, pero la ausencia de estas reglas puede llevar a hábitos de programación propensos de errores.

Algunos problemas surgen debido a que el significado preciso de ciertos tipos y operadores no está bien especificado; el manejo de tipos complicados con varios modificadores no es claro; la libertad en el uso de apuntadores puede llevar a errores serios, etc. La utilización de C, bajo estas circunstancias, resulta complicada y su aprendizaje se dificulta.

CICLO DE PROGRAMACION

Usualmente, en el desarrollo de programas, el usuario sigue un ciclo tradicional de programación, empleando en cada etapa diferentes herramientas. Estas herramientas son: un editor, un compilador, un ligador y cargador y un depurador (Figura 2.1).

CICLO TRADICIONAL DE PROGRAMACION

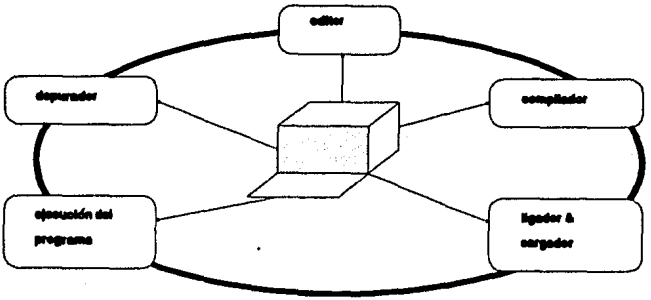

Figura 2.1

Inicialmente requiere de un editor que le permita introducir y modificar su programa. Posteriormente, con un compilador verifica la sintaxis del mismo, si existen errores emplea de nuevo el editor para corregir el programa. Cuando la sintaxis del programa es correcta, el programador invoca al ligador para resolver las referencias externas uniendo sus programas objeto con rutinas del sistema operativo y diversas librerías. En este punto pueden surgir de nuevo errores obligándolo a reiniciar el ciclo con el editor.

Si no existen errores, el cargador únicamente carga el programa en memoria para que pueda ser ejecutado y pasar entonces a la verificación de la lógica del programa. Si se detecta algún problema se puede recurrir a un depurador que le ayude al programador a la localización del error. El ciclo se cierra al emplear de nuevo el editor para modificar

el texto del programa.

Cada herramienta en este ciclo tiene que ser invocada en forma particular, el usuario tiene que aprender las diferentes instrucciones para cada una de ellas y familiarizarse con las interfaces, además de que al utilizar una herramienta no tiene acceso a la información generada por las otras.

OBJETIVOS DE APC

Para facilitar la tarea del programador, las herramientas descritas anteriormente se pueden reunir en un solo sistema integrado formando lo que se conoce como "un ambiente de programación".

Un ambiente de programación está formado por una colección versátil de herramientas y métodos que auxilian al programador en el proceso de transformar especificaciones dadas en programas que funcionan. Se caracteriza por tener una interface consistente entre las herramientas y una representación común de la información, que comparten todas las herramientas.

Los objetivos que se persiguen en APC son la integración del editor, compilador, intérprete y depurador en un ambiente de programación dentro del cual el programador pueda interactuar con estas herramientas en forma interactiva. Se busca también que las herramientas reúnan características, que se mencionarán posteriormente, para facilitar las diversas tareas. El sistema está concebido para el desarrollo de programas pequeños y para un solo usuario (figura 2.2). Es un prototipo en el cual se implanta un subconjunto de C sin incluir las instrucciones al preprocesador y en donde los tipos básicos se restringen a caracteres y enteros.

Se ha piensa que el sistema que se presenta en esta tesis cubre una gran necesidad en el desarrolllo de software relacionado con ambientes de programación, lo anterior se comprueba con la gran actividad que ha tenido lugar en el desarrollo de sistemas de esta naturaleza. Siendo que los autores tienen un interés primordial en la programación intentaron construir una herramienta que auxilie en el desarrollo de programas, que sea un sistema abierto al que se le puedan hacer todas las modificaciones pertinentes y con el que se puedan experimentar diversas opciones y extender capacidades. Los sistemas comerciales que se pueden adquirir son, por lo regular, cerrados, o sea, no se tiene conocimiento de su implantación y no es posible realizar modificaciones.

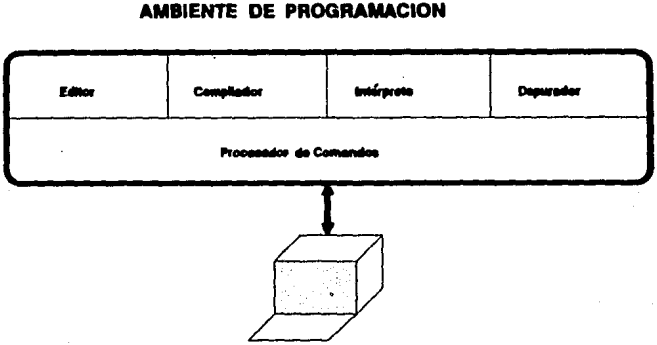

Figura 2.2

En cuanto a las características deseables para el sistema, se ha pensado que la interface con el usuario se realice a través de una terminal con pantalla y que cuente con un manejador de ventanas que permita el acceso simultáneo al texto del programa y a las

diversas herramientas.

En lo referente al tipo de editor empleado, algunos de los sistemas encontrados en la literatura contienen editores dirigidos por sintaxis en los que los elementos primarios que se manipulan son partes de alguna estructura genérica como contrucciones de un lenguaje de programación, lo cual constriñe la interface con el usuario, es menos flexible y, en ocasiones, complica las tareas de edición, que normalmente son sencillas con un editor de texto. Otros sistemas cuentan con editores de línea que no utilizan las capacidades de las pantallas, lo que los hace lentos y difíciles de usar. Por ello, es conveniente que APC cuente con un editor interactivo de pantalla, orientado a texto, tomando como modelo EMACS [STAL81], editor cuyo uso está muy generalizado en la comunidad académica.

Para que el usuario pueda realizar cambios en las expresiones sin tener que recompilar el programa, el compilador no genera ningúna representación intermedia para las expresiones, el intérprete las toma y evalúa directamente del texto; genera, sin embargo, un código intermedio para las proposiciones de C, que representan el flujo del programa. El código intermedio que se genera trata de facilitar la liga entre el texto del programa y las operaciones contenidas en el código.

El lenguaje de comandos del intérprete debe estar formado por expresiones de C, pudiendo contener funciones definidas por el usuario en el programa, o bien pertenecientes a las librerías de C o de APC, estas funciones pueden ser, desde luego, recursivas.

Se busca que el depurador opere a nivel del lenguaje fuente y de las abstracciones definidas por el programador, y de que haga referencia a las localidades del programa por su nombre simbólico, aunque también, si el usuario lo desea, sea posible el manejo de elementos por medio de direcciones. El lenguaje de comandos del depurador debe ser el mismo que el de los programas que se van a depurar, o sea C, para brindarle al

usuario el poder del lenguaje mismo para las funciones de depuración.

Entre las capacidades deseables para el depurador están: la de contar con una función de seguimiento para monitorear el programa, que vaya indicando a nivel de expresión, aquella que se está evaluando en ese momento; la posibilidad de suspender y reactivar la ejecución de un programa mediante la inserción y eliminación de puntos de suspensión en los operadores, realizando esta actividad sin tener que recompilar el programa; la capacidad de desplegar las variables cuyos valores se van modificando. Estas capacidades deberán de poder ser utilizadas en forma conjunta.

También se quisiera que el empleo del depurador fuese flexible, pudiendo emplear las capacidades de depuración a través de diversos mecanismos como: una extensión a los comandos de edición, una extensión al lenguaje de programación por medio de una librería de funciones, o bien, comandos al sistema.

Al integrar las herramientas se quisiera hacer un uso eficaz de la información almacenada en las diversas estructuras generadas por los herramientas, como pueden ser: las tablas de símbolos, la pila de evaluación de expresiones, la pila de registros de activación de las funciones, etcétera, asimismo, permitir al usuario el acceso a estas estructuras y a las diferentes representaciones del programa, como son texto y código intermedio.

ANTECEDENTES

En la literatura se encontraron sistemas que se acercan a lo que se ha descrito como un ambiente de programación y que integran algunas herramientas, sin embargo no reúnen los objetivos que se mencionaron en el inciso anterior. Se ha notado que la mayor parte de los ambientes de programación producidos en círculos académicos se

han orientado hacia la utilización de editores dirigidos por sintaxis, mientras que los sistemas comerciales utilizan editores de texto.

Entre los sistemas existentes están CONA: A Conversational Algol System [ATKI78], que está concebido para ALGOL, tiene algunas limitaciones: no permite la recursividad de funciones y contiene un editor de líneas. Otro similar es COPAS: A Conversational Pascal System, orientado a PASCAL [ATKI81].

El Sintetizador de Programas de Cornell [TEIT81] es un ambiente de programación dirigido por sintaxis. Considera un programa como una composición jerárquica de estructuras computacionales y como tales deben ser editadas, ejecutadas y depuradas. Se desarrolló para PL/1 y posteriormente para Pascal.

El IPE: Incremental Programming Environment [FEIL80], que forma parte del proyecto Gandalf, fue desarrollado para GC, que es una variación de C que verfica tipos. En él, el programador tiene una visión uniforme del programa en términos del lenguaje de programación. El programa se manipula a través de un editor dirigido por sintaxis y su ejecución está controlada por un depurador, cuyas acciones se proveen a través de comandos al editor.

Todos los sistemas anteriores han sido generados en ambientes universitarios, otros más recientes son de tipo comercial, y por esta razón no se tiene acceso a detalles de implantación, nos hemos enterado de su existencia y capacidades a través de anuncios que aparecen en revistas, Byte [BYTE] por ejemplo. La mayor parte de ellos utiliza editores de texto, lo que en cierta forma le da peso a la idea que se tuvo de emplear en APC un editor de texto. Entre ellos están: Living C [LIV/C] que no contempla la evaluación y ejecución de funciones aisladas. El único utilisado por los autores es Run/C [Run/C], contiene un editor de línea, no permite la ejecución de funciones aisladas ni el empleo de puntos de suspensión.

3. EL SISTEMA APC

Como ya se mencionó, el sistema APC está integrado por cuatro herramientas: un editor para introducir y modificar programas, un compilador que genera una versión ejecutable del programa fuente, un intérprete para ejecutar el programa y un depurador que le permite al programador observar la ejecución de un programa, detectar un comportamiento erróneo y localizar su causa.

En este capítulo se presenta el sistema APC desde el punto de vista del usuario, enfatizando la interface con el intérprete y el depurador, se hará también referencia a las capacidades de depuración con que cuenta el sistema.

La interface con el sistema se realiza a través de un procesador de comandos, el cual recibe las instrucciones del usuario y las envía a la herramienta correspondiente. Estos comandos son de dos tipos: por un lado, están los comandos de edición que se utilizan para introducir y modificar un programa y, por otro lado, están los comandos al intérprete y depurador que se utilizan para ejecutar y depurar programas.

Algunas funciones de depuración se activan por medio de comandos que son una extensión a los comandos del editor, otras están incluidas en las funciones del intérprete, por lo que realmente el depurador se halla inmerso en el intérprete, la única diferencia estriba en la intención que se tenga al emplear el intérprete. En lo sucesivo, al ha-

blar de depurador se estará haciendo referencia al empleo del intérprete con fines de depuración. Para auxiliar al usuario en el proceso de depuración, se crearon algunas funciones que forman parte de la librería de APC y que pueden ser invocadas por el usuario.

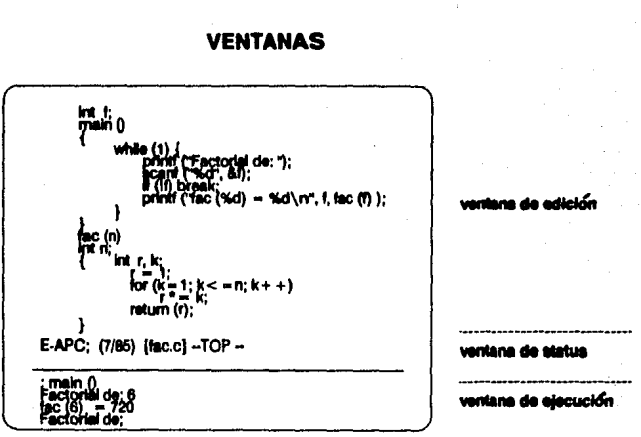

Figura 3.1

VENTANAS

Para facilitar la interface con el usuario, el sistema divide la pantalla en tres áreas o ventanas. El área más grande, situada en la parte superior de la pantalla, es la

ventana de edición, en ella aparece el texto del programa fuente. En esta ventana, el usuario crea y modifica su programa, es decir, las funciones de edición tienen lugar aquí, también se pueden indicar ciertas operaciones de depuración. Si el usuario lo desea, puede obtener retroalimentación acerca de la parte del programa que se está ejecutando en ese momento. Esta información la provee el editor posicionando el cursor cerca de la expresión que se está evaluando.

La segunda ventana, o de ejecución, es la que provee un área de diálogo entre el usuario y el sistema. En ella se escriben los comandos al sistema, como el que ordena la ejecución de un programa y también se despliegan los errores de compilación. La información que despliega el programa del usuario aparece en esta ventana, así como los valores de las variables que se van modificando en el transcurso de la ejecución.

La tercera ventana, o de status, despliega información acerca del estado actual del proceso como el nombre del programa, una marca que nos indica si se ha modificado un programa y no está salvado, un indicador de la posición que ocupa el cursor con respecto al tamaño del programa, etc (figura 3.1).

EDICION

Dentro del editor existe un solo modo de operación, todo carácter que se pulse está considerado como una instrucción. Para extender este conjunto de caracteres se ha "reinterpretado" el teclado alfanumérico, es decir, empleando teclas especiales como la de control *CTRL* o la de escape *ESC* las cuales se oprimen al mismo tiempo que un carácter alfanumérico, se genera otro conjunto de caracteres, sólo que en este caso no tienen una representación visible. Al aumentar el número de caracteres, se extiende también, en forma considerable, el número de instrucciones.

Las teclas de los caracteres visibles están asociadas a una instrucción denominada autoinserción, como resultado el carácter se inserta en el texto en la posición indicada por el cursor. Generalmente, los caracteres no visibles, formados por secuencias de control y escape, generan instrucciones que modifican el documento, cambian la posición del cursor o, en nuestro caso, invocan funciones del compilador o del depurador, también es posible salirse del editor para entrar al modo de comandos.

Para navegar se tienen comandos y así ir al principio o al final de un documento, al principio o al final de una línea, realizar movimientos verticales y horizontales del cursor, moverse hacia el principio o final de un documento hasta posicionar el cursor en la primera ocurrencia de una cuerda dada y dirigirse hacia la parte del documento que se encuentra en la pantalla anterior o posterior de la actual. En cuanto a la edición de un documento, se pueden insertar o eliminar caracteres, insertar o eliminar líneas, reemplazar una cuerda dada por otra. Existen otros comandos como son buscar un documento dado para editarlo o bien almacenarlo en disco.

Los comandos de APC que se efectúan desde la ventana de edición y que son una extensión de los de edición utilizan una secuencia de caracteres de control que empieza siempre con *CTRL-C*, como por ejemplo el comando para transferirse a la ventana de ejecución es *CTRL-C CTRL-X*.

COMPILACION

Una vez que el usuario ha terminado de introducir el programa o bien, le ha hecho los cambios pertinentes, puede proceder a compilarlo, indicando esta operación mediante el comando al editor *CTRL-C CTRL-C*, si existe algún error sintáctico, el compilador se lo indica y posiciona el cursor cerca del sitio en el cual aparece el primer error; si la

compilación es exitosa, aparece en la ventana de ejecución el "prompt" del intérprete:

:

esperando un comando. Si el usuario desea regresar a la ventana de edición se lo indica al intérprete presionando el símbolo '!'

LENGUAJE DE COMANDOS

El lenguaje de comandos está formado, como vimos, por expresiones de C, las cuales están formadas por una secuencia de operadores y operandos, en donde los operandos pueden ser variables o constantes. Para que una expresión sea válida, las variables que intervienen deben ser globales y aquellas que sean automáticas deben ser accesibles en ese momento. Las expresiónes pueden contener funciones ya sean aquellas definidas por el usuario en el programa o bien pueden formar parte de las librerías de C y APC, lo que permite la ejecución de funciones aisladas o bien la ejecución del programa completo mediante la invocación de la función *main*() que es el punto de entrada para programas en C. El intérprete responde a cada expresión del lenguaje de comandos con el símbolo '>' seguido por el valor obtenido de la evaluación de la expresión.

CAPACIDADES DE DEPURACION

El depurador, que forma parte de APC, es interactivo y simbólico. Es interactivo ya que al ejecutar un programa, el usuario puede interactuar con el depurador desde

una terminal. Antes de ejecutar un programa se pueden indicar ciertas operaciones al depurador, como suspender el programa en ciertos puntos. El programa del usuario entonces, se ejecuta mientras no encuentra algún punto de suspensión, cuando este es el caso, el depurador tiene de nuevo el control y espera más instrucciones. El usuario puede entonces, examinar el estado de la computación, modificar valores de variables, así como expresiones contenidas en el texto del programa. El usuario puede también solicitar, en forma interactiva, que se realice un monitoreo del programa en el que se señale, en el texto fuente, la parte del programa que se está ejecutando en ese momento.

Es simbólico ya que puede hacer referencia a localidades del programa por medio de sus nombres simbólicos. El contenido de una localidad puede ser examinado sin necesidad de especificar su dirección. En forma semejante, se puede indicar un punto de suspensión en términos del texto fuente, sin tener que especificar una dirección absoluta, simplemente posicionando el cursor sobre un operador y emitiendo el comando correspondiente.

Entre las funciones que se programaron para auxiliar al usuario en el proceso de depuración están: *stop*(), *trace*() y *show*(). Estas funciones se utilizan respectivamente para: controlar la ejecución del programa, para monitorear la ejecución del mismo y para desplegar valores de variables que se van modificando durante la ejecución.

CONTROL INTERACTIVO DE LA EJECUCION

Para suspender la ajecución de un programa, el usuario se sitúa en la ventana de edición, posicionando el cursor bajo un operador, estableciendo con ello que desea suspender la ejecución antes de efectuar la operación implicada por el operador y pulsa el comando

CTRL-C CTRL-B , el editor refleja esta indicación representando el operador en modo inverso. Se pueden poner tantos puntos de suspensión o "breakpoints" como se deseen. Con el comando *CTRL-C B* se pueden eliminar todos o determinados puntos de suspensión. Tanto para indicar puntos de suspensión como para eliminarlos no es necesario recompilar el programa.

Al terminar de indicar los puntos de suspensión y para ejecutar el programa, el usuario pulsa el comando *CTRL-C CTRL-X*, el cual lo transfiere a la ventana de ejecución, en donde puede interactuar con el intérprete. Para regresar nuevamente a la ventana de edición, ya sea para modificar el programa o para poner o eliminar puntos de suspensión, pulsa el comando '!'.

Cuando se está ejecutando un programa, al llegar a un punto de suspensión, en la pantalla de edición aparece el texto del programa fuente, con el cursor apuntando al operador en el cual se suspendió la ejecución. En la ventana de ejecución, el intérprete despliega:

brk:

con lo que está listo para recibir un comando. Al ocurrir la suspensión dentro de una función, el usuario tiene acceso, además de a las variables globales, a los parámetros y a las variables automáticas, definidas en esa función. Puede el usuario, inclusive, haber desarrollado rutinas personales de depuración, suspender la ejecución en cierto punto clave e invocar alguna de estas rutinas. El usuario indica que desea continuar con la ejecución, pulsando '!', a lo que el sistema contesta con ... *continue* y aparece de nuevo el prompt del intérprete.

SEGUIMIENTO DE LA EJECUCION DEL PROGRAMA

Para dar seguimento a la ejecución de un programa, el depurador cuenta con la función *trace*() que reporta el punto actual de ejecución. Al estar activada esta capacidad, en la ventana de edición aparece el texto del programa fuente con el cursor apuntando a la expresión que se está evaluando en ese momento. Esta capacidad se desactiva por medio de la función *untrace*().

Las funciones *trace*() y *untrace*() pueden ser utilizadas en tres modalidades: desde la ventana de edición, como una extensión a los comandos de edición; desde la ventana de ejecución, como parte de un comando al intérprete, o invocadas desde el programa del usuario, formando parte del texto fuente.

En la primera modalidad, estando en la ventana de edición y pulsando el comando *CTRL-C CTRL-T*, queda activada la función *trace*(), se puede pasar entonces a la ventana de ejecución para ejecutar el programa. Para desactivar la función, siguiendo este mismo método, el usuario se sitúa en la ventana de edición y pulsa el comando *CTRL-C T*.

Estando en la ventana de ejecución se puede invocar la función como un comando al intérprete:

: *trace*()

y posteriormente ejecutar alguna función, como:

: *f*()

También, al suspender el programa se puede indicar que a partir de ese momento se

desea monitorear la ejecución:

brk: *trace*()

y tal vez, en otro punto de suspensión, eliminar este monitoreo, por medio de:

brk: *untrace*()

En la tercera modalidad, invocada desde un programa del usuario, como:

if ($j < n$) *trace*();

DESPLIEGUE DE VALORES

El despliegue de valores de variables que se van modificando, se activa mediante la invocación de la función *show*(). Al ejecutar un programa, estas variables y sus valores aparecen entre paréntesis cuadrados en la ventana de ejecución, como por ejemplo:

[$k = 10$]

Esta capacidad se desactiva mediante la función *unshow*(). Al igual que las funciones descritas en el inciso anterior, se pueden utilizar como: un comando al editor, parte de un comando al intérprete, o invocadas desde el programa del usuario. Como un comando al editor y estando en la ventana de edición mediante el comando ***CTRL-C CTRL-S*** se invoca la función *show*() y mediante el comando ***CTRL-C S*** se invoca la función *unshow*().

La función *show*() se puede combinar con la función *trace*( ), apareciendo, en la ventana

de edición, el programa fuente con el cursor apuntando a la expresión que se está evaluando y en la de ejecución los valores de las variables que se van modificando.

MODIFICACION DE VARIABLES

Esta capacidad de depuración se deriva directamente del lenguaje de comandos del intérprete, ya que, como vimos, el lenguaje de comandos está formado por expresiones de C, y en particular, la asignación es una expresión. Supongamos que tenemos una variable *y* de tipo *int* y queremos asignarle el valor 496, se lo indicamos al intérprete por medio de la expresión:

: $y = 496$

ya sea antes de ejecutar un programa o bien en algún punto de suspensión. El usuario en todo momento tiene acceso a las variables globales, a los parámetros y a las variables automáticas, únicamente al suspender el programa dentro de la función en la cual están definidas.

CONSULTA DE VALORES

Con esta capacidad se recupera y despliega el valor de una variable, como en el caso anterior, se obtiene como un resultado natural del lenguaje de comandos del intérprete. A toda expresión indicada por el usuario, el intérprete responde desplegando el valor que resultó de la evaluación de la expresión. Así tenemos entonces, por ejemplo, que si se desea consultar el valor de la variable de tipo entero *y*, dada la expresión:

```
: y
```

el intérprete desplegará:

```
> 496
```

DEPURACION A BAJO NIVEL

Si el usuario desea realizar depuración a bajo nivel, es decir, por medio de referencias a direcciones absolutas, C provee expresiones con apuntadores para el manejo de direcciones. Así, por ejemplo, si tenemos la variable *y*, obtenemos su dirección por medio de la expresión:

```
: &y
```

Para almacenar un valor en una dirección específica, supongamos que se tiene un apuntador a enteros *int* ∗*p* y que en la localidad 3260 se desea almacenar el valor 121, por medio de las expresiones:

```
: p = 3260
: *p = 121
```

se lleva a cabo. Este tipo de manejo requiere cuidado ya que el usuario debe estar consciente de la forma en que se manejan internamente los tipos y las variables para no cometer errores.

Por ejemplo, para consultar el valor de una variable *y*, bit por bit, se puede hacer uso

de la función de la biblioteca de C *printf* con un formato hexadecimal, como el que sigue:

: $printf("\%x", y)$

En el caso de *printf*, se obtiene en la ventana de ejecución el valor de y en formato hexadecimal y posteriormente el intérprete entrega el resultado de evaluar *printf*, que, si se realiza sin problemas, será igual a 1.

En el ejemplo sencillo que aparece en la figura 3.1, notamos que el usuario, en la ventana de ejecución, ha pedido la ejecución del programa indicándolo mediante la expresión *main*(); los tres renglones siguientes corresponden a las instrucciones *printf*() y *scanf*() que pertenecen al programa fuente. Si se quisiera verificar, en forma aislada, la función *fac*(), el usuario lo puede indicar por medio de la expresión:

: $fac(5)$

a lo que el intérprete contestaría:

> 120

Otra forma posible sería por medio de la secuencia:

: $t = 3$
> 3
: $fac(t)$
> 6

en donde el usuario tiene acceso a la variable t, ya que es global. Si el usuario utiliza

la expresión:

: *trace*()

y posteriormente ejecuta la función *fac*(), en la ventana de edición se despliega el texto del programa y el cursor se va posicionando bajo cada uno de los operadores que forman parte de las expresiones que aparecen en la función:

$f = 1$
$k = 1$
$k <= n$
$f* = k$
$k++$
...

con lo que se puede monitorear la ejecución. Si además desea ver los valores de las variables que se van modificando, lo indica con la función *show*() y en la ventana de edición se desplegará:

$[f = 1]$
$[k = 1]$
$[f = 1]$
$[k = 2]$
$[f = 2]$
...

y así sucesivamente hasta terminar. Supongamos que pone ahora un punto de suspensión en el operador '* =' de la expresión $f* = k$ y posteriormente indica:

```
: fac(3)
```

al llegar a la expresión que contiene el punto de suspensión, se suspende el programa y el usuario puede, por ejemplo, consultar los valores de los parámetros actuales o de las variables locales a las que tiene acceso en ese momento:

```
brk: n
```

y el intérprete le responde:

```
brk > 3
```

puede invocar de nuevo la función *fac*(), y no hay ningún problema, pués se genera simplemente otra activación de la misma función:

```
brk: fac(4)
brk > 24
```

en la que no tiene efecto el punto de suspensión. La secuencia:

```
brk: !
... continue
> 6
:
```

continua con la evaluación.

4. DISEÑO DEL SISTEMA

El sistema está formado por dos módulos: un editor y un traductor dirigido por sintaxis, en el que están integrados el compilador, el intérprete y el depurador. El traductor contiene las reglas gramaticales y acciones semánticas que le permiten desarrollar estos diferentes procesos. Al diseñar un intérprete poderoso, se logró que la mayor parte de las actividades de depuración quedaran inmersas en el intérprete. Como se mencionó con anterioridad, la comunicación entre el usuario y estos módulos se realiza a través de un procesador de comandos (figura 4.1).

EDITOR

La arquitectura del editor es modular y está formado por un procesador interno de comandos, un módulo de edición, un módulo de navegación, un módulo de despliegue y una interface con el ambiente de programación.

El procesador interno recibe un comando, lo analiza y ordena la ejecución de la rutina semántica apropiada. Las rutinas por medio de las cuales el usuario selecciona la parte del programa que va a ser manipulada y desplegada en la pantalla se hallan reunidas en el pmódulo de navegación. El módulo de edición contiene las rutinas por medio de las cuales se modifica el programa. El módulo de despliegue se encarga de mostrar

la parte del programa seleccionada por el usuario y de reflejar los cambios indicados. La interface con el ambiente de programación contiene rutinas que ligan el texto del programa con ciertas funciones de depuración.

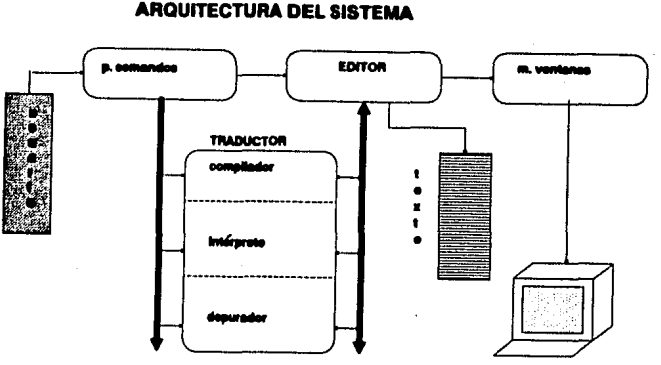

Figura 4.1

El módulo de despliegue contiene las rutinas que muestran el texto y reflejan los cambios realizados. Sus funciones consisten básicamente en mapear esta información al dispositivo físico de salida de la manera más eficiente posible. Para permitir la portabilidad de este módulo, las rutinas son independientes del tipo de dispositivo empleado, en lugar de tener secuencias explícitas de caracteres de control para las terminales en las rutinas de despliegue, se emplean funciones de biblioteca que contienen las secuencias apropiadas de control para la terminal anfitrión. Este módulo es también el que tiene a su cargo el manejo de las ventanas.

El editor cuenta con un buffer de tamaño variable que reside en memoria, al cual copia el documento que se desea editar. Este buffer contiene todas las líneas del documento delimitadas por un carácter nulo. El manejo del espacio en el buffer se realiza utilizando las funciones de la librería de C que asignan y liberan espacio en memoria.

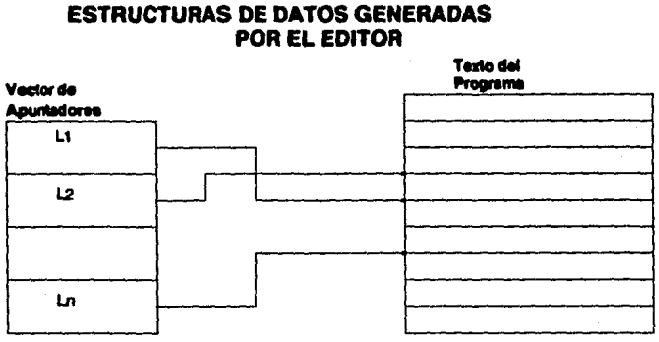

Figura 4.2

Existe también, en memoria, un vector, en el cual cada entrada contiene un apuntador a una línea de texto. Estos apuntadores siguen un orden secuencial lógico, la primera entrada del vector se refiere a la primer línea lógica, la segunda a la segunda línea lógica y así sucesivamente. Cuando el documento se copia inicialmente a memoria, la primer línea física del buffer es también la primer línea lógica. Esta organización permite que las operaciones que manejan líneas se realicen sobre el vector, en lugar de hacerlo sobre el buffer. Para insertar líneas, se recorren los apuntadores en el

vector para permitir la inserción de un número dado de apuntadores que corresponde al número de líneas insertadas; para eliminar líneas, basta únicamente eliminar los apuntadores correspondientes a esas líneas y recorrer los apuntadores para cerrar los huecos. Cualquier modificación al texto se realiza en dos sitios: sobre la pantalla e internamente en las estructuras que contienen el texto (figura 4.2).

TRADUCTOR DIRIGIDO POR SINTAXIS

El módulo que comprende al compilador, intérprete y depurador es un traductor dirigido por sintaxis. El compilador toma un programa fuente y produce, como resultado de la traducción, un código intermedio; el intérprete, a partir de una expresión en C, produce como resultado su valor; el depurador, como se vió anteriormente está inmerso en el intérprete. Se utilizó una herramienta, la cual, a partir de una definición dirigida por sintaxis del lenguaje de programación, produce un traductor.

Para efectuar la traducción de una construcción que forma parte de un lenguaje de programación, es necesario tomar en cuenta varios aspectos asociados con las construcciones, éstas pueden ser, por ejemplo, el tipo de construcción, una cuerda de caracteres, una localidad en memoria, etc. Nos referiremos a estas cantidades como atributos que están asociados a construcciones. Para especificar las traducciones de las construcciones que forman el lenguaje de programación C se ha empleado un formalismo conocido como definición dirigida por sintaxis.

Una definición dirigida por sintaxis utiliza una gramática libre de contexto para especificar la estructura sintáctica de la entrada. Con cada símbolo gramatical, asocia un conjunto de atributos, y con cada producción un conjunto de reglas semánticas para evaluar los valores de los atributos asociados con los símbolos que aparecen en esa

producción. La gramática y el conjunto de reglas semánticas constituyen la definición dirigida por sintáxis. Cuando se incluyen fragmentos de programas, a los que se les llama acciones semánticas, dentro de los lados derechos de las producciones, se le llama esquema de traducción. Realmente, un esquema de traducción es similar a una definición dirigida por sintáxis, excepto que el orden de evaluación de las reglas semánticas se muestra en forma explícita.

En la actualidad existen varias herramientas especializadas diseñadas para la construcción automática de traductores; para generar el traductor de APC se utilizó Yacc [JOHN75], cuyo nombre proviene de "yet another compiler-compiler", aunque realmente no genera un compilador completo sino únicamente lo que corresponde al "front end" del mismo. Yacc recibe la especificación del proceso de entrada; esto incluye reglas gramaticales que definen la estructura de un lenguaje, código que se invoca cuando se reconocen estas reglas y una rutina de bajo nivel que realiza el proceso básico de entrada. Yacc genera entonces una función que controla el proceso de entrada y produce una traducción. En general, los traductores producidos utilizando este método son eficientes.

Para obtener la rutina de bajo nivel que realiza el proceso básico de entrada, es decir, aquella que procesa la cuerda de caracteres que forma las construcciones del lenguaje, se utilizó Lex [LESK75], también es una herramienta de la utilería de Unix. La especificación para Lex consiste en un conjunto de expresiones regulares junto con una acción para cada expresión regular. La rutina, así obtenida, es ineficiente, pero para los propósitos de este trabajo esto no tiene demasiada importancia, ya que lo que se desea es un prototipo que funcione, haciendo caso omiso, por lo pronto, de consideraciones de eficiencia. En el capítulo siguiente se da una breve explicación sobre Lex y Yacc, así como de las notaciónes que se requieren para su especificación las cuales son, respectivamente, las expresiones regulares y una gramática libre de contexto (figura 4.3).

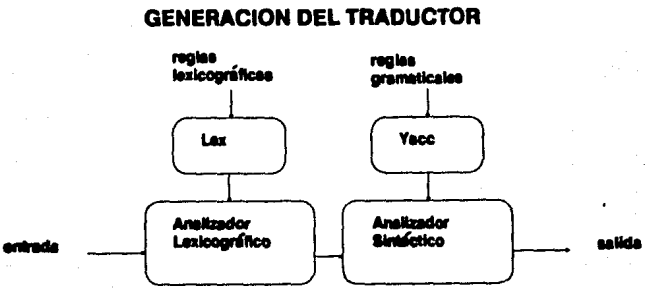

Figura 4.3

COMPILADOR

Usualmente, el proceso de compilación se divide en dos partes: análisis y síntesis, conocidas también, respectivamente, como "front end" y "back end". La parte de análisis rompe el programa fuente en sus partes constitutivas y crea una representación intermedia del programa fuente. Esta parte depende principalmente del lenguaje fuente y es independiente de la máquina objeto.

La parte de síntesis construye el programa objeto deseado a partir de la representación intermedia. Esta parte incluye las porciones del compilador dependientes de la máquina objeto. La representación intermedia generada por el compilador está formada por un código intermedio. En APC, el compilador realiza únicamente la parte de análisis o

"front end" del compilador. Para la parte de síntesis se tiene un intérprete, el cual en lugar de producir un programa objeto, como producto de la traducción, recibe expresiones y produce su valor, ejecutando para ello las operaciones contenidas en el el código intermedio.

El proceso de compilación, en su parte de análisis, se ha particionado, como es usual, en dos subprocesos o fases para facilitar el diseño y la implantación. La primera fase o de análisis lexicográfico lee de izquierda a derecha la cuerda de caracteres que forma el programa fuente y la agrupa en elementos léxicos o "tokens", los cuales son secuencias de caracteres que tienen un significado colectivo, entre estos tenemos palabras reservadas, identificadores, caracteres especiales, operadores y constantes. La salida del analizador lexicográfico es una serie de elementos léxicos que se pasan a la siguiente fase que es la del análisis sintáctico, este analizador lexicográfico es el producido por Lex.

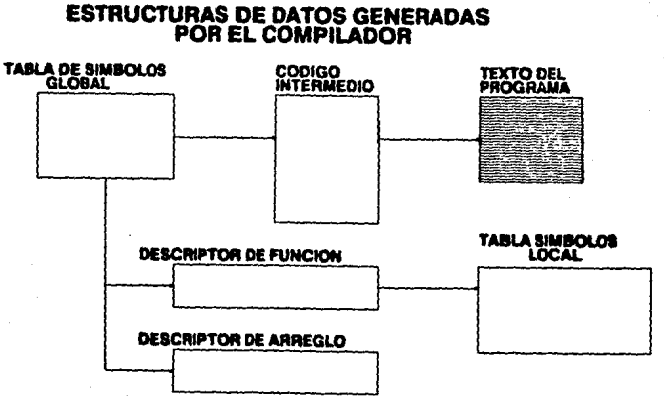

Figura 4.4

La segunda fase o análisis sintáctico toma los elementos léxicos y los agrupa para formar estructuras sintácticas, como expresiones, y posteriormente combina éstas para formar proposiciones del lenguaje fuente, generando en el camino el código intermedio. Este es el analizador sintáctico o traductor generado por Yacc.

Un programa en C está formado por tres tipos diferentes de elementos: declaraciones, proposiciones y expresiones. Con cada una de ellas el compilador realiza un tipo de acciones diferentes englobadas en las acciones semánticas de la especificación para Yacc. En este proceso genera estructuras con información utilizadas posteriormente el intérprete (figura 4.4).

DECLARACIONES

A partir de las declaraciones, registra los identificadores empleados en el programa fuente y reúne información acerca de los atributos de estos identificadores, como son: el espacio asignado a un identificador, su tipo y alcance, o sea, la porción del programa en la cual se aplica la declaración. En el caso de funciones, el número y tipo de sus argumentos, así como las variables automáticas declaradas en esa función. La tabla de símbolos es la estructura de datos que contiene esta información, cada entrada en la tabla proviene de la declaración de un nombre. En realidad no se trata de una sola tabla, sino que se tiene una global en la que se almacenan las variables globales, de las cuales forman parte las funciones y otras tablas locales que contienen los parámetros y variables automáticas para cada función definida.

Las tablas de símbolos están formadas por arreglos de descriptores que contienen los atributos asociados a identificadores, los cuales se verán con mayor detalle en el capítulo dedicado a la evaluación de expresiones. Tenemos a continuación el descriptor con sus

campos correspondientes:

```
typedef struct {
        char *sname;
        ushort satt;
        ushort ssize;
        char *sival;
        char *sdesc;
} symT;
```

en donde :

- *sname* es un apuntador al lexema o cuerda de caracteres que denota el nombre.

- *satt* contiene el tipo básico y sus modificadores,

- *ssize*, si se trata de una variable corresponde al tamaño y si se trata de una función, es el número de instrucciones generadas en código intermedio.

- *sival*, si es una variable, contiene un desplazamiento el cual indica la posición relativa que le será asignada al ser almacenada y si es una función, contiene un apuntador a su código intermedio.

- *sdesc*, cuando la variable es una función o un arreglo, contiene un apuntador a un descriptor con información adicional para estos elementos.

El descriptor que contiene información acerca de las funciones es el siguiente:

```
typedef struct {
        char npar;
        char type;
        symT *ht[SIZEHF];
} fundT;
```

en donde:

- *npar* es el número de parámetros de la función.

- *type* es el tipo del valor de regreso de la función.

- *ht* es un apuntador a la tabla local de símbolos para esta función.

El descriptor que contiene información acerca de los arreglos es el siguiente:

```
typedef struct {
        tiny ndim;
        ushort dim[MAXDIM][2];
} arrdT;
```

en donde:

- *ndim* es el número de dimensiones del arreglo.

- *dim* es el número de elementos y tamaño de los mismos por cada dimensión.

PROPOSICIONES Y EXPRESIONES

A medida que el compilador reconoce una proposición del lenguaje fuente genera una representación intermedia de la misma, formada por un código intermedio. En las proposiciónes intervienen dos tipos de elementos: por un lado está el esqueleto de la proposición, que representa el flujo del programa y por el otro, se encuentran las expresiones; el código intermedio que se genera corresponde únicamente al esqueleto de las proposiciones.

Para las expresiones, no genera ningún tipo de representación intermedia, solamente registra la posición que ocupa la expresión en el texto fuente y almacena en el código intermedio, en los campos reservados para operandos, apuntadores a las expresiones que intervienen en las proposiciones, las cuales serán evaluadas por el intérprete, directamente del texto fuente.

Este esquema se utilizó con el objeto de facilitar el proceso de depuración y evitar recompilaciones, en el caso de que el usuario modifique una expresión o establezca puntos de suspensión en el programa, ello implica, sin embargo, un "overhead" considerable en el tiempo de ejecución. Se han considerado algunas ideas para reducir este "overhead" como es, por ejemplo, el contar con un "cache" de expresiones en alguna representación intermedia y únicamente en el caso de una modificación realizar de nuevo los análisis lexicográfico y sintáctico. Esto disminuiría mucho el "overhead" sobre todo tratándose de expresiones que están dentro de los ciclos de un programa, ya que sólo se analizarían la primera vez.

Una función está integrada por un conjunto de proposiciones, por lo que cada función tiene asociado un conjunto de operaciones del código intermedio. El compilador, al detectar la definición de una función, inserta en el campo *sival* del descriptor de la tabla de símbolos global un apuntador a su código intermedio correspondiente.

El código intermedio está representado por un arreglo de estructuras, en donde a cada estructura está asociada una instrucción. Las instrucciones están ordenadas secuencialmente y la posición relativa, que ocupan con respecto al principio del código para esa función constituye su dirección en el arreglo. Una instrucción está formada por tres campos: el que contiene el operador y los dos que contienen los operandos, éstos pueden ser: o bien apuntadores al principio y final de una expresión contenida en el texto, o bien, una constante numérica, la cual puede referirse a una dirección del código.

A continuación tenemos el descriptor para el código intermedio:

```
typedef struct {                                    /* instrucción */
        tiny op;                                    /* operador */
        long a1, a2;                                /* operandos */
} quadT;
quadT memcode [MAXCODE];            /* arreglo de instrucciones */
```

Los operadores que maneja el código intermedio son:

- EVAL: Evalúa la expresión cuya dirección está indicada por los operandos y ejecuta la instrucción $i + 1$.

- IF: Condicional, evalúa la expresión, si es igual a cero, ejecuta la instrucción $i + 1$, si es diferente a cero ejecuta la instrucción $i + 2$.

- SWITCH: Evalúa la expresión y ejecuta cada una de las instrucciones que siguen con el operador CASE, empezando con la instrucción $i + 1$.

- CASE: Compara el valor de la expresión obtenida de SWITCH con el primer operando de CASE, que es una constante, si son iguales ejecuta la instrucción

$i + 2$, si no lo son ejecuta la instrucción $i + 1$.

- DEF: Sin comparar, ejecuta la instrucción $i + 1$, equivale a una instrucción nula.

- GOTO: Ejecuta la instrucción cuya dirección está indicada por el primer operando.

- RETV: Regresa de la función, tomando como valor de retorno el de la expresión.

- RET: Regresa de la función, sin valor de retorno.

INTERPRETE

El intérprete se construyó utilizando también el principio de traducción dirigida por sintaxis, como entrada recibe una expresión que le entrega el usuario a través de la terminal y como salida produce el resultado de su evaluación.

Como vimos, para las expresiones no se genera una representación intermedia, por lo que, a medida que el intérprete va reconociendo los elementos sintácticos que integran las expresiones, va calculando su valor; en este sentido realiza, al igual que el compilador, el análisis lexicográfico y el análisis sintáctico de las expresiones. El intérprete tiene también a su cargo el análisis semántico de las expresiones, el cual se realiza en forma dinámica durante la evaluación de las mismas. Una de las funciones más importantes en el análisis semántico consiste en la verificación de tipos.

Las reglas gramaticales que definen las expresiones son las mismas, tanto para el compilador como para el intérprete, en lo que difieren es en las acciones semánticas asociadas a ellas. Si la expresión, que forma parte del lenguaje de comandos, no contiene funciones, basta con la realización del análisis sintáctico de sus elementos para obtener su valor.

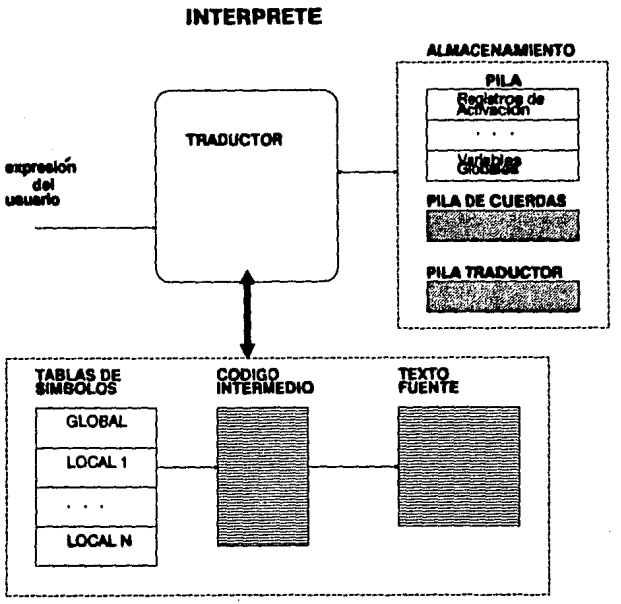

Figura 4.5

Ahora bien, si en la expresión aparece la invocación a una función, el intérprete, como parte de la evaluación, tiene que ejecutar el código intermedio asociado a la función.

Para realizar este proceso se utilizó el mismo traductor, introduciendo reglas gramaticales, cuyo propósito es ejecutar el código intermedio. Los elementos sintácticos de estas reglas son los operadores y operandos que integran el código intermedio y las instrucciones para ejecutar el código están incluídas en las acciones semánticas asociadas a las reglas.

Algunas operaciones del código intermedio contemplan la evaluación de expresiones las cuales se toman directamente del texto del programa, en donde el intérprete al mismo tiempo que realiza el análisis sintáctico lleva a cabo la evaluación. Como se puede notar, el proceso que se sigue para evaluar las expresiones recibidas a través de la terminal y las expresiones contenidas en el texto fuente es similar, aún cuando la fuente de la cual provienen es diferente. En el proceso se distinguen también dos niveles de anidamiento: en el primer nivel se encuentran la expresiones indicadas por el usuario desde la terminal y en un segundo nivel están las expresiones que forman parte del programa fuente.

El intérprete, además de evaluar expresiones y de ejecutar el código intermedio, relaciona el texto del programa con las acciones que deben ocurrir a tiempo de ejecución para poder ejecutar el programa. Para lograr este fin cuenta con las acciones semánticas que manejan el medio ambiente a tiempo de ejecución (figura 4.5).

La especificación del traductor está entonces, formada por un conjunto de reglas gramaticales y acciones semánticas, de las cuales algunas corresponden solamente al compilador, algunas al intérprete y otras se comparten por el compilador y el intérprete. Entre las primeras, tenemos las que se refieren a las definiciones de variables y funciones, así como las asociadas a las proposiciones. Entre las que corresponden sólo al intérprete, tenemos las que se refieren al código intermedio, las que comparten el compilador y el intérprete son las que se refieren a expresiones.

5. HERRAMIENTAS LEX Y YACC

La construcción del traductor dirigido por sintaxis que engloba al compilador, intérprete y depurador, se realizó empleando herramientas que utilizan resultados interesantes de la teoría de análisis lexicográfico y sintáctico. Unix provee dos herramientas poderosas, Lex y Yacc, para facilitar estos procesos. Lex genera un analizador lexicográfico a partir de expresiones regulares y Yacc construye un analizador sintáctico LALR(1) a partir de una gramática libre de contexto. Estos programas facilitan las tareas de diseño e implantación; el módulo producido por Yacc, a partir de sus especificaciones, es rápido y eficiente, sin embargo, el módulo producido por Lex, aunque es práctico, es ineficiente.

LEX

Varias herramientas han sido creadas para construir analizadores lexicográficos a partir de notaciones de propósito específico basadas en expresiones regulares. Una herramienta, en particular, llamada Lex, ha sido ampliamente empleada, sobre todo para especificar analizadores lexicográficos en la búsqueda de patrones en texto. Nos referiremos a esta herramienta como el compilador Lex y a la especificación de entrada como el lenguaje Lex.

Dado que la especificación para Lex consiste de un conjunto de expresiones regulares, empezaremos por indicar qué se entiende por ello, introduciendo notación adicional que se puede emplear en Lex. Posteriormente indicaremos la forma en que se usa Lex.

EXPRESIONES REGULARES

Las expresiones regulares son una notación importante para especificar patrones. Daremos a continuación las reglas que definen las expresiones regulares sobre el alfabeto A:

- e es una expresión regular que denota e, es decir el conjunto que contiene la cuerda vacía.

- Si a es un símbolo en A, entonces a es una expresión regular que denota a, es decir, el conjunto que contiene la cuerda a.

- Supongamos que r y s son expresiones regulares que denotan los lenguajes $L(r)$ y $L(s)$, entonces,

a) $(r)|(s)$ es una expresión regular que denota $L(r)UL(s)$.
b) $(r)(s)$ es una expresión regular que denota $L(r)L(s)$.
c) $(r)^*$ es una expresión regular que denota $(L(r))^*$.
d) (r) es una expresión regular que denota $L(r)$.

en donde la barra vertical '|' denota "o", el asterisco '*' significa "cero o más instancias" de la expresión en paréntesis y la yuxtaposición de una *letra* con el resto de la expresión denota concatenación. Estos tres operadores son asociativos por la izquierda, teniendo '*' la mayor precedencia y '|' la menor. Cada expresión regular r denota un lenguaje

$L(r)$. Las reglas especifican la manera en que $L(r)$ se forma por medio de la combinación de lenguajes denotados por las subexpresiones de r.

Lex permite también introducir otras construcciones como son:

- Por medio del empleo del operador unario postfijo '+' se puede indicar "una o más instancias de". Si r es una expresión regular, que denota el lenguaje $L(r)$, entonces $(r)^+$ es una expresión regular que denota el lenguaje $(L(r))^+$. Por ejemplo, una expresión regular a^+ denota el conjunto de todas las cuerdas formadas por una o más a.

- Por medio del empleo del operador unario postfijo '?' se indica "cero o una instancia de". La notacion $r?$ equivale a $r|e$. Si r es una expresión regular, entonces $(r)?$ es una expresión regular que denota el lenguaje $L(r) \cup e$.

- Mediante el par '[ ]' se pueden indicar clases de caracteres. Por ejemplo, $[abc]$ denota la expresión regular $a|b|c$, y con $[a-z]$ se denota la expresión regular $a|b|...|z$.

COMPILADOR LEX

En general para emplear Lex, en primer lugar se prepara la especificación de un analizador lexicográfico mediante la creación de un programa *c.l* en el lenguaje Lex. Este programa se corre a través del compilador Lex para producir un programa en C denominado *lex.yy.c*. El programa *lex.yy.c* consiste de una representación tabular de un diagrama de transición construido a partir de las expresiones regulares contenidas en *c.l* junto con una rutina estándar que usa la tabla para reconocer lexemas. Las acciones

asociadas con las expresiones regulares en *e.l* son porciones de código en C, llevadas directamente a *lex.yy.c*. Finalmente, *lex.yy.c* se corre a través del compilador de C para producir un programa objeto *lex.yy.o*, el cual es el analizador lexicográfico que transforma la cuerda de entrada en una secuencia de elementos léxicos (figura 5.1).

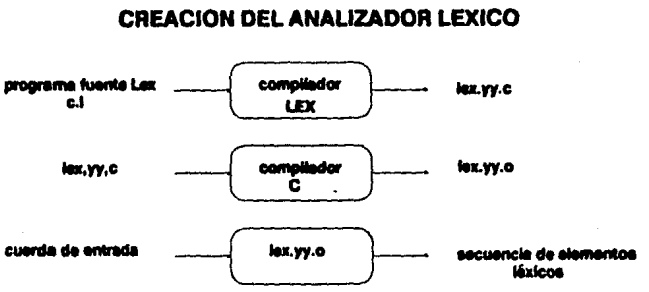

Figura 5.1

Un programa en Lex consiste de tres partes:

declaraciones
%%
reglas de traducción
%%
procedimientos auxiliares

La sección de declaraciones incluye declaraciones de variables, constantes manifiestas y definiciones regulares. Las definiciones regulares son proposiciones mediante las cua-

les se le da un nombre a una expresión regular para, posteriormente, en las reglas de traducción, definir expresiones regulares, empleando estos nombres como si fuesen símbolos.

Las reglas de traducción de un programa en Lex son proposiciones de la forma:

p1 { *acción-1* }
p2 { *acción-2* }
...
pn { *acción-n* }

donde cada *pi* es una expresión regular y cada acción *accion-i* es un fragmento de programa describiendo qué acción debe tomar el analizador lexicográfico cuando el patrón *pi* casa con un lexema, en Lex, las acciones están escritas en C.

La tercera sección contiene los procedimientos necesarios que necesitan las acciones. Estos procedimientos pueden existir en forma separada y cargarse con el analizador lexicográfico.

Un analizador lexicográfico creado por Lex se comporta, en relación a un analizador sintáctico, de la siguiente manera: al ser activado por el analizador sintáctico, el analizador lexicográfico empieza a leer el remanente de la cuerda de entrada, un carácter a la vez, hasta que encuentra el prefijo más largo de la entrada que casa con una de las expresiones regulares *pi*; entonces ejecuta la *accion-i*. Típicamente, la *accion-i* le regresa el control al analizador sintáctico. Si esto no ocurre, el analizador lexicográfico procede a encontrar más lexemas, hasta que una acción regrese el control al analizador sintáctico. La búsqueda repetida de lexemas, hasta encontrar un regreso explícito, permite al analizador lexicográfico procesar espacios y comentarios de manera conveniente.

El analizador lexicográfico regresa solo un valor, que es el elemento léxico, al analizador sintáctico. Para pasar información adicional acerca del lexema existe una variable global denominada *yylval*.

La especificación de los elementos léxicos de C, por medio de expresiones regulares, se expone en detalle en la primera parte de este trabajo [SIGU89]. De las expresiones regulares adicionales, que se necesitaron para la implementación del intérprete y depurador, hablaremos posteriormente en los capítulos dedicados a estos dos módulos.

YACC

Yacc es un generador de analizadores sintácticos LALR, ampliamente empleado para la implantación de compiladores. Fue creado por S. C. Johnson a principios de 1970, es una abreviatura de "yet another compiler-compiler". La especificación para Yacc se realiza utilizando una gramática libre de contexto.

GRAMATICA LIBRE DE CONTEXTO

Por medio de una gramática se trata de describir la estructura jerárquica de un lenguaje de programación. Una gramática libre de contexto es una notación empleada para especificar la sintaxis de un lenguaje, también se le llama BNF de Backus-Naur Form. A continuación damos la definición de una gramática libre de contexto a la que en lo sucesivo nos referiremos como gramática únicamente. Una gramática consta de cuatro componentes: terminales, noterminales, un símbolo de inicio y producciones.

- Los terminales son los elementos léxicos a partir de los cuales se forman

las cuerdas. Las palabras "token" , terminal y elemento léxico son sinónimos.

- Los noterminales son variables sintácticas que denotan conjuntos de cuerdas. Los noterminales definen conjuntos de cuerdas que ayudan en la definición del lenguaje generado por la gramática. También imponen una estructura jerárquica sobre el lenguaje que facilita el análisis sintáctico y la traducción.

- Un noterminal se distingue como el símbolo de inicio. El conjunto de cuerdas que denota es el lenguaje definido por la gramática.

- Las producciones de una gramática especifican la manera en la cual se pueden combinar terminales y noterminales para formar cuerdas. Cada producción consiste de un noterminal seguido por una flecha, seguido por una cuerda de noterminales y terminales.

Como una convención, la producción para el símbolo de inicio se lista primero. Las producciones con el mismo noterminal en la izquierda pueden tener sus lados derechos agrupados, con los lados derechos alternativos separados por el símbolo '|', el cual se puede leer como "o".

GENERADOR DE ANALIZADORES SINTACTICOS YACC

Yacc recibe la especificación del proceso de entrada; esto incluye reglas gramaticales que definen la estructura de un lenguaje, código que se invoca cuando se reconocen estas reglas y una rutina de bajo nivel que realiza el proceso básico de entrada. Yacc genera entonces una función para controlar el proceso de entrada. Esta función constituye el analizador sintáctico o "parser". La rutina de bajo nivel a la que se hace referencia es

la producida por Lex.

En el analizador sintáctico, producido por Yacc intervienen un programa controlador que es una máquina de estados finitos con la capacidad para ver y recordar el siguiente elemento léxico de entrada o "lookahead token", una tabla de parseo formada por dos partes una entrada y una salida. El programa controlador es el mismo para cualquier lenguaje, lo que cambia es únicamente la tabla. El programa de parseo toma del buffer de entrada elementos léxicos, uno a uno. Emplea una pila para almacenar una cuerda de la forma $s0\ X1\ s1\ X2\ s2 \ldots Xm\ sm$, en donde sm se encuentra en el tope. Cada Xi es un símbolo gramatical y cada si es una símbolo llamado estado. Cada símbolo de estado resume la información contenida en la pila bajo él. La combinación del símbolo de estado en el tope de la pila y el símbolo actual de entrada son empleados para accesar la tabla de parseo y determinar la decisión en el parseo de corrimiento o reducción.

La tabla de parseo consiste de dos partes, una función de parseo *ACCION* y una función *GOTO*. El programa controlador del analizador sintáctico se comporta como sigue: determina el estado sm, actualmente en el tope de la pila y con ai, el elemento actual de entrada; consulta entonces $ACCION[sm, ai]$, que es la entrada en la tabla de parseo en la porción de acción para el estado sm y la entrada ai. Esta entrada puede tener uno de cuatro valores:

- corrimiento s donde s es un estado,
- reducción por una producción de la gramática
- aceptación
- error

La función *GOTO* toma un estado y un símbolo gramatical como argumentos y produce un estado. La función *GOTO*, de la tabla de parseo, es la función de transición de un autómata finito determinista. La tabla que se produce es una tabla de parseo LALR o

lookahead-LR (figura 5.2).

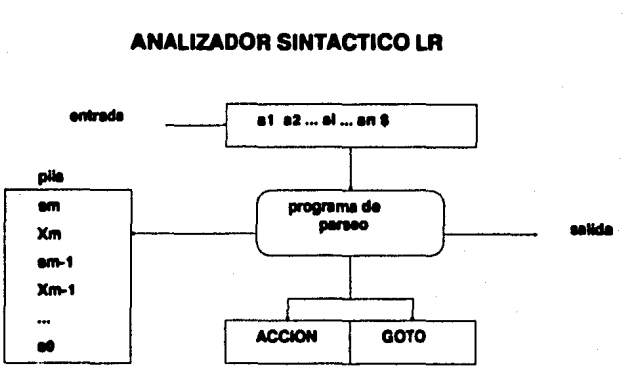

Figura 5.2

Para emplear Yacc se prepara inicialmente una especificación, *c.y* del traductor. Al correrla con Yacc, ésta se transforma en un programa en C, llamado *y.tab.c*. Este programa es una representación de un analizador sintáctico LALR escrito en C, junto con otras rutinas en C que el usuario pueda haber preparado. Al compilar *y.tab.c* obtenemos el programa objeto deseado *y.tab.o* que realiza la traducción especificada por el programa Yacc original. Otros procedimientos requeridos, pueden ser compilados o cargados con *y.tab.c* como cualquier otro programa en C (figura 5.3).

Un programa fuente Yacc está formado por tres secciones:

declaraciones
%%

reglas de traducción

%%

rutinas de soporte en C

Existen dos partes opcionales en la sección de declaraciones. En la primera parte van declaraciones normales de C delimitadas por '%{' y '%}', que puedan ser empleadas por las reglas de traducción o por los procedimientos de la segunda y tercera sección. También en la sección de declaraciones se encuentran las declaraciones de los elementos léxicos de la gramática, los cuales pueden ser empleados en la segunda y tercer sección de la especificación para Yacc. En la segunda sección se encuentran las reglas de traducción. Cada una de estas reglas consiste de una producción gramatical y la acción semántica asociada, que estarían escritas en Yacc como sigue:

<lado izq> : < *alt 1*> { *acción semántica 1* }
: < *alt 2*> { *acción semántica 2* }
...
: < *alt n*> { *acción semántica n* }
;

En una producción para Yacc, un solo carácter *c* se toma como el símbolo terminal *c*, y las cuerdas de letras y números sin apóstrofe, que no se declararon como elementos léxicos, se toman como noterminales. Los lados derechos alternativos se pueden separar por una barra vertical; un punto y coma sigue a cada lado izquierdo con sus alternativas y sus acciones semánticas. El primer lado izquierdo se toma como el símbolo de inicio.

Una acción semántica en Yacc es una secuencia de proposiciones en C. En una acción semántica, el símbolo '$$' se refiere al valor del atributo asociado con el símbolo noterminal de la izquierda, mientras $*i* se refiere al valor asociado con el iésimo símbolo gramatical, terminal o noterminal de la derecha. La acción semántica se realiza cuando

se reduce por medio de la producción asociada, de tal forma que, normalmente, la acción semántica calcula un valor para $$ en términos de las $i.

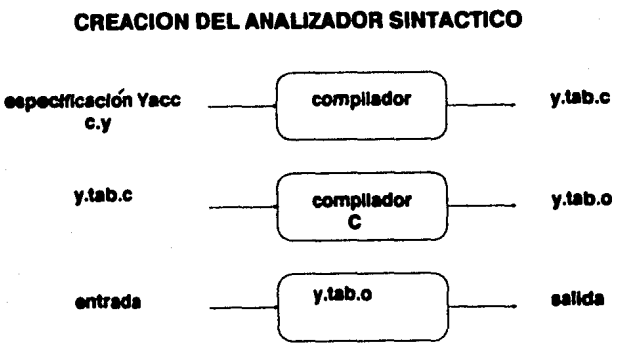

Figura 5.3

La sección de rutinas de soporte en C debe contener una rutina que realice el análisis lexicográfico y cuyo nombre debe de ser *yylex*(). Otros procedimientos, como rutinas para recuperación en caso de error, también se encuentran aquí. El analizador lexicográfico *yylex*() produce parejas formadas por un elemento léxico y el valor del atributo asociado, el cual se pasa al analizador sintáctico a través de una variable definida por Yacc, llamada *yylval*. Este analizador lexicográfico puede ser generado mediante Lex.

MANEJO DE GRAMATICAS AMBIGUAS EN YACC

Al especificar una gramática ambigua, el algoritmo LALR genera conflictos en las acciones del analizador sintáctico. Yacc para resolverlos sigue dos reglas:

- Un conflicto de tipo reducción/reducción se resuelve escogiendo de las producciones conflictivas aquella que aparece primero en la especificación.

- Un conflicto corrimiento/reducción se resuelve en favor del corrimiento.

El conflicto que surge del manejo de la proposición *if ... then ...* e *if ... then ... else ...* se resuelve con la segunda regla.

Yacc provee también un mecanismo más general para resolver los conflictos de tipo corrimiento/reducción. En la sección de declaraciones se pueden asignar precedencias y asociatividades a terminales. La precedencia de los elementos léxicos será en el orden en que aparecen en la sección de declaraciones, los de precedencia más baja van primero, los elementos básicos en la misma declaración tendrán la misma precedencia. Este mecanismo resulta adecuado para el manejo de operadores en las expresiones, los cuales se pueden declarar como asociativos por la izquierda o derecha o bien no-asociativos.

Con esta información, Yacc resuelve los conflictos corrimiento/reducción asignando una precedencia y una asociatividad a cada producción involucrada en el conflicto, así como a cada terminal involucrado en un conflicto. Si es necesario escoger entre hacer un corrimiento a un símbolo de entrada o reducir por una producción, Yacc reduce si la precedencia de la producción es mayor que el símbolo de entrada o si las precedencias son iguales y la asociatividad de la producción es izquierda. En cualquier otro caso, la acción del analizador sintáctico será de corrimiento.

6. INTERPRETE

El intérprete evalúa expresiones provenientes de la terminal o incluidas en el programa del usuario, ejecuta el código intermedio y trata de relacionar el texto del programa fuente con las acciones que deben ocurrir a tiempo de ejecución para poder ejecutar el programa. La parte del traductor que corresponde al intérprete comprende las reglas gramaticales y las acciones semánticas asociadas a ellas que realizan esencialmente estos tres procesos.

Para lograr la integración en un solo traductor de las funciones que realizan el compilador y el intérprete, se desarrolló un mecanismo que permite "dirigir" al traductor hacia determinadas reglas gramaticales. Este mecanismo consiste en la introducción de elementos léxicos o terminales, a los que hemos llamado ESPECIALES, que no forman parte de la gramática de C sino que se generan en forma artificial por el intérprete.

La información que se requiere para compilar un programa está contenida totalmente en las reglas gramaticales y en las acciones semánticas que forman parte del analizador sintáctico. Sin embargo, el intérprete y el depurador, como resultado de ciertos procesos de depuración, reciben adicionalmente, en forma interactiva, requerimientos del usuario que modifican el medio ambiente y hacen necesario un cambio en el flujo del traductor.

Los elementos especiales se generan de diversas formas: en algunos casos, el intérprete los inserta al principio y al final de una expresión o los envía por medio del analizador

lexicográfico, el cual a su vez los pasa al analizador sintáctico como otro elemento léxico; otras veces, el depurador o el intérprete le solicitan a la rutina de bajo nivel que lee los caracteres de entrada, que los genere y envíe al analizador lexicográfico, el cual los envía a su vez al analizador sintáctico.

Al ser invocado el intérprete, se encuentra con un medio ambiente formado por un conjunto de estructuras generadas como producto del preproceso que realizaron sobre el programa fuente, tanto el editor como compilador. Estas estructuras contienen la información que requiere el intérprete para realizar su proceso.

El editor ha almacenado el texto del programa en un arreglo de caracteres el cual tiene asociado un arreglo de apuntadores al primer carácter de cada línea de texto. El compilador ha generado varias tablas de símbolos, una global y otras locales, que contienen las variables con sus atributos asociados, en particular el tipo y la dirección relativa que les corresponderá durante la ejecución del programa. Para las funciones, el compilador ha generado el código intermedio correspondiente incluyendo apuntadores al texto del programa donde se encuentran las expresiones. Bajo este marco de referencia inicia el intérprete sus procesos, teniendo ante sí, como primera tarea, la organización del medio ambiente.

MEDIO AMBIENTE

Las acciones semánticas que manejan el medio ambiente a tiempo de ejecución se encargan de organizar la memoria, asignan y recuperan espacio, generan las ligas entre los nombres empleados en el programa y las localidades físicas, en estas localidades almacenan y recuperan valores, realizan además diversas acciones necesarias para ejecutar las funciones.

ALMACENAMIENTO

Son varios los elementos a los que se les asigna almacenamiento para poder ejecutar un programa. Lo más obvio es el almacenamiento requerido para el código intermedio y para las estructuras de datos, variables y constantes definidas por el usuario. Menos obvio es el espacio requerido para la información acerca de las ligas entre funciones, las variables temporales requeridas para la evaluación de expresiones y para la transmisión de parámetros.

La cantidad de espacio necesario para almacenar una variable está determinada por su tipo. Un tipo de dato elemental como carácter o entero, se almacena en un número entero de "bytes"; para tipos agregados, como los arreglos, este espacio debe ser lo suficientemente grande para contener todas sus componentes, siendo su dirección la de la primera de sus componentes. La disposición de los datos en el espacio de almacenamiento está influenciado por las restricciones en el direccionamiento de la máquina objeto. En nuestro caso, enteros y apuntadores deben estar alineados en direcciones divisibles por 2. El almacenamiento está formado por un área contigua en memoria, en la cual un "byte" es la unidad más pequeña direccionable.

Para todas las variables ya sean globales o automáticas, el compilador registra una dirección relativa, a partir de la cual se encuentra almacenado el valor de la variable. Para efectuar la liga entre el nombre de la variable y la localidad física que le ha sido asignada, el intérprete toma una base y le suma esta dirección relativa para obtener la dirección absoluta.

La asignación de almacenamiento para las tablas de símbolos, el código intermedio, así como para las variables globales se realiza en forma estática a tiempo de compilación. Las variables globales y el almacenamiento dinámico, comparten un área definida como:

char stack[*STACKSIZE*];

en la cual las variables globales ocupan las localidades bajas y las localidades altas se utilizan para el almacenamiento dinámico.

La dirección relativa que registra el compilador para las variables globales es un número positivo y como base el intérprete toma la dirección de inicio de *stack*.

Veamos un ejemplo, supongamos que tenemos una variable global *int x* cuya dirección relativa es *DIR_REL*, tenemos que *stack* corresponde a la direccion inicial del área estática para las variables globales, a *char* $*px$ le asignaremos la dirección absoluta de x, que será:

$$px = stack + DIR\_REL;$$

Para almacenar y recuperar valores de las localidades físicas, es necesario accesar, a partir de su dirección absoluta, el número adecuado de "bytes" que corresponde al tipo de variable. Esta labor la realiza el manejador mediante conversiones o "casts".

Retomando el ejemplo anterior, para almacenar un valor VAL, en la localidad física correspondiente a x, procederíamos de la siguiente manera:

$$*((int\ *)px) = VAL;$$

mediante un "cast" hemos convertido px de un apuntador a *char* a un apuntador a *int* por lo que accesará 4 "bytes" a partir de px, que es el tamaño de los enteros en esta implantación. Cuando el tipo del valor que se quiere accesar es *char*, no es necesario realizar ningún "cast". Para la recuperación de valores se procede en forma análoga.

ALMACENAMIENTO DINAMICO

Un programa en C está formado por un conjunto de definiciones individuales de funciones, las cuales en su forma más sencilla asocian un identificador con una sola proposición. No se permiten definiciones anidadas de funciones. Cualquier función puede ser llamada recursivamente y sus variables locales son "automáticas", en el sentido de que se crean de nuevo con cada invocación.

El método que se utiliza generalmente para asociar los parámetros actuales con los formales es el de llamada por valor. En él los parámetros actuales se evalúan y su valor se pasa a la función. Cuando el nombre de un arreglo aparece como argumento a una función, la dirección del inicio del arreglo es lo que se pasa, los elementos no se copian, en este caso el paso de parámetros es por referencia.

El número de parámetros con que se llama una función es variable, los cuales se van tomando de localidades con direcciones ascendentes, partiendo de una base fija. En la función *printf*, por ejemplo, dado el fomato de escritura, la función va tomando tantos parámetros actuales como hayan sido indicados en el formato.

A cada ejecución del cuerpo de una función nos referiremos como a una activación de la función. Una función es recursiva si una nueva activación puede empezar antes de que una activación anterior de la misma función haya terminado.

La información necesaria para la activación de cada función se maneja en un bloque contiguo de memoria llamado registro de activación, el cual está formado por varios campos. Estos registros se han organizado en forma de pila, para cada activación de una función colocamos en el tope de la pila su registro de activación, al terminar la activación correspondiente eliminamos este registro de la pila.

La pila, para el manejo de registros de activación se inicia en las localidades altas de un área de memoria y va creciendo hacia las localidades bajas. Esta región de memoria $stack[STACKSIZE]$, es la que se comparte con las variables globales que ocupan en forma estática, las localidades bajas de memoria (figura 6.1).

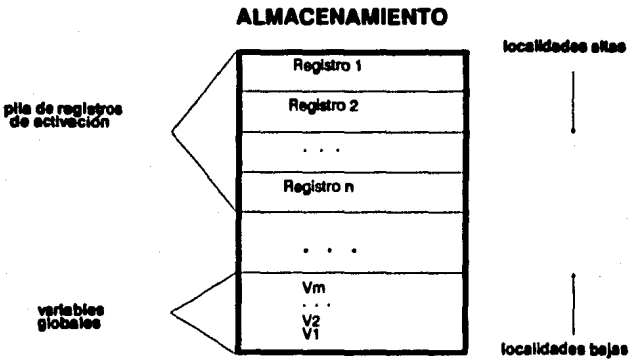

Figura 6.1

Los campos contenidos en el registro de activación son los siguientes:

a).- El campo para los parámetros actuales empleados por la función que realiza la llamada para proveer los parámetros a la función que ha sido invocada.

b).- Un apuntador a la entrada en la tabla de símbolos de la función que ha sido invocada.

c).- Un campo de estado que contiene información acerca del estado del

programa antes de que se realizara la llamada.

d).- Una liga de control que apunta al registro de activación anterior que corresponde a la función que hizo la llamada.

e).- El campo que contiene las variables automáticas o locales que requiere la función para su ejecución.

Los campos b), c) y d), cuyo tamaño es fijo, ocupan la parte central del registro de activación. El campo a), para parámetros formales y cuyo tamaño es variable, ocupa localidades más altas que la liga de control. El campo b) para variables automáticas, que también es variable, ocupa localidades más bajas que la liga de control. Esto permite al compilador registrar las direcciones relativas, tanto para los parámetros formales como para las variables automáticas, tomando como base la liga de control. Las direcciones relativas de los parámetros formales son positivas y crecientes, las de las variables automáticas son negativas y decrecientes. Para hallar la dirección absoluta de los parámetros actuales y de las variables automáticas, el manejador toma como base la dirección de la liga de control y le suma la dirección relativa.

Dado que el número de parámetros con que se llama una función es variable el manejador no puede saber de antemano cuánto espacio reservar para los parámetros actuales. Al encontrar una llamada a una función, el intérprete va almacenando los parámetros en la pila, empezando por el último y terminando por el primero, así la función puede ir accesando tantos parámetros actuales como requiera, partiendo de una base fija.

Para almacenar en este orden los parámetros actuales en el registro de activación, la regla gramatical que reconoce la llamada a una función utiliza recursividad derecha.

```
rlvalue     : ID '(' expL ')'
            ;
expL        :                          /* vacía */
            | exp
            | exp ',' expL
            ;
```

La información que contiene el campo fijo de status del registro de activación es la siguiente (figura 6.2):

- Una palabra que contiene información acerca de la fuente de la cual proviene la expresión que se está evaluando.

- Un apuntador al tope de la pila de cuerdas de caracteres.

- Un apuntador al inicio del código intermedio de esa función.

- El índice de la siguiente instrucción del codigo intermedio que será ejecutada al regresar de la función.

- Un apuntador a la expresión que está siendo evaluada.

El almacenamiento para las cuerdas de caracteres utilizadas localmente se realiza en forma dinámica, implantando una pila definida como:

char str_stack[STRSIZE];

Cada vez que se invoca una función, se guarda la dirección del tope de esta pila, las cuerdas locales se almacenan a partir de allí. Cuando se efectúa el retorno de la

función, se libera este espacio actualizando el tope de la pila al valor que tenía cuando fue invocada la función.

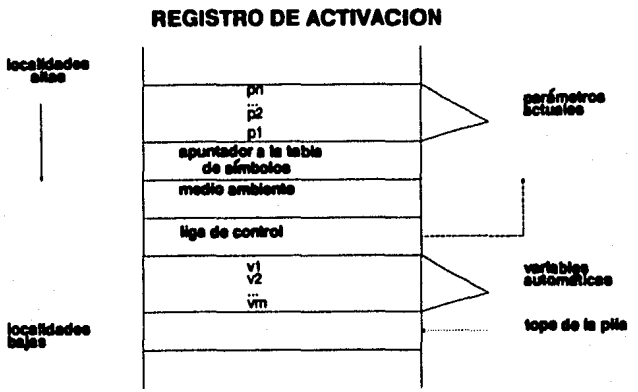

Figura 6.2

El almacenamiento para las variables temporales que se requieren en la evaluación de expresiones se maneja en forma dinámica por medio de una pila formada por descriptores de expresiones. En este caso, el intérprete no se encarga de su administración, sino que emplea la pila que utiliza y administra internamente Yacc.

REGLAS GRAMATICALES ASOCIADAS A FUNCIONES

Una función es una expresión, por lo que las reglas gramaticales que corresponden al

manejo de funciones se comparten por el intérprete y el compilador. Al reconocer una función, el intérprete ejecuta el código intermedio asociado, para ello, realiza ciertas acciones, llamadas prólogo, que preparan el medio ambiente para la activación de la función, otras, conocidas como epílogo, al término de la activación, restauran el medio ambiente al estado prevaleciente antes de la llamada a la función. A continuación presentamos las reglas gramaticales para funciones, así como las que se refieren a la ejecución del código intermedio.

```
nivalue     : ID '(' expL ')' { call(); } code
            ;
code        :                          /* vacía */
            | gen_expL { ret(); }
            ;
gen_expL    : gen_exp
            | gen_expL gen_exp
            ;
gen_exp     : M_E expjr ETX
            | M_IF expjr ETX
            | M_SWITCH expjr ETX
            | M_RETV expjr ETX
            | M_GOTO CONSTANT
            | M_CASE CONSTANT
            | M_DEF CONSTANT
            | M_RET
            ;
```

Como parte de las reglas hemos incluido las acciones semánticas *call*() y *ret*() que corresponden, respectivamente, al prólogo y al epílogo de la función. Al reconocer la

llamada a una función, el intérprete ya ha colocado en la pila de registros de activación los parámetros actuales. La rutina *call*() almacena entonces, en la pila, el estado que impera en el medio ambiente antes de ejecutar la función, para que al efectuar el retorno de la función pueda restaurarlo, esta información ocupa la parte fija del registro de activación. Posteriormente, deja el espacio necesario para el almacenamiento de las variables automáticas y actualiza el apuntador al tope de la pila a este nuevo valor para que en el caso de que aparezca otra llamada a una función pueda almacenar los parámetros actuales a partir de esa localidad. Inicializa también algunas variables como son los apuntadores a la tabla de símbolos local y al código intermedio; el contador de programa, etc. La rutina *ret*() toma del registro de activación la información necesaria para restaurar el medio ambiente.

En la primera regla de este grupo podemos notar el elemento sintáctico *code*, que en el caso del compilador es vacío y en el caso del intérprete está formado por una lista *gen_expL* formada por las operaciones del código interemedio. El intérprete al reconocer una función sabe que tiene que ejecutar también el código intermedio, es decir, tiene que "dirigir" al traductor hacia las reglas gramaticales que lo ejecutan. Para ello utiliza el mecanismo descrito anteriormente y que consiste en el envío de un elemento léxico especial; la función *call*() es la que se encarga de generarlo. Este elemento léxico está formado por el operador de la primera instrucción del código intermedio. De este punto en adelante cada instrucción del código se encarga de enviar el elemento léxico especial que dirige al traductor a la siguiente instrucción de código intermedio, salvo las instrucciones con operador *RET* o *RETV*, que corresponde a la instrucción de retorno de la función. Las componentes terminales *M_E, M_IF, M_SWITCH, M_RETV, M_GOTO, M_CASE, M_DEF y M_RET* forman los elementos léxicos especiales asociados al código intermedio.

Si la llamada es a una función de las que forman parte de la librería de C o de APC, se efectúa un prólogo semejante al descrito anteriormente, lo que cambia es que en este

caso no se tiene un código intermedio, sino que se hace una llamada a la función por medio de un apuntador al código de máquina generado por el compilador de C y se le pasan los parámetros actuales que se encuentran en la pila de registros de activación. El epílogo, al igual que para las otras funciones, restablece el medio ambiente.

7. EXPRESIONES

El lenguaje C posee un conjunto amplio de operadores, si se le compara con otros lenguajes de programación como pueden ser Pascal o Fortran, que provee acceso a la mayor parte de las operaciones que sustenta el hardware. Las variables y constantes son los objetos básicos que se manipulan en las expresiones. Por medio de las declaraciones sabemos qué variables se van a emplear y cuál es su tipo. Los operadores especifican que es lo que se va a hacer con ellas. Las expresiones combinan variables, constantes y operadores para producir nuevos valores.

Como se vió en el Capítulo 3, la evaluación de expresiones se efectúa utilizando el principio de traducción dirigida por sintaxis, en donde traducción significa calcular el valor de la expresión a medida que se van reconociendo los elementos sintácticos. Antes de evaluar una expresión, el compilador ha realizado un preproceso durante el cual se generaron las tablas de símbolos, información que se toma como base para la evaluación. El intérprete, al procesar las expresiones, realiza también el análisis lexicográfico, sintáctico y semántico de las mismas.

El analizador lexicográfico al detectar un elemento léxico regresa como valor de retorno una constante que corresponde al tipo de elemento léxico encontrado. En el caso de los operadores, el valor es diferente para cada uno de ellos. Para cada elemento léxico, salvo los operadores, genera un descriptor que contiene un conjunto de atributos asociados a estos elementos, obteniendo esta información de las tablas de símbolos.

El analizador sintáctico, como vimos, para realizar el proceso de traducción utiliza una pila en la que intervienen símbolos gramaticales. En ella, guiado por la gramática, procede a efectuar las operaciones de corrimiento o reducción de símbolos gramaticales. Al realizar una reducción, las acciones semánticas de la regla gramatical indican la forma en que se calculan los nuevos atributos del símbolo gramatical a partir de los atributos que se encuentran en la pila para los símbolos gramaticales del lado derecho de la regla.

ATRIBUTOS

La evaluación de expresiones está guiada por una gramática libre de contexto en la que se le ha asociado a cada símbolo gramatical un conjunto de atributos. Estos atributos representan diversos aspectos: un tipo, una localidad de memoria, un número, etc. Para las expresiones estos atributos se calculan a partir de los valores de los atributos de los hijos en ese nodo del árbol de parse. Las acciones semánticas del intérprete indican la forma en que se evalúan estos atributos. Veremos a continuación el conjunto de atributos que maneja el intérprete.

Lexema

Asociado a cada elemento léxico se tiene un lexema que es el conjunto de caracteres que forma el elemento léxico. En APC se llega a él en forma indirecta a través de un apuntador a la tabla de símbolos.

Valor-l y Valor-d

Un objeto es una región en la memoria que puede ser examinada y en la cual se puede almacenar un valor. Asociados a cada objeto tenemos dos conceptos: su localización y su valor. En una expresión de asignación, estos conceptos se nos manifiestan de manera aparente:

$$A = B$$

En la expresión anterior el significado será pon el valor de B en la localidad denotada por A. Esto es aún cuando A y B no tienen ninguna marca que las distinga, la posición de B en el lado derecho del simbolo de asignación nos dice que nos estamos refiriendo a su valor. En cuanto a la posición de A a la izquierda, nos manifiesta que se trata de su localización. Por lo tanto, nos referimos al valor de un objeto como su valor-d en donde "d" se refiere a la posición a la derecha del operador de asignación y nos referimos a la localidad del objeto como valor-i, en donde la "i" se refiere a la posición izquierda.

Algunos identificadores o nombres son valores-i, como los nombres de variables de tipo aritmético, estructura, enumeración, unión y apuntador; los nombres de funciones y arreglos no lo son. Algunas operaciones sobre expresiones que no tienen valor-i pueden producir un valor-i. Por ejemplo, el nombre de un arreglo no es un valor-i, sin embargo las referencias a los elementos del arreglo por medio de expesiones con subíndices son valores-i. Dicho de otro modo, uno no puede modificar todo un arreglo mediante una asignación, pero uno si puede modificar elementos individuales. Una expresión, a medida que se compone con otras expresiones y operadores, va adquiriendo o perdiendo su valor-i.

En APC este valor-i está declarado como un apuntador a un objeto de tipo *char*, que corresponde a la mínima unidad de almacenamiento, en este caso un byte. Para accesar

objetos de otro tipo se realiza un "cast" a este apuntador, como se vió en el capítulo anterior.

Tipo

Un tipo es un conjunto de valores y de operaciones, sobre esos valores. Por ejemplo, los valores de un tipo entero consisten de enteros en un rango especificado y las operaciones sobre esos valores consisten de suma, resta, pruebas de igualdad, etc. Los valores para un tipo de punto flotante incluyen números que se representan en forma diferente a los enteros y un conjunto de operaciones diferentes.

C tiene una estructura de tipos con un número potencialmente infinito de tipos. Para empezar tenemos los tipos básicos: *char*, *short*, *int*, *long*, sus versiones sin signo: *uchar*, *ushort*, *unsigned*, *ulong*, y *float* y *double*, y finalmente *struct*, *union* y *enum*. Tenemos además tres modificadores que pueden ser aplicados a los tipos para generar otros: si t es un tipo, podemos potencialmente tener los tipos: apuntador a t, función que regresa t y arreglo de t. Un tipo arbitario en C consiste entonces de un tipo básico y cero o más modificadores.

Supongamos que B es un tipo básico y que *id* es cualquier identificador, entonces la declaración:

B id;

indica que *id* tiene el tipo básico B. A partir de esta declaración podemos ir aplicando los modificadores:

B $*id$;

significa que tenemos un apuntador *id* a un elemento de tipo básico B.

B *id*[ ];

indica que se trata de un arreglo *id* cuyos elementos son de tipo B.

B *id*();

se refiere a una función *id* que regresa un valor de tipo B.

Ahora bien, estos modificadores se pueden ir componiendo, teniendo en cuenta su precedencia : '( )' y '[ ]' tienen más alta precedencia que '*'. Así por ejemplo:

int $*x$[ ];

representa un arreglo de apuntadores a enteros,

int $*f$();

es una función que regresa un apuntador a un entero. Por medio de paréntesis se pueden cambiar estas precedencias, asi por ejemplo:

int$(*x)$[ ];

será un apuntador a un arreglo de enteros,

int$(*f)$();

será un apuntador a una función que regresa un entero.

Al componer estos modificadores hay que recordar que podemos tener arreglos de apuntadores a funciones, mas no arreglos de funciones. No es posible que una función regrese ni una función, ni un arreglo, pero si es posible que regrese un apuntador a un arreglo o un apuntador a una función, esta restricción surge debido a la definición de C que no lo permite.

Como se vió anteriormente, los tipos básicos que se consideraron son únicamente *int* y *char*. Sin embargo, el número máximo de modificadores que permitimos es hasta seis. Para la implantación de tipos se ha empleado un *ushort* de 16 bits, en donde los cuatro bits de la derecha contienen el tipo básico, los bits restantes, divididos en campos de dos bits, contienen ningún modificador o alguno de los tres descritos anteriormente. Estos modificadores se leen de derecha a izquierda, terminando con un campo sin modificador (figura 7.1).

Se utilizó este esquema ya que con cuatro bits se puede hacer referencia a 16 tipos básicos. El campo para modificadores se definió de dos bits ya que son cuatro los modificadores posibles, incluyendo el nulo, se requieren 2 bits para codificar estas cuatro opciones. En una palabra de 16 bits caben el tipo básico y seis modificadores, lo que permite el manejo de tipos complejos.

REPRESENTACION INTERNA DE TIPOS

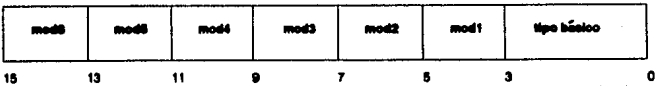

Figura 7.1

Hemos codificado los tipos básicos, como:

CHAT 0001 para *char*
INTT 0010 para *int*

y los modificadores:

NOP 00 ningún modificador
PTR 01 apuntador a
FUN 10 función que regresa
ARR 11 arreglo de

Por ejemplo:

int(*x[ ])();

que es un arreglo de apuntadores a funciones que regresan enteros, será representado como:

NOP NOP NOP FUN PTR ARR INTT

y estaría codificado como:

0000001001110010

Además de ahorrar espacio, este esquema permite llevar cuenta de los modificadores que aparecen en un tipo y, mediante corrimientos, substituciones e inserciones de modificadores, podemos obtener los tipos resultantes de la aplicación de ciertos operadores, como son '*', '[ ]' y '&'. Posteriormente, al presentar cada uno de estos operadores se

verá en más detalle su funcionamiento. Para verificar la equivalencia de tipos de comparan las representaciones de tipo respectivas. En el caso de que el último modificador de cada tipo sea un apuntador, es necesario verificar también el tamaño del objeto al cual apunta para determinar la equivalencia. De allí que como parte de los atributos de una expresión, se haya incluido el número de bytes necesarios para representar su valor.

Tamaño

El tamaño, cuando se refiere a una expresión, está dado por la cantidad de memoria, medida en unidades de almacenamiento, necesarias para almacenar el valor de la expresión. En nuestro caso estaremos hablando de bytes. Cuando se refiere a un arreglo, es el tamaño total del arreglo. En la impantación de APC para la CODATA, las variables de tipo *int* ocupan 4 bytes, los apuntadores también requieren 4 bytes y las variables de tipo *char* ocupan un byte.

Dimensión

Este atributo se utiliza únicamente para arreglos y nos indica la dimensión del arreglo a la cual se está haciendo referencia. En arreglos parte de uno, para otro tipo de variables es cero.

Asociado a expresiones se tiene un descriptor que contiene los atributos anteriores:

```
typedef struct {
        symT *esym;
```

```
        char *eival;
        long erval;
        ushort eatt;
        ushort esize;
        utiny elevel;
} expT;
```

en donde:

- *esym* es un apuntador a la tabla de símbolos, en donde se encuentra un apuntador al lugar en el cual está almacenado el lexema correspondiente al elemento léxico.

- *eival* es un apuntador con el valor-i de la expresión.

- *erval* es el valor-d de la expresión.

- *eatt* contiene el tipo básico con sus modificadores.

- *esize* es el tamaño de la expresión.

- *elevel*, cuando se trata de arreglos, contiene la dimensión a la cual se está haciendo referencia.

ACCIONES SEMANTICAS ASOCIADAS A EXPRESIONES

La evaluación de una expresión consiste en evaluar los atributos asociados a los elementos gramaticales que están contenidos en el descriptor de expresiones. Los atributos

se evalúan de abajo hacia arriba por el intérprete a medida que la cuerda de entrada se va analizando. Durante el proceso de evaluación se realiza el análisis semántico, en el cual se verifica que cada operador tenga el tipo de operandos permitidos por la especificación del lenguaje fuente.

ACCIONES SEMANTICAS ASOCIADAS A ELEMENTOS LEXICOS

Empezaremos por considerar las acciones semánticas que realizan el analizador lexicográfico y posteriormente el analizador sintáctico al reconocer un elemento léxico.

Además de detectar elementos léxicos, como constantes, cuerdas, identificadores, palabras reservadas y operadores, APC incluye otros dos elementos léxicos. Por un lado, se encuentran los elementos léxicos especiales, generados por el sistema, y por otro lado, se encuentran, como otro elemento léxico, las funciones pertenecientes a la librería de C y a la librería de APC.

Acciones Semánticas del Analizador Lexicográfico

Para cada elemento léxico, reconocido por, el analizador lexicográfico, regresa un valor que lo identifica e inicializa en el descriptor de expresiones, los valores de algunos atributos. La variable *yylval* predefinida en LEX, ha sido declarada como un descriptor de expresión de tipo *expT*.

A continuación presentamos una lista de los elementos léxicos que se reconocen, y los atributos que se inicializan:

- Cuerdas: regresa el valor STRING. En tamaño almacena la longitud de la cuerda y en valor-i un apuntador al lexema.

- Constantes: regresa el valor CONSTANT. Considera variables de tipo decimal, octal, hexadecimal y de caracteres; calcula su valor y lo almacena en valor-d, tipo real.

- Palabras Reservadas: regresa un valor distinto para cada una de ellas.

- Operadores: si está formado por un solo carácter, regresa como valor el carácter mismo. En caso de que esté formado por dos caracteres, regresa un valor distinto y específico para cada uno de estos operadores.

- Identificadores: regresa el valor ID y un apuntador a la tabla de símbolos.

- Funciones de la librería de APC o de C: regresan el valor BLTIN y en valor-i un apuntador al código de máquina que ejecuta la función.

- Elementos léxicos especiales: cuando forman parte del texto como *STX* y *ETX*, regresan, respectivamente el valor STX y ETX. Cuando los envía el intérprete se regresa como valor el indicado por el intérprete.

Acciones Semánticas del Analizador Sintáctico

Cuando el terminal es un identificador, copia de la tabla de símbolos al descriptor de tipo *exp*T, asociado a este elemento, el tipo y tamaño. Los valores de atributos que no se mencionan en forma explícita se inicializan, si se trata de un apuntador con un

valor *NULL* y en el resto de los casos con cero. Analizando el tipo y empleando el primer modificador, o sea, el que se encuentra a la extrema derecha, realiza las acciones siguientes:

- Si se trata de PTR (apuntador) o de NOP (sin modificador), en valor-i almacena la dirección de la variable, siguiendo el procedimiento indicado en la sección de asignación de memoria. En valor-d almacena el valor de la variable cuya dirección se encuentra en valor-i.

- Si es ARR (arreglo), en valor-i almacena la dirección inicial del arreglo e inicializa dimensión con uno.

Al detectar una constante, almacena en valor-d su valor y en tamaño el número de bytes necesarios para almacenar este valor. Si la constante es una cuerda, la instala, siguiendo el procedimiento descrito en asignación de memoria, y en valor-i guarda un apuntador a este lugar.

ACCIONES SEMANTICAS ASOCIADAS A OPERADORES

Los atributos de la expresión resultante difieren de aquellos contenidos en los operandos. Se realizan sobre todo cambios en el valor-d, en otros casos si los dos operandos tenían valor-i, al aplicarles un operador, éste se pierde, indicándose por medio de un valor nulo *NULL*. Algunos operadores, como '*' y '&', al ser aplicados a un operando además causan un cambio en el tipo del operador. Para algunas operaciones, el tipo del resultado es *int*, lo que equivale a una representación interna, de la forma:

NOP NOP NOP NOP NOP NOP INTT.

OPERADORES SOBRECARGADOS

En C existen algunos símbolos matemáticos sobrecargados. El operador '*', por ejemplo, en la expresión $A * B$, tiene diferentes significados cuando A y B son operandos de tipo entero u operandos en punto flotante. Los símbolos aditivos '+' y '−', además de referirse a operaciones de enteros o en punto flotante pueden referirse a operaciones aritméticas de apuntadores cuando alguno de los operandos es un apuntador.

Un símbolo sobrecargado es aquel que tiene diferentes significados dependiendo de su contexto. Este problema se resuelve cuando se puede determinar un significado único para una ocurrencia de un símbolo sobrecargado. Cuando los símbolos sobrecargados son operadores aritméticos, basta con conocer el tipo de los operandos para resolver el significado de la operación.

ARITMETICA DE APUNTADORES

Las operaciones aritméticas de apuntadores tienen un significado particular. Si p es una expresión del tipo apuntador a t e i es un valor entero, entonces la expresión:

$$p + i$$

se define como un apuntador al i-ésimo objeto de tipo t que se encuentra después de aquel al cual apunta p. Siendo p un apuntador, su valor se refiere a una dirección, por lo que para encontrar la nueva dirección es necesario multiplicar i por el tamaño de t y sumárselo a p. El tamaño de p equivale a la aplicación del operador *sizeof* a p. Veamos un ejemplo, supongamos que tenemos un arreglo de enteros b cuya dirección inicial es DIR y p un apuntador a enteros:

```
int b[10], *p;
p = &b[0];
p = p + 3;
```

la nueva dirección contenida en *p* será;

```
DIR + 3 * INTSIZE
```

en donde el tamaño del entero INTSIZE depende de la computadora en cuestión.

CONVERSIONES DE TIPO

Se dice que la conversión de un tipo a otro es implícita si se realiza en forma automática por el compilador, también se le llama coersión. Se dice que la conversión es explícita si el programador necesita escribir algo para lograr la conversión. En C existen varios casos en los cuales los valores de un tipo se convierten en valores de otro tipo.

Casts

Un "cast" se puede emplear para, en forma explícita, convertir un valor a otro tipo. En general, cualquier apuntador a un tipo *p* se puede convertir a un apuntador a un tipo *q*. Existe, sin embargo, el peligro de que se pierda información, dependiendo de los tamaños relativos de los tipos *p* y *q*, causados por un ajuste en la alineación de memoria.

Operaciones

Un operando se puede convertir en forma implícita a otro tipo como preparación a

la realización de alguna operación aritmética lógica. Estas conversiones son de dos clases: primero, los valores aritméticos de un tipo más reducido se promueven al tipo más amplio, así por ejemplo, el tipo *char* se amplía al tipo *int*. Segundo, referencias que son de tipo apuntador, como arreglos y funciones, se convierten efectivamente en apuntadores a arreglos y funciones.

Así por ejemplo, si tenemos un operando:

int(*a[ ])();

la representacion interna del tipo es:

NOP NOP NOP FUN PTR ARR INTT

el tipo que se obtiene por la conversión implicita efectuada será:

NOP NOP NOP FUN PTR PTR INTT

que es un apuntador al primer elemento del arreglo.

En forma semejante, para funciones:

int *f*();

la representación interna del tipo es:

NOP NOP NOP NOP NOP FUN INTT

el tipo que se obtiene por la conversión implicita efectuada será:

por eso si tenemos las declaraciones siguientes: *char a*[10]; o *int f*(); el empleo de *&a* y *&f* causa una advertencia en varios compiladores ya que, de hecho, *a* y *f* son apuntadores a caracteres y funciones, respectivamente.

Por la implantación de tipos que se hizo, estas conversiones realizadas a los tipos de los operandos se efectuán de manera muy sencilla mediante la aplicación de máscaras y operaciones de corrimiento al tipo.

Asignación

Un objeto de un tipo se puede asignar a una localidad de otro tipo ocasionando con ello una conversión implícita. Usualmente, las expresiones a la izquierda y derecha del operador de asignación son las mismas. En el caso de que no lo sean, se hace un esfuerzo por convertir el valor del lado derecho al tipo del valor del lado izquierdo. En algunas ocasiones, se puede realizar sin ningún problema, en otros se genera una "advertencia" y en otros se indica un "error" y no se realiza la operación.

Argumentos de Funciones

Un argumento actual de una función se puede convertir implícitamente a otro tipo antes de efectuar la llamada a la función. En la implantación del sistema todos los parámetros formales son de tipo *int*, por lo que si un parámetro actual es de tipo *char* se promueve a uno de tipo *int*. Si el parámetro actual es un apuntador no existe problema, ya que en la presente implantación el tamaño del tipo *int* es igual al del tipo apuntador.

Valor de Regreso de una Función

El valor de regreso de una función se puede convertir a otro tipo antes de efectuar el regreso de una función. Para este caso se siguen los lineamientos expuestos en el punto anterior.

OPERADORES UNARIOS

Entre estos operadores tenemos los prefijos, operadores postfijos y los operadores *cast* y *sizeof*. Los operadores unarios tienen una precedencia mayor que los binarios y ternarios, los operadores postfijos tienen una precedencia más alta que los prefijos.

Menos Unario

El operador prefijo '—' calcula la negación aritmética de su operando, el cual debe ser de tipo aritmético. El intérprete verifica que se trate de un tipo básico sin modificadores, de lo contrario regresa "error". La expresión que resulta no tiene valor-i.

Negación Lógica

El operador prefijo '!' calcula la negación lógica de su argumento. El resultado es 1 si el operando es diferente de cero y 0 si es igual a cero. El tipo de la expresión que resulta es *int* y no tiene valor-i.

Negación Bit a Bit

El operador prefijo '~' calcula la negación bit a bit del operando que debe ser de tipo aritmético, de lo contrario regresa "error". El resultado no tiene valor-i.

Operador de Dirección

El operador prefijo '*&*' regresa un apuntador a su operando, el cual debe tener un valor-i o sea que valor-i != NULL. Si el tipo del operando es *t*, entonces el tipo del resultado es apuntador a *t*. Por ejemplo, si teníamos un operando:

int * *i*;

con representación interna del tipo:

NOP NOP NOP NOP NOP PTR INTT

el tipo del resultado sería:

NOP NOP NOP NOP PTR PTR INTT

Cuando el operando es de tipo arreglo o función, el intérprete emite un "advertencia", ya que al emplear operandos de tipo referencia, como son los arreglos y funciones, se realiza efectivamente, en forma implícita, una conversión a "apuntador al primer elemento de un arreglo" y "apuntador a función", respectivamente.

El tamaño de la expresión resultante es igual al número de bytes que requiere un apuntador para su representación. En esta implantación es de 4. Así, si se tiene un operando de tipo *char c*, su tamaño es de 1 y al aplicarle el operador '*' se obtiene un tipo *char* * *c*, cuyo tamaño es 4.

Indirección

El operador prefijo '*' realiza una indirección a través de un apuntador. Los operadores

'&' y '*' son el inverso uno de otro. El operando al cual se le aplica el operador debe ser de tipo apuntador. El resultado es un valor-i que se refiere al objeto apuntado por el operando, es decir, si el tipo del operando es apuntador a *t*, el tipo del resultado es *t*.

Cuando el operando es de tipo arreglo o función, como preparación a la aplicación de este operador, se realiza una conversión implícita, obteniendo con ello un operando del tipo adecuado. Tomando el ejemplo:

int(*a[ ])();

cuya representacion interna del tipo es:

NOP NOP NOP FUN PTR ARR INTT

con la conversión implicita, como preparacion a la aplicación del operador '*', obtenemos:

NOP NOP NOP FUN PTR PTR INTT

que es del tipo adecuado. Al aplicar '*', el tipo resultante será:

NOP NOP NOP NOP FUN PTR INTT

Además del cambio en el tipo del resultado, el tamaño de la expresión resultante también se modifica. Si anteriormente teníamos que, por tratarse de un apuntador, su tamaño era PTRSIZE ahora lo que se debe indicar es el tamaño del objeto al cual apunta. Cuando el operando es un arreglo, también se modifica el nivel, que indica la dimensión a la cual estamos haciendo referencia.

Si tenemos, por ejemplo, un operando:

int a[5][3];

cuyo tipo, tamaño y nivel son, respectivamente:

NOP NOP NOP NOP ARR ARR INTT, 5 * 3 * INTSIZE, 1

por la conversión implicita obtenemos:

NOP NOP NOP NOP ARR PTR INTT, PTRSIZE, 1

al aplicar '*' obtenemos:

NOP NOP NOP NOP NOP ARR INTT, 3 * INTSIZE, 2

Operadores de Preincremento y Predecremento

El operador '++' realiza el preincremento y el operador '--' el predecremento del operando, el cual debe ser un valor-i y de tipo escalar. La constante 1 se le suma o resta, respectivamente, al operando y el resultado se almacena en el valor-i. El resultado es el valor incrementado o decrementado del operando, no es un valor-i. Se conserva el mismo tipo del operando. Se realizan las conversiones usuales de la asignación, si el operando es un apuntador se siguen los lineamientos establecidos anteriormente para la suma o resta de constantes a un apuntador.

Operadores de Postincremento y Postdecremento

El operador '++' realiza el postincremento, el operador '--' el postdecremento del

operando, el cual debe ser un valor-i y de tipo escalar. La constante 1 se le suma o resta, respectivamente, al operando y el resultado se almacena en el valor-i. El resultado es el valor del operando antes de haber sido incrementado o decrementado, no es un valor-i, conserva el mismo tipo del operando. Se realizan las conversiones usuales de la asignación. Si el operando es un apuntador, se siguen los lineamientos establecidos anteriormente para la suma o resta de constantes a un apuntador.

Operador Cast

El operador *cast* hace que el tipo del operando se convierta al tipo indicado entre paréntesis. Este tipo de operación se ha restringido y únicamente opera para tipos básicos sin modificadores. El resultado de la operación es el valor y tipo del operando convertido.

Operador Sizeof

El operador *sizeof* se emplea para obtener el tamaño de un tipo o de un objeto. Como en el caso del operador *cast*, los tipos se han restringido únicamente a los básicos sin modificadores. Por tamaño del objeto nos referimos a la cantidad de memoria, expresada en bytes, necesaria para almacenar el objeto dado. Este operador no invoca ninguna de las conversiones que se aplican a objetos de tipo referencia, como arreglos, es decir, un arreglo no se convierte en "un apuntador a", por lo que al aplicar el operador *sizeof* a un arreglo se obtiene el número total de bytes que ocupa el arreglo. El resultado es de tipo *int* y no es un valor-i.

OPERADORES BINARIOS

Una expresión con un operador binario consiste de dos expresiones separadas por un

operador binario. Consideraremos en esta sección los operadores aditivos, los multiplicativos, los relacionales, los bit a bit y los de corrimiento. Todos estos operandos son asociativos por la izquierda y antes de realizar la operación se evalúan los dos operandos. El orden de evaluación de los operandos no está determinado.

Operadores Aditivos

El operador binario '+' denota adición, el operador '−' denota substracción. Se realizan las conversiones binarias usuales sobre los operandos, los cuales deben ser los dos de tipo aritmético o bien uno de tipo apuntador y el otro de tipo entero. Para la substracción, los dos operandos pueden ser apuntadores del mismo tipo. Al reconocer el intérprete el tipo de los operandos realiza la operación apropiada: suma o resta aritmética o bien, aritmética de apuntadores. El resultado no es un valor-i.

Operadores Multiplicativos

Tenemos tres operadores con la misma precedencia, '*' denota multiplicación, '/' denota división y '%' denota residuo. Para la multiplicación y la división, los operandos pueden ser de tipo aritmético y para el residuo de tipo entero. Se realizan las conversiones binarias usuales y el tipo del resultado es igual al de los operandos convertidos. El resultado no es un valor-i. En la división y el residuo, si el segundo operando es igual a cero se produce un "error".

Operadores de Corrimiento

El operador binario '<<' indica un corrimiento a la izquierda y el operador binario '>>' indica un corrimiento a la derecha. Los operandos deben ser de tipo entero. El resultado no es un valor-i.

Operadores Relacionales

Tenemos los operadores '<' menor que, '<=' menor o igual que, '>' mayor que, '>=' mayor o igual que, '==' igual y '! =' diferente. Los operandos pueden ser ambos de tipo aritmético o ambos apuntadores del mismo tipo. Se realizan las conversiones para operadores binarios siendo el resultado de tipo *int*, no es un valor-i. El valor-d de la expresión es 1 si se cumple la relación y 0 si no se cumple.

Operadores Bit a Bit

El operador '&' denota la función bit a bit "y", el operador '|' denota la funcion bit a bit "o" y el operador '^ ' denota la función "o-excluyente". Los operandos deben ser los dos de tipo entero. Se realizan las conversiones binarias usuales y el tipo del resultado es el de los operandos convertidos sin ser un valor-i.

Expresiones Lógicas

Una expresión lógica consiste de dos expresiones separadas por los operadores lógicos '&&' y '| |'. Estas expresiones tienen la característica de que, cuando el primer operando provee suficiente información para determinar el resultado de la operación, el segundo operando no se evalúa. Los operandos pueden ser de tipo aritmético o apuntadores del mismo o diferente tipo. El resultado no es un valor-i. El valor-d es igual a 1 o 0 y es de tipo *int*.

El operador lógico '&&' se refiere al "y lógico". El operando izquierdo se evalúa siempre y si es igual a cero, entonces ya no se evalúa el operando derecho y el resultado es igual a cero. Si el operando izquierdo es diferente a cero, entonces se evalúa el operando derecho, si éste es igual a cero el resultado será 0 y de otra forma será igual a 1.

El operador lógico '| |' se refiere al "o lógico". El operando izquierdo de este operador se evalúa siempre, si es diferente a cero, entonces ya no se evalúa el operando derecho y el resultado es igual a 1. Si el operando derecho es igual a cero, el operando derecho se evalúa y si es diferente a cero el resultado será 1 y si es igual a cero el valor del resultado será 0.

El intérprete, aunque conozca la veracidad o falsedad de la expresión lógica, tiene que continuar con el análisis sintáctico de la expresión, pero sin evaluarla, esta actividad la realiza con el fin de no desincronizarse. Para manejar está situación emplea una bandera de evaluación que prende o apaga.

Expresiones Condicionales

Una expresión condicional consiste de tres expresiones, con la primera y segunda expresión separadas por '?' y por ':' la segunda y tercera expresión. El primer operando se evalúa, si es distinto a cero, se evalúa la segunda expresión y si es igual a cero se evalúa la tercera. El resultado no es un valor-i. Estas expresiones son asociativas por la derecha. En forma similar a las expresiones lógicas, el intérprete continúa con la interpretación de la expresión condicional prendiendo o apagando la bandera de evaluación.

Expresiones de Asignación

Todos los operadores de asignación son asociativos por la derecha y con la misma precedencia. Los operadores de asignación requieren de un valor-i como su operando izquierdo, en caso contrario de produce un "error". El resultado de la operación no es un valor-i. El operador de asignación puede ser un simple '=', con lo que se indica que el valor del operando derecho se almacena en el operando izquierdo o bien compuesto, en cuyo caso se tiene una expresión del tipo $e1\ op = e2$, equivalente a $e1 = e1\ op\ e2$, con

la ventaja de que el operando *e1* solamente se evalúa una sola vez. Cualquier operador binario aritmético, bit a bit, o de corrimiento puede aparecer como parte del operador de asignación compuesto. El intérprete lo supone como un solo elemento léxico sin espacios intermedios.

Los dos operandos pueden ser o los de cualquier tipo aritmético o apuntadores al mismo tipo de objeto, si no sucede así se emite una "advertencia" y la operación se lleva a cabo. Es válido asignar la constante 0 a cualquier operando del tipo apuntador. El nombre de un arreglo o función no puede aparecer como operando izquierdo puesto que no tiene un valor-i.

En las expresiones con operadores de asignación compuestos, primero se evalúan los dos operandos, la operación indicada se aplica a los dos operandos procediendo con las conversiones usuales, para luego almacenar el resultado de esta operación en el valor-i del operando izquierdo, realizando las conversiones usuales de asignación.

Expresiones Coma

Dos expresiones separadas por el operador ',' forman una expresión coma. El operando izquierdo se evalúa primero, descartando su valor. Se evalúa entonces el operando derecho, siendo éste el valor y tipo del resultado. El resultado no es un valor-i.

Expresiones con Subíndices

Son expresiones de la forma *e1*[*e2*] empleadas para referirse a subíndices de arreglos. La primer expresión debe ser del tipo apuntador y la segunda de tipo entero. El resultado es un valor-i. Esta expresión es equivalente a $*((e1) + (e2))$. Al recibir el intérprete una expresión con subíndices la descompone en estas dos operaciones.

Cuando se trata de un arreglo, el intérprete tiene registrada la dimensión a la cual se está haciendo referencia y por medio del descriptor del arreglo, conoce el tamaño del objeto al cual hace referencia el, pudiendo así realizar, en forma adecuada, la aritmética de apuntadores. Considérese el ejemplo:

int a[4][2][5];

con representación interna

NOP NOP NOP ARR ARR ARR INTT

al realizar la conversión usual para arreglos obtenemos:

NOP NOP NOP ARR ARR PTR INTT

En este caso la expresión *e1* apunta a un objeto cuyo tamaño es 2 * 5 * INTSIZE. Si la dirección del inicio del arreglo es DIR entonces, la dirección a la cual hace referencia *a*[3] es DIR + 3 * 2 * 5 * INTSIZE, en donde *e2* tiene el valor 3. En este caso *a*[3] no es un valor-i por lo que no puede ser empleado del lado izquierdo de una expresión de asignación.

GRAMÁTICA

El traductor, desde el momento en que inicia el proceso de traducción, ya sea el referente al compilador o al intérprete, se "dirige" hacia las reglas gramaticales adecuadas, ésto se logra mediante el empleo de elementos léxicos especiales. Las reglas gramaticales que aparecen a continuación son las que se reducen a *c_pro*, que es el elemento de inicio

de la gramática.

```
c_pro :                          /* vacío */
      | sc  Ddec  ';'  c_pro
      | fun_def  c_pro
      | usr_exp
      ;
```

Ahora bien, la primera regla que es vacía, puede corresponder a cualquiera de los dos procesos, la segunda y tercera reglas se refieren a un programa en C formado por declaraciones y definiciones de funciones. La cuarta regla se refiere a la expresión indicada por el usuario desde la terminal y debe ser evaluada por el intérprete. Para dirigir al traductor hacia la cuarta regla, el intérprete inserta, al principio y al final de esta expresión, los elementos léxicos especiales *STX* y *ETX*, así, desde que el traductor reconoce el primer elemento léxico *STX* sabe que se trata de un proceso referente al intérprete y entonces evalúa la expresión. Al reconocer el elemento *STX* sabe que el proceso ha terminado.

```
usr_exp :   STX  expjr  ETX  ;
```

Se vió que, como parte del código intermedio, aparecen expresiones para evaluar, pués bien, la presencia de *STX* al principio de una expresión nos indica que proviene del usuario y la ausencia de ella indica que se trata de una expresión que forma parte del programa fuente del usuario.

8. IMPLANTACION DE LAS CAPACIDADES DE DEPURACION

Estas capacidades se implantaron en forma adicional, ya que no se obtienen directamente del intérprete. Algunas de estas capacidades trabajan en forma estrecha con el editor, en el capítulo referente a la interface con el usuario se muestra su utilización.

CAPACIDAD DE SEGUIMIENTO DEL PROGRAMA

La implantación de la capacidad de seguimiento del programa se realizó generando las funciones *trace*() y *untrace*(), Estas funciones pertenecen al grupo de las BLTIN y forman parte de la librería de APC.

Para poder seguir la ejecución de un programa es necesario contar con una correspondencia entre el programa fuente y el código intermedio, para que, a medida que se ejecuta el código intermedio, sea posible señalar en el texto del programa la parte correspondiente que se está ejecutando. Esta correspondencia está dada por medio de los apuntadores a expresiones que forman parte de las operaciones del código intermedio.

Al recibir el comando de seguimiento ya sea a través del intérprete o del editor, se

invoca la función *trace*() la cual simplemente prende una bandera. El intérprete, al ejecutar el código intermedio, cada vez que se encuentra con una expresión a evaluar consulta esta bandera y si está prendida le informa al editor para que despliegue, en la ventana de edición, la porción de texto que contiene la expresión que se está evaluando y posicione el cursor bajo la operación que se está realizando.

La función *untrace*() tiene el efecto inverso y apaga la bandera. Estas funciones forman parte del conjunto de funciones *BLTIN*.

CAPACIDAD DE DESPLIEGUE

Esta capacidad es la que muestra las variables cuyo valor se modifica en el transcurso de la ejecución, al hablar de valor nos estaremos refiriendo al atributo valor-d de la variable. Para implantar esta capacidad se programaron las funciones *show*() y *unshow*(), que como en el caso anterior, pertenecen al grupo de funciones *BLTIN* y forman parte de la librería de APC.

Cuando el intérprete o el editor reciben el comando de despliegue, invocan la función *show*() la cual es muy sencilla, ya que únicamente enciende una bandera. Dentro del intérprete, las modificaciones a variables se efectúan durante la aplicación del operador de asignación. El intérprete cada vez que aplica este operador consulta la bandera, si está prendida, despliega en la ventana de ejecución la variable y el nuevo valor asignado. Esta información la obtiene del descriptor asociado a la variable: por medio del apuntador a la tabla de símbolos tiene acceso al lexema de la variable, el valor-d le proporciona el valor actual y el tipo le indica la forma en que debe desplegar el valor. La función *unshow*() inhibe esta capacidad apagando la bandera de despliegue.

CAPACIDAD DE SUSPENSIÓN DEL PROGRAMA

La suspensión de un programa es un proceso más complicado que involucra, en forma estrecha, al editor. Requiere del empleo de varios elementos léxicos especiales e introduce un nivel más en las expresiones. Hasta este momento se tenían dos niveles: el que corresponde a la expresión del usuario y el que corresponde a las expresiones del programa fuente, en el caso de un punto de suspensión se genera un tercer nivel correspondiente a las expresiones indicadas por el usuario durante la suspensión del programa. Cada uno de estos niveles determina la fuente de la cual el intérprete toma los elementos léxicos.

Los puntos de suspensión se realizan sobre los operadores por medio de un comando al editor. Al recibirlo, el editor prende el bit más significativo del operador. Este proceso se puede realizar sobre múltiples operadores.

Existe una rutina de bajo nivel en APC cuya función es leer los caracteres de la cuerda de entrada y pasarlos al analizador léxico. Pués bien, al detectar un carácter con el bit más significativo prendido, lo apaga para pasarlo al analizador léxico y prende la bandera de suspensión de programa. Cada vez que el analizador sintáctico encuentra un operador consulta la bandera de suspensión y si está prendida envía al analizador léxico un elemento léxico especial.

Para cada conjunto de operadores con la misma precedencia se ha definido un elemento léxico especial que se denota como bn, en donde la n corresponde al nivel de precedencia del operador correspondiente, se tienen 15 niveles de precedencia. La asociatividad de estos elementos es la misma que la del operador correspondiente.

El punto de suspensión para los operadores multiplicativos es, por ejemplo, b3 y ha sido declarado como terminal de la gramática, tiene la misma precedencia que '*', '/'

y '%' y es asociativo por la izquierda.

```
% token < ival >          b3
% left '*' '/' '%'        b3
```

El analizador sintáctico, al detectar uno de estos elementos léxicos, se dirige hacia las reglas gramaticales que incluyen puntos de suspensión. Al suspender un programa, el intérprete puede recibir una lista de expresiones, las cuales utiliza el usuario para depurar el programa. En la siguiente regla gramatical se envía un elemento léxico *BB* especial que actúa como delimitador para esta lista de expresiones:

```
exp          : b9 exp { yychar = BB ; } ;
```

tenemos entonces como válida una expresión seguida por una lista de expresiones de usuario que se encuentra delimitada por *BB* y *EB*.

```
exp          : exp BB  { exec_stop(); }
               usr_expL EB  { end_stop(); }
             ;
```

Al suspender un programa se efectúa un cambio en el medio ambiente, por lo que se tiene que almacenar el estado prevaleciente, en este sentido es parecido a la llamada a una función solamente que no se ejecuta un código intermedio, ni es necesario generar un registro de activación. La acción semántica *exec_stop*() actúa como prólogo antes de pasar a recibir la lista de expresiones del usuario, *end_stop*() actúa como epílogo. Dentro de la lista de expresiones indicadas por el usuario en una suspensión pueden intervenir, desde luego, llamadas a funciones, en este caso, sí se realiza el proceso completo descrito anteriormente para el manejo de funciones.

9. CONCLUSIONES

El contar con un sistema como APC, que reúne en forma integrada varias de las herramientas empleadas durante el ciclo de programación, definitivamente facilita esta tarea, reduciendo el tiempo dedicado a pruebas y depuración. Como elemento didáctico, para la enseñanza del lenguaje de programación C, el sistema puede tener utilidad.

La inclusión de un simple editor de texto permite al usuario trabajar con naturalidad, en la forma en que está acostumbrado, en contraposición con los editores dirigidos por sintaxis cuyo uso no se ha generalizado a pesar de haber sido propuestos hace ya varios años.

El dotar al intérprete/depurador con un lenguaje de comandos formado por expresiones de C, es una buena idea que paulatinamente ha demostrado su potencialidad. Utilizar el esquema de traducción dirigida por sintaxis, aún para el intérprete, facilitó enormemente la organización y programación de todo el sistema.

La realización de las dos tesis relativas a APC implicó un considerable esfuerzo de programación, el cual se realizó en el mismo lenguaje C. El objetivo no era hacer un producto acabado, sino un prototipo que demostrara la viabilidad de algunas ideas y que sirvieran de partida para un desarrollo posterior más completo. Este desarrollo posterior contemplaría los siguientes puntos, algunos de los cuales han sido sugeridos

por el uso limitado que se ha hecho del sistema:

- Extensión del sistema para manejar el lenguaje C completo, incluyendo el preprocesador.

- Cache de expresiones precompiladas para hacer más eficiente su evaluación.

- Inclusión a la librería de muchas otras funciones que sirvan como apoyo en la depuración de programas.

- Inclusión de funciones orientadas hacia el aprendizaje con acceso a las estructuras creadas por el compilador, como son las tablas de símbolos y el código intermedio, así como a las estructuras de almacenamiento estático y dinámico manejadas por el intérprete.

BIBLIOGRAFIA

AHO79 **Aho A.V. y J.D. Ullman**, Principles of Compiler Design, Addison-Wesley, Reading Mass, 1979.

ATKI78 **Atkinson L.V. y J.J. McGregor**, CONA-A Conversational Algol System, Software-Practice and Experience, vol. 8, 699-708, 1978

ATKI81 **Atkinson L.V. y S.D. North**, COPAS-A Conversational Pascal System, Software-Practice and Experience, vol. 11, 819-829, 1981

BYTE **McGraw-Hill Inc.**, One Phoenix Mill Lane, Peterborough, NH 03458.

CAMP84 **Campbell R.H. y P.A. Kirslis**, The SAGA Project: A System for Software Development, ACM 0-89791-131-8/84/0400/0073, 1984.

CHES84 **Chesi M., et al**, ISDE: An interactive Software Development Environment, ACM 0-89791-131-8/84/0400/0081, 1984.

ELLI82 **Elliot B.**, A High-level Debugger for PL/1, Fortran and Basic, Software-Practice and Experience, vol. 12, 331-340, 1982.

FEIL80 **Feller H.P.** y **R. Medina-Mora**, An Incremental Programming Environment, Department of Computer Science, **Carnegie-Mellon University**, abril 1980.

FITZ81 **Fitzhorn A.P.** y **R.G. Johnson**, C: Toward a Concise Syntatic Description, SIGPLAN Notices, vol.16, núm. 12, diciembre 1981.

HARB84 **Harbison S.P.** y **G.L. Steele**, A C Reference Manual, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

HART70 **Hart J.J.**, The Advanced Interactive Debugging System (AIDS), SIGPLAN Notices, vol. 14, núm. 12, 110-121 diciembre 1979.

JOHN79 **Johnson S.C.**, A Tour Through the Portable C Compiler, Bell Laboratories, 1979.

JOHN75 **Johnson S.C.**, YACC: Yet Another Compiler-Compiler, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, julio 1975.

KERN78 **Kernighan B.W.** y **D.M. Ritchie**, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

LESK75 **Lesk M.E.**, LEX: A Lexical Analyzer Generator, Comp. Sci. Tech. Rep. núm. 39, Bell Laboratories, Murray Hill, New Jersey,

octubre 1975.

LIV/C **Living Software**, 250 Orange Avenue, Orlando, Florida.

MEYR82 **Meyrokwitz N.** y **A. Van Dam**, Interactive Editing Systems, Computing Surveys, vol. 14, núm. 3. septiembre 1982.

RITC78 **Ritchie D.M.** y **K. Thompson**, The UNIX Time-Sharing System, The Bell System Technical Journal, vol. 57, núm. 6, julio-agosto 1978.

RUN/C **Lifeboat Associates**, 1651 Third Ave., New York, NY 10128.

STAL81 **Stallman R.M.**, EMACS: The Extensible, Customizable, Self-Documenting Display Editor, SIGPLAN/SIGOA Conference on Text Manipulation, ACM, Nueva York, 147-156, 1981.

STEF84 **Steffen J.L.**, Experience with a Portable Debugging Tool, Software-Practice and Experience, vol. 14, 323-334, abril 1984.

TEIT81 **Teitelbaum T.** y **R. Thomas**, The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, Communications of the ACM, vol. 24, núm. 9, 563-573, septiembre 1981.

VERM83 **Vermaak D.**, PROGEN: A Programming Environment, SIGPLAN Notices, vol. 18, núm. 3, marzo 1983.

GRAMATICA

```
%start c_pro

%union {
        short       ival;
        symT        *sval;
        long        txt;
        struct list *lst;
        expT        expr;
        aexpT       aexp;
}

%token <ival> STX
%token <ival> ETX

%token <expr> M_E
%token <expr> M_IF
%token <expr> M_SWITCH
%token <expr> M_RETV
%token <expr> M_GOTO
%token <expr> M_CASE
%token <expr> M_RET
%token <expr> M_DEF

%token <expr> BLTIN
```

GRAMATICA

```
%start c_pro

%union {
        short       ival;
        symT        *sval;
        long        txt;
        struct list *lst;
        expT        expr;
        aexpT       aexp;
}

%token <ival> STX
%token <ival> ETX

%token <expr> M_E
%token <expr> M_IF
%token <expr> M_SWITCH
%token <expr> M_RETV
%token <expr> M_GOTO
%token <expr> M_CASE
%token <expr> M_RET
%token <expr> M_DEF

%token <expr> BLTIN
```

```
%token <ival> b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 BB EB

%token <expr> ID
%token <ival> SWITCH
%token <ival> CASE
%token <ival> DEFAULT
%token <ival> IF
%token <ival> ELSE
%token <ival> FOR
%token <ival> WHILE
%token <ival> DO
%token <ival> BREAK
%token <ival> CONTINUE
%token <ival> GOTO
%token <ival> RETURN
%token <ival> SIZEOF
%token <ival> STATIC
%token <ival> CHAR
%token <ival> INT

%token <expr> CONSTANT
%token <expr> STRING
%token <ival> LABEL

%token <ival> INCREMENT                 /* ++ */
%token <ival> DECREMENT                 /* -- */
%token <ival> MEMBER                    /* -> */

%token <ival> SR                        /* >> */
%token <ival> SL                        /* << */
%token <ival> LE                        /* <= */
%token <ival> GE                        /* >= */
%token <ival> EQ                        /* == */
%token <ival> NE                        /* != */
```

```
%token <ival> LOR                                   /* || */
%token <ival> LAND                                  /* && */

%token <ival> A_P                                   /* += */
%token <ival> A_M                                   /* -= */
%token <ival> A_S                                   /* *= */
%token <ival> A_D                                   /* /= */
%token <ival> A_MOD                                 /* %= */
%token <ival> A_SR                                  /* >>= */
%token <ival> A_SL                                  /* <<= */
%token <ival> A_AND                                 /* &= */
%token <ival> A_X                                   /* ^= */
%token <ival> A_OR                                  /* |= */

/*
* precedence :
*/

%left ','                                                  b15
%right '=' A_P A_M A_S A_D A_MOD A_SR A_SL A_AND A_X A_OR  b14
%right '?' ':'                                     b13 /* */
%left LOR                                          b12 /* */
%left LAND                                         b11 /* */
%left '|'                                                  b10
%left '^'                                                  b9
%left '&'                                                  b8
%left EQ NE                                                b7
%left '<' LE '>' GE                                        b6
%left SR SL                                                b5
%left '+' '-'                                              b4
%left '*' '/' '%'                                          b3
%right '!' '~' INCREMENT DECREMENT SIZEOF          /* b2 */
%left '(' ')' '[' ']' MEMBER '.'                   /* b1 */
```

```
%left BB EB

%type <sval> SID DdecR Dspec FdecR CT BT
%type <ival> Tspec T oexp
%type <txt>  M00
%type <aexp> oexp expresion
%type <lst> statL stat compound_stat loop cond_stat N00
%type <lst> switch_stat action null
%type <lst> caseL case_stat default
%type <ival> Mfu00 Mfu01 M_nextQuad M_begin M_end
%type <ival> M_case M_default M_if M_else
%type <ival> M_while M_do M_for M_switch
%type <expr> gen_exp expjr gen_expL usr_exp code
%type <expr> exp_coma exp exp_aux una_exp bin_exp ass_exp
%type <expr> con_exp con_aux nlvalue lvalue expL usr_expL

%%

c_pro       :   /* */
            |   M00 sc Ddec ';' c_pro
            |   M00 fun_def c_pro
            |   usr_exp
            ;
/*
 *    function definitions :
 */

fun_def     :   sc Tspec FdecR Mfu00 par_dec Mfu01 compound_stat
            |   sc FdecR Mfu00  par_dec Mfu01 compound_stat
            ;
Mfu00       :   /* */
            ;
Mfu01       :   /* */
            ;
```

```
FdecR       :   SID auxFdecR '(' ')'
            |   SID auxFdecR '(' parL ')'
            |   '*' FdecR
            |   FdecR '[' ']'
            |   FdecR '(' ')'
            |   '(' FdecR ')'
            ;
auxFdecR    :   /* */
            ;
SID         :   ID
            |   '(' ID ')'
            ;
parL        :   ID
            |   ID ',' parL
            ;
par_dec     :   /* */
            |   se Tspec par_def ';' par_dec
            ;
par_def     :   DdecR
            |   DdecR ',' par_def
            ;
/*
 *    data declarations :
 */

internalDdec :  /* */
            |   se Ddec ';' internalDdec
            ;
Ddec        :   Tspec Dspec
            |   Tspec FdecR
            ;
Dspec       :   DdecR
            |   DdecR ',' Dspec
            ;
```

```
DdecR       :   SID
            |   CT
            ;
CT          :   BT
            |   '*' CT
            |   CT '[' ocexp ']'
            |   CT '(' ')'
            |   '(' CT ')'
            ;
BT          :   SID '[' ocexp ']'
            |   '*' SID
            ;
ocexp       :   /* */
            |   CONSTANT
            ;
/*
* statements :
*/

M_nextQuad:   /* */
      ;
statL :     stat
      |     statL M_nextQuad stat
      ;
stat  :     compound_stat
      |     switch_stat
      |     cond_stat
      |     loop
      |     action
      |     null
      |     expresion ';'
      ;
M_begin:     /* */
      ;
```

```
M_end :      /* */
      ;
compound_stat:  '{' internalDdec M_begin statL M_end '}'
      ;
M_switch:     /* */
      ;
M_case:      /* */
      ;
M_default:     /* */
      ;
switch_stat:SWITCH M_switch '(' ezpresion ')' '{' caseL M_nextQuad default '}'
      ;
caseL :      case_stat
      |      caseL M_nextQuad case_stat
      ;
case_stat:     CASE M_case CONSTANT ':' statL
      ;
default:     /* */
      |      DEFAULT M_default ':' statL
      ;
M_if :      /* */
      ;
M_else:      /* */
      ;
cond_stat:     IF M_if '(' ezpresion ')' stat
      |      IF M_if '(' ezpresion ')' stat M_else ELSE M_nextQuad stat
      ;
M_while:     /* */
      ;
M_do :      /* */
      ;
M_for :      /* */
      ;
```

```
loop    :   WHILE M_while '(' expresion ')' M_nextQuad stat
        |   DO M_do stat M_nextQuad WHILE '(' expresion ')' ';'
        |   FOR M_for '(' oexp ';' oexp ';' oexp ')' stat
        ;
action: BREAK ';'
        |   CONTINUE ';'
        |   GOTO ID ';'
        |   LABEL stat
        |   RETURN ';'
        |   RETURN expresion ';'
        ;
null    :   ';'
        ;
oexp    :   /* */
        |   expresion
        ;
/*
* expressions :
*/

usr_exp     :   STX expjr ETX
            |   STX '!' ETX
            ;
usr_expL    :   usr_exp
            |   usr_expL usr_exp
            ;
gen_exp     :   M_E expjr ETX
            |   M_IF expjr ETX
            |   M_SWITCH expjr ETX
            |   M_RETV expjr ETX
            |   M_GOTO CONSTANT
            |   M_CASE
            ;
```

```
expresion   :   M00 expjr
            ;
M00         :   /* */
            ;
expjr       :   exp
            |   exp_coma
            ;
exp_coma    :   expjr ',' expjr
            ;
exp         :   lvalue
            |   exp_aux
            |   '(' exp_aux ')'
/*          |   b1 exp
            |   b2 exp
*/
            |   b3 exp
            |   b4 exp
            |   b5 exp
            |   b6 exp
            |   b7 exp
            |   b8 exp
            |   b9 exp
            |   b10 exp
            |   b11 exp
            |   b12 exp
            |   b13 exp
            |   b14 exp
            |   b15 exp
            |   exp BB usr_expL EB
            ;
```

```
exp_aux     :   una_exp
            |   '(' exp_coma ')'
            |   bin_exp
            |   ass_exp
            |   con_exp
            |   nlvalue
            ;

una_exp     :   '&' lvalue      %prec '!'
            |   '-' exp         %prec '!'
            |   '!' exp
            |   '~' exp
            |   INCREMENT lvalue        /* ++ */
            |   DECREMENT lvalue        /* - */
            |   lvalue INCREMENT        /* ++ */
            |   lvalue DECREMENT
            |   '(' T ')' exp   %prec '!'
            |   SIZEOF exp
            |   SIZEOF T
            ;
```

```
bin_exp     :   exp '*' exp
            |   exp '/' exp
            |   exp '%' exp
            |   exp '+' exp
            |   exp '-' exp
            |   exp SR exp
            |   exp SL exp
            |   exp '|' exp
            |   exp '&' exp
            |   exp '^' exp
            |   exp '>' exp
            |   exp '<' exp
            |   exp LE exp
            |   exp GE exp
            |   exp EQ  ex
            |   exp NE  exp
            |   exp LOR exp
            |   exp LAND exp
            ;
ass_exp     :   exp '=' exp
            |   exp A_P exp
            |   exp A_M exp
            |   exp A_S exp
            |   exp A_D exp
            |   exp A_MOD exp
            |   exp A_SR exp
            |   exp A_SL exp
            |   exp A_AND exp
            |   exp A_X exp
            |   exp A_OR exp
            ;
con_exp     :   exp '?' con_aux
            ;
```

```
con_aux     :   exp ':' exp
            ;
nlvalue     :   CONSTANT
            |   STRING
            |   BLTIN
            |   ID '(' expL ')' code
            ;
code        :   /* */
            |   gen_expL
            ;
lvalue      :   ID
            |   '*' lvalue          %prec '!'
            |   '*' nlvalue         %prec '!'
            |   '*' '(' exp ')'     %prec '!'
            |   '(' lvalue ')'
            |   lvalue '[' exp ']'
            ;
expL        :   /* */
            |   exp
            |   exp ',' expL
            ;
/*
 *      type analysis
 */

Tspec       :   T
            ;
T           :   INT
            |   CHAR
            ;
sc          :   /* */
            |   STATIC
            ;
%%
```

INDICE DE FIGURAS