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 EDITOR Y COMPILADOR A CODIGO INTERMEDIO

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

Rodrigo Armando Sigüenza Vega

Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas

México, D. F. **1989**

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

CONTENIDO

Introducción

El proyecto que se presenta en esta tesis fue desarrollado a raíz de inquietudes originadas en el campo de trabajo. La diaria jornada de las personas que se dedican a la programación de sistemas sugiere, naturalmente, la concepción de nuevas herramientas para facilitar esta labor.

La palabra herramienta tiene su origen en *ferramenta* del Latín, que quiere decir instrumento de hierro. Sin embargo, con el tiempo, esta palabra ha adquirido otra connotación más general, dejando atrás la idea de que sólo es un instrumento de hierro.

En el campo de la computación éstas herramientas pueden tener una presencia física como, por ejemplo, una tarjeta digitalizadora, un mouse, un modem, etc., pero de igual manera pueden ser ideas, abstracciones producidas por programadores que vierten sus conocimientos y creatividad en instrumentos de trabajo intangibles, como algunos algoritmos, estructuras de datos o programas.

Los programadores constantemente tienen la necesidad de recurrir a diferentes herramientas de trabajo, que forman parte del proceso de programación y están involucradas en las siguientes etapas:

El programa es capturado por medio de un *editor* que nos permite manipular el texto de acuerdo con las especificaciones, gustos y estilos del programador; posteriormente, cuando ya está editado, se revisa la sintaxis del programa por medio de otra herramienta denominada *compilador*, éste checa el programa y si se detecta algún error, el programador corrige con el editor de texto, si no hay fallas en la sintaxis; entonces el programa es sometido a otra herramienta, el ligador, que resuelve las referencias externas uniendo programas objeto con las rutinas del sistema operativo, así como con diversas librerías. Después de esta etapa, al ejecutar el programa, pueden salir a la luz otros errores, los de lógica, que nos obligarán a recomenzar el proceso antes descrito.

Este ciclo, cuyas etapas no están integradas ni tienen una interface uniforme, es uno de los más grandes problemas a que se enfrenta un programador, ya que pierde mucho tiempo en el proceso de aprender a usar todas las herramientas mencionadas con anterioridad.

En este orden de ideas nace el proyecto **Ambiente de Programación para C (APC)** : que tiene como objeto el crear un ambiente de programación, interactivo, integrado y uniforme para el desarrollo de programas en lenguaje *C* con las siguientes características: la integración de un editor, un compilador y un intérprete-depurador en un solo sistema interactivo; la presentación de una interface uniforme para el usuario , así como hacer que cada una de estas herramientas tenga características que, a nuestro modo de ver, son importantes, pues aumentan la productividad del desarrollo de programas y/o sistemas.

En el proyecto *APC* intervienen dos tesistas de la maestria en Ciencias de la Computación de la *UACPyD* del *CCH*. Dos de las herramientas, *el Editor y el Compilador*, son el tema eje de la presente tesis, sin embargo, por las características del proyecto, es importante tener alguna información sobre la otra parte del sistema, *el intérprete y el depurador* a cargo de la Mat. Jennie Becerra Bertram, con objeto de tener una visión general del proyecto y ubicar dentro del mismo la parte que corresponde a este trabajo. Por esta razón, más adelante se hace una descripción de todos los módulos del proyecto *APC*.

La presente tesis, tiene por finalidad:

- Implantar un editor de texto que tenga una interface hacia las demás herramientas del ambiente.
- Implantar un compilador de*C*que genere un código intermedio para ser usado por el intérprete-depurador, y que use al editor de texto como el medio de comunicación.
- Evaluar la eficiencia del ambiente, asi como comprobar que las ideas usadas para resolver algunos problemas funcionen adecuadamente.

ANTECEDENTES

En la literatura se encontraron sistemas similares a nuestro concepto de ambiente de programación, sin embargo, no reúnen todas las características que *APC* tiene. Se ha notado que el grueso de los ambientes de programación producidos en ambientes académicos, se han orientado hacia la utilización de editores dirigidos por sintaxis, mientras que los sistemas comerciales utilizan editores de texto.

Algunos de los sistemas existentes son:

- CONA A Conversational Algol System [ATKI78], tiene las siguientes limitaciones: no permite la recursividad de funciones y contiene un editor de líneas.
- COPAS A Conversational Pascal System, orientado a PASCAL [ATKI81], similar al anterior.
- IPE Incremental Programming Environment [FEIL80], que forma parte del proyecto Gandalf fue desarrollado para GC, que es una variación de *C* que verifica tipos. 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.
- 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 PI/1 y posteriormente para Pascal.

Esta tesis está organizada en tres capítulos, las conclusiones, la bibliografía y un apéndice. En el capítulo 1 se da un panorama de cada módulo del ambiente, así como su relación y un ejemplo de sesión típica de *APC*. El capítulo 2 describe el editor, desde el punto de vista del diseño y la implantación, al final se proporciona una lista de todos los comandos. El capítulo 3 trata lo relacionado con el compilador, dando el mismo enfoque del capítulo 2. Y por último, en el apéndice A se muestra la gramática de *C* que se usó en *APC*.

1 Diseño Conceptual del Sistema

1.2 División Modular del Sistema

El ambiente *APC* consta de tres módulos principales: **Editor, Compilador e Intérprete-depurador.** El editor se encarga de manejar el texto de los programas fuentes, así como la interface con el usuario, el compilador tiene como función generar el código intermedio y el intérprete usa el código intermedio y ejecuta el programa, proporcionando al usuario una serie de facilidades que le permiten corregir su programa rápidamente.

EDITOR

El editor consta de: un procesador de comandos, un módulo de edición, un manejador de ventanas, un manejador de terminales y un módulo de interface con el ambiente de programación.

El procesador de comandos toma los caracteres de la terminal, los analiza e invoca las rutinas apropiadas.

El módulo de edición está constituido por un conjunto de funciones que hacen lo necesario para editar un texto.

El manejador de ventanas controla la operación de las tres ventanas de *APC*, la primera ventana es la de edición, la segunda es la de estatus, donde se escriben los resultados de los comandos ejecutados, y la última es la de ejecución, en la que se escriben los resultados del programa que interpreta *APC*.

Es común que en una máquina se tengan terminales de diferentes tipos y marcas, cada tipo de terminal requiere ser manejada de forma diferente, esto hace que los programas que manejan la pantalla para escribir caracteres en lugares determinados se compliquen. Para no limitar el

editor a manejar sólo un tipo de terminales, se implantó el módulo de manejo de terminales.

El módulo de interface con el ambiente de programación tiene por objeto invocar las demás herramientas desde el editor y proporcionar información del estado de la edición a los demás elementos del medio ambiente, como son el compilador y el intérprete.

COMPILADOR

Traducción del programa a código intermedio

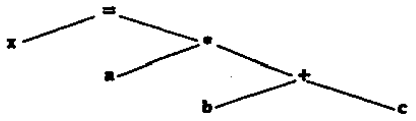
Un método empleado comúnmente en compiladores e intérpretes es la traducción de las instrucciones a una representación más simple; en el caso de un compilador se usa esta representación para aplicar sobre ella algoritmos de optimización de código y para separar el proceso de compilación en módulos *(front end* y *back end)*. Los intárpretes manejan representaciones que les permiten interpretar el código de manera conveniente. El código intermedio es el lenguaje de una computadora no existente, que el diseñador define y el intérprete simula.

El proceso de obtener una representación intermedia, por lo general está dirigido por sintaxis, esto se hace a medida que se va analizando sintácticamente el programa.

Existen varias formas de representación intermedia. Algunas de éstas se describen a continuación empleando la instrucción:

$$x = a * (b + c)$$

a) Arboles sintácticos



b) Código de tres direcciones con variables temporales

$$T1 = b + c$$

$$T2 = a * T1$$

$$x = T2$$

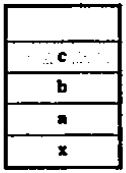
c) Máquina de stack

En este caso se traduce la expresión a una representación conocida como notación postfija:

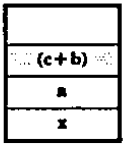
x a b c + * =

Una vez generada la notación anterior, se puede evaluar fácilmente empleando un stack. El procedimiento sería recorrer el código postfijo de izquierda a derecha y hacer un push de cada operando que se encuentre. Si se encuentra un operador de *k* operandos, su primer argumento (el de más a la izquierda) será la *k-1* posición hacia abajo del tope del stack, su último argumento será el tope del stack. A estos valores se les hace un pop y al resultado de aplicar el operador se le hace un push. Este procedimiento se ejemplifica a continuación.

Estado inicial x a b c + * =

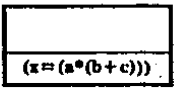


Primer operador x a b c + * =



Segundo operador x a b c + * =

Tercer operador x a b c + * =



Una decisión natural para *APC* hubiera sido escoger una de estas representaciones, traducir el programa e interpretar el lenguaje intermedio, lo cual tendría las siguientes ventajas:

- Una vez hecha la traducción, es rápida su evaluación y por lo tanto la ejecución del programa que se interpreta es más rápida.
- El programa o función encargada de la interpretación del código intermedio es más simple en su implantación y en su depuración.

Existen sin embargo las siguientes desventajas:

- Si se modifica la expresión original, la representación intermedia ya no sirve; se debe recompilar y obtener una nueva representación.
- Se requiere más memoria, ya que, además del programa fuente, se deben almacenar las representaciones intermedias de cada expresión.

Dados los objetivos de *APC* estas desventajas son determinantes, no se quiere que un pequeño cambio en el programa implique una recompilación total. El campo de la compilación incremental ha recibido cierta atención en la literatura, pero el problema dista de estar resuelto de manera satisfactoria. En *APC* se eligió una solución intermedia, en la cual el esqueleto del programa (control de flujo) está traducido a código intermedio, para lo cual se escogió el código tres direcciones, mientras que las expresiones se dejan como texto después de haberlas, simplemente, analizado para verificar que sintácticamente sean correctas.

El intérprete desifra el código intermedio del esqueleto pero, al llegar a una expresión, se evalúa mediante el análisis sintáctico. Esta última parte se deriva directamente de la idea del programa clásico de una calculadora de escritorio [Aho 77], este programa tiene por entrada

una cuerda con la expresión y proporciona, en la salida, el resultado de evaluarla.

Existen muchas formas de solucionar el problema de la calculadora de escritorio, una de ellas es la llamada traducción dirigida por sintaxis [Aho 77], la cual se eligió como el *motor* del intérprete de *C* de *APC*.

La traducción dirigida por sintaxis consiste en hacer una o varias acciones cada vez que se reconoce una regla de la gramática; a estas acciones se les denomina acciones semánticas, que son por lo regular fragmentos de código de un lenguaje de alto nivel. Esta forma de traducción es usada por *YACC* [JOHN 75], programa generador de parsers, que se empleó en la implantación de este trabajo.

YACC toma como entrada una especificación de la gramática de un lenguaje y las acciones semánticas asociadas a las reglas gramaticales, y genera el programa fuente de una función en *C*, llamada *yyparse ()*.

Las acciones semánticas son fragmentos de código en *C* que se ejecutan cuando el parse reconoce la regla gramatical.

El parser generado por *YACC* hace uso de una función llamada *yylex ()*, que tiene la responsabilidad de leer el archivo de trabajo, analizarlo lexicográficamente y proporcionar el número de token correspondiente. Esta función puede ser escrita por el usuario o bien puede ser generada por el programa *LEX* [LESK 75], que es un generador de scanners que funciona en forma similar a *YACC*. *LEX* también se empleó en la implantación de este proyecto.

En el siguiente ejemplo se muestra la gramática de una calculadora que permite hacer: sumas, restas, multiplicaciones, divisiones y resolver adecuadamente la agrupación de subexpresiones por medio de paréntesis.

```
1   list:       /* Regla Vacia */
2           |   list '\n'
3           |   list expr '\n'
4           {
5               printf ("\t\t%.8g\n", $2);
6           }
7           ;
8
9   expr:       NUMBER    { $$ = $1; }
10          |   expr '+' expr { $$ = $1 + $3; }
```

```
11          |   expr '+' expr { $$ = $1 - $3; }
12          |   expr '+' expr { $$ = $1 * $3; }
13          |   expr '+' expr { $$ = $1 / $3; }
14          |   '(' expr ')'    { $$ = $2; }
15  ;
```

Las acciones semánticas se encuentran encerradas por los caracteres {}, en el caso de la regla del renglón 3, la acción es imprimir el valor regresado por la expresión EXPR (renglón 5), simbolizada por $2.

Los *tokens* definidos para esta gramática son los cuatro operadores aritméticos, los paréntesis y el *token NUMBER* , en el caso de detectar una constante.

De esta forma, el scanner o analizador lexicográfico lee la cuerda de entrada carácter por carácter, cuando reconoce una constante regresa su valor y el *token NUMBER*; O bien, si detecta cualquiera de los otros tokens regresa el carácter mismo, por ejemplo '+'.

Las acciones semánticas del ejemplo están en el formato que usa *YACC* y la gramática es LALR(1), de tal forma que este mismo ejemplo puede generarse con *YACC*.

La regla de la línea 9, describe una expresión que puede ser una constante la acción $$ = $1 hace que se asigne a esta regla el valor de la constante representada por el $1.

De igual forma, las reglas de los renglones 10, 11, 12, 13 y 14 regresan, en función de su operador, el valor adecuado.

De esta forma tan sencilla es como nuestro intérprete se encarga de evaluar las expresiones del programa, tomando como cuerda de entrada el texto de dicha expresión.

El compilador de *APC* está encargado de generar un conjunto de tablas que emplean las otras herramientas del ambiente, una de estas tablas es en sí la representación intermedia. Dadas las características del lenguaje *C*, el compilador identifica tres tipos de instrucciones, las declaraciones, el control de flujo y las expresiones.

Cuando el compilador detecta declaraciones, como por ejemplo de una variable, genera una entrada en la tabla de símbolos, donde se describen las características de la variable declarada.

Cuando analiza el control de flujo, el compilador genera una tabla de código intermedio usada posteriormente por el intérprete.

Al detectar expresiones, requerimos el principio y final de éstas en el texto fuente para que sean evaluadas al momento de ejecución.

INTERPRETE

El intérprete utiliza las tablas de símbolos y esqueleto que previamente el compilador preparó. Las acciones semánticas asociadas a las reglas gramaticales son las que interpretan el código. En su implantación se usaron *LEX* y *YACC*. La gramática definida para el compilador se extendió, de manera que el intérprete usa la misma para ejecutar el código intermedio y evaluar expresiones.

El intérprete es el encargado de la asignación de almacenamiento, el compilador registra para todas las variables ya sean globales, automáticas o parámetros de funciones, como un offset, a partir del cual se encuentra almacenado el valor de la variable. El intérprete toma una base y le suma el offset para obtener la dirección absoluta en memoria de la variable. Todas las variables comparten una misma área de memoria. Las variables globales ocupan las localidades bajas de esta área y el almacenamiento para ellas es estático, no varía durante los diversos procesos, a menos que se efectúe una nueva compilación.

Para una parte del almacenamiento dinámico se utilizan las localidades altas de esta misma área. Toda la información necesaria para la ejecución de una función se ha agrupado en una porción de memoria llamada registro de activación. Este registro contiene espacio para el paso de parámetros, para las variables automáticas definidas en esa función, un apuntador al registro de activación anterior, la dirección de regreso de la función y cierta información adicional. Para el manejo de estos registros se ha implantado un mecanismo de pila que parte de las localidades altas y crece hacia las localidades bajas. La implantación de los registros de activación en esta forma permite el manejo de funciones en *C* llamadas con un número variable de parámetros.

El almacenamiento para las variables temporales que se requieren en la evaluación de expresiones se maneja también en forma dinámica por medio de una pila, empleándose para ello la pila que utiliza y administra internamente *YACC*.

Funciones de depuración

APC tiene tres funciones de depuración, el *trace* que consiste en posicionar el cursor en cada expresión del programa al momento de su ejecución, el *show* que despliega el valor de toda variable que experimenta algún cambio y el *break* que detiene la ejecución del programa en algún lugar previamente definido.

La implantación de las funciones de *trace()*, *show()* y *break()* se hizo con base en banderas que se prenden y apagan. Dependiendo de la función, alguno de los módulos se encarga de interpretarla. En el caso de *trace()*, al estar prendida esta bandera, el editor que se encarga de leer las expresiones del texto del programa muestra el cursor bajo la operación que se está realizando en ese momento. Para *show()* el intérprete, cuando detecta que está prendida esta bandera, al modificar una variable en el transcurso de la evaluación de una expresión, despliega su nuevo valor. Al prender *break()* a bandera correspondiente, el intérprete la recibe y suspende la evaluación de la expresión.

Por medio del editor es posible indicar el lugar donde debe detenerse el programa, poniendo una marca sobre algún operador. Cuando el editor recibe esta indicación, prende el bit más significativo del carácter que representa la operación. Posteriormente al leer esta expresión, detecta que está encendido este bit y suspende la ejecución del programa.

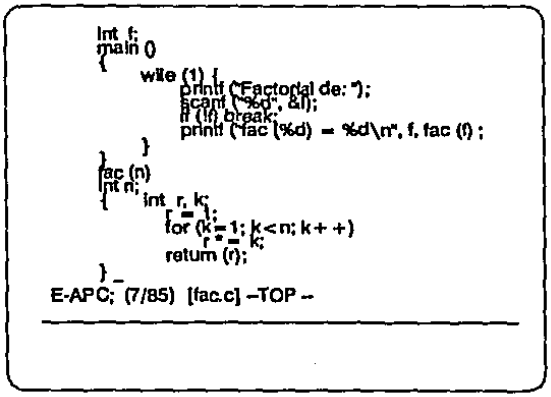
Al recibir el intérprete, en la ventana de ejecución, una expresión para evaluar, si ésta no contiene funciones, encuentra su valor y se lo entrega al usuario. Ahora bien, si una función forma parte de la expresión, el intérprete procede a la ejecución del código intermedio asociado a ella. Como vimos, el código intermedio está formado por instrucciones que representan el flujo del programa y por direcciones de expresiones que se encuentran en el texto mismo del programa del usuario. El intérprete entonces procede a la ejecución de las instrucciones contenidas en el código intermedio y a la evaluación de las expresiones indicadas por los apuntadores contenidos en el código.

El intérprete también controla la pila de registros de activación, proporcionando la información necesaria para realizar la liga entre funciones y registrando los diferentes contextos en los cuales aparecen las expresiones.

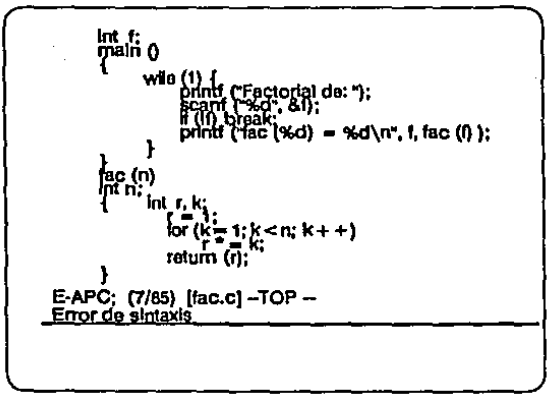
EJEMPLO.

Una vez unidas todas las herraminetas e ideas en *APC*, una sesión típica sería:

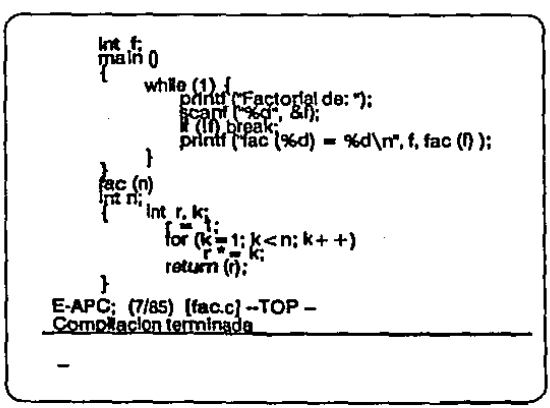
Introduccion de un programa fuente por medio del editor



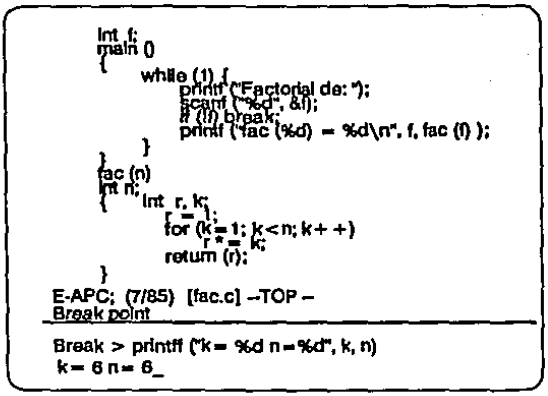
Compilacion



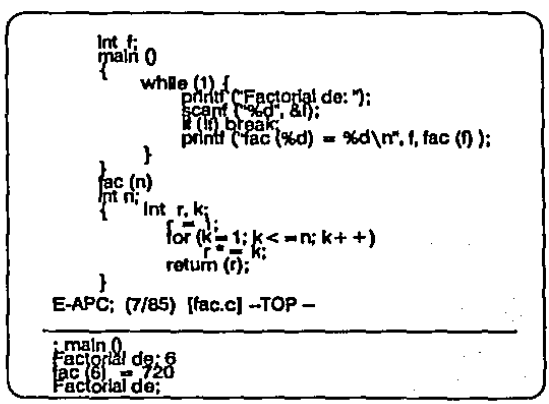
Correcion del error de sintaxis



Depuracion



Correccion



2 Editor

A los programas que tienen como finalidad permitir la escritura y modificación de textos se les llama editores. Un editor se caracteriza por permitir crear, borrar, insertar o modificar un texto o partes de él.

2.1 Introducción

Los editores de texto se han utilizando desde los principios de la computación. Inicialmente el problema de editar era mucho mayor, ya que sólo se disponía de teletipos, pero con el advenimiento de las terminales con pantalla se logró que el medio ambiente de edición se hiciera un poco más accesible. En una pantalla se despliega el texto y se pueden observar las modificaciones, mientras que en un teletipo esta situación es virtualmente imposible.

Entre estos editores se distinguen principalmente: los de línea, que cuentan con comandos para seleccionar la línea o líneas que se quieren modificar, de alguna manera, los editores de línea vienen a recordar las tarjetas perforadas; en donde cada tarjeta contenía una línea o renglón del texto del archivo. Ejemplos de este tipo de editores son el *EDI* de *RSX11M* el *ED* de *UNIX*; el otro tipo, es el de pantalla, en el cual el cursor puede ser movido en forma más o menos libre alrededor de la pantalla y del archivo para hacerle modificaciones como en el caso del *VI* de *UNIX*.

Entre los editores de pantalla que más éxito tienen, por su versatilidad y por las facilidades que brinda, así como por lo sencillo de su operación, se encuentra el editor llamado *EMACS [CIC77]* que se desarrolló sobre un editor más primitivo de caracteres *TECO*, *EMACS* está hecho a base de macros. En este editor, con comandos de control y de escape, es posible hacer cualquier modificación a un texto en forma sencilla, rápida y teniendo en todo momento a la vista el texto que se está editando.

La herramienta de edición que se integró al ambiente de programación *APC*, usa un subconjunto de instrucciones de *EMACS* para la modificación del texto (programa fuente), y al mismo tiempo permite invocar otras herramientas como el compilador y el intérprete, así como pasar información de control al depurador.

2.2 Características generales

A continuación se hace un análisis de las características que decidimos que debía tener el editor, mismas que influyeron en el diseño e implantación.

Tamaño y simplicidad

Se consideró que el editor ocupara poca memoria con el objeto de poder usarlo dentro del ambiente sin tener que recurrir a técnicas de *OVERLAYS*, logrando así tener al mismo tiempo, en la memoria, el programa fuente que se está editando y las demás herramientas del ambiente. Para lograr este objetivo, se eliminaron algunas de las funciones de *EMACS* consideradas de poca utilidad, más adelante se proporcionarán las funciones de edición incluidas en este editor.

Con el objeto de simplificar el editor, se decidió que el tamaño de los archivos a editar debían ser menores de 32K caracteres, aproximadamente 10 cuartillas. Esto nos permite tener el archivo de trabajo completo en la memoria principal, evitando así tener que manejarlo en memoria secundaria.

Consideramos que esta restricción no afecta, ya que los programadores dividen sus sistemas en un conjunto de archivos que, normalmente, son pequeños (unos 10K), esto es posible gracias a las características del lenguaje *C* y a las técnicas de programación estructurada.

Tratándose de un editor de propósito especial, nos fue posible limitar el conjunto de caracteres que debía manejar a los que soporta en sí el lenguaje de programación *C*, de esta forma se logró reducir aún más el diseño porque el único carácter de control que contiene el texto editado es el de fin de línea, de esta forma sólo se guardan en el texto caracteres imprimibles.

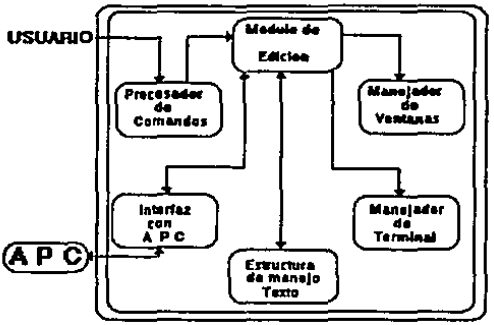
Una de las políticas que se consideraron necesarias, fue la de lograr un alto grado de transportabilidad, razón por la cual se eligío *C* como lenguaje. La mayor parte del sistema es independiente del hardware; y aquellas secciones que dependen de él se han reunido en dos módulos

específicos, de tal manera que para transferir el sistema a otra máquina sería necesario sólo modificar estas secciones.

2.3 Diseño e Implantación

Arquitectura del editor

El editor consta de: un procesador de comandos, un módulo de edición, un manejador de ventanas, un manejador de terminales y un módulo de interface con el ambiente de programación, tal como se muestra a continuación:



El procesador de comandos toma los caracteres de la terminal, los analiza e invoca las rutinas apropiadas. El editor está siempre en modo de inserción y no es necesario cambiar de modo para ejecutar comandos. Todas las teclas, de control, de escape y de caracteres visibles realizan una función determinada. En particular, las teclas de caracteres visibles provocan que en la posición indicada por el cursor se inserte el carácter en el texto. Para ejecutar estas funciones, el procesador de comandos contiene una tabla en la cual para cada carácter invocado se tiene un apuntador a la función que realiza el comando asociado al carácter. El cuerpo de este bloque es un ciclo infinito en el cual se *lee* un carácter de la terminal y se ejecuta la función asociada a él.

Tabla de funciones del procesador de comandos:

Indice		Función	Descripción
Decimal	ASCII		
0	NULL	marcar ()	Guarda la dirección del cursor para marcar una región
1		err ()	Manda un mensaje de error
2			
3	^C	cc ()	véase comando del editor ^C^C
4-6		err ()	Manda un mensaje de error
7	^G	cg ()	Abortar un comando
8-9		err ()	Manda un mensaje de error
10	^J	cj ()	véase comando del editor ^J
11	^K	ck ().	véase comando del editor ^K
12	^L	cl ()	Hace scroll poniendo el cursor a mitad de la pantalla
13	^M	cm ()	véase comando del editor ^M
14	^N	cn ()	véase comando del editor ^N
15	^O	co ()	Abre una línea
16	^P	cp ()	véase comando del editor ^P
17		err ()	Manda un mensaje de error
18	^R	cr ()	véase comando del editor ^R
19	^S	cs ()	véase comando del editor ^S
20		err ()	Manda un mensaje de error
21			
22	^V	cv ()	véase comando del editor ^V
23		err ()	Manda un mensaje de error
24	^X	cx ()	Modo de extensión de comandos
25		err ()	Manda un mensaje de error
26	^Z	cz ()	Salir de *APC*
27	ESC		Modo de extensiom con tecla alternas
28-31		err ()	Manda un mensaje de error
32-126		insert ()	Inserta el carácter ASCII
127	DEL	bdb ()	véase comando del editor ^D
128-161		err ()	Manda un mensaje de error

Indice		Función	Descripción
Decimal	ASCII		
188	M<	mtop 0	véase comando del editor $<
190	M>	mbot 0	véase comando del editor $>
214	MV	mv 0	véase comando del editor $V
215-255		err 0	Manda un mensaje de error

Cualquier cambio al texto se refleja, por un lado en la pantalla y por otro, internamente en las estructuras que contienen el texto. Para lograr una apariencia de mayor velocidad se ejecutan, hasta donde es posible, en forma independiente, las funciones que manipulan la pantalla y aquéllas que manejan la representación interna del texto. Esto es muy importante, en un ambiente multiusuario como *UNIX*, porque cuando el sistema está cargado, los cambios en la pantalla se interrumpen a cada momento.

El módulo de edición lo conforma el conjunto de funciones llamadas por el procesador de comandos enumeradas en la tabla anterior, este módulo deja en la memoria los cambios que se hacen al texto y llama al manejador de ventanas para reflejarlos en la terminal.

El módulo manejador de terminales se encarga de manejar correctamente éstas, basándose en la información particular de cada una de ellas registrada en el sistema operativo. Así por ejemplo, sabe que para borrar la pantalla de una terminal *H19* se debe mandar la secuencia de caracteres *[ESC], [[], [2], [j]* y en una terminal *VT52* la secuencia es [ESC], [M]. Por lo tanto, el editor de *APC* funciona con cualquier tipo o marca de terminal que esté registrada por *UNIX*.

El manejador de ventanas es el encargado de llevar el control del video de la terminal y separarlo en las dos ventanas que usa el ambiente. Este módulo sabe a dónde (renglón y columna) debe mandar un carácter que se envía a la ventana de edición u otro que se envía a la ventana de comandos.

El módulo de interface con el ambiente de programación tiene por objeto invocar las demás herramientas desde el editor y proporcionar información del estado de la edición a los demás elementos del medio ambiente, como son el compilador y el intérprete. También deja información en el bit más significativo de los caracteres del texto para posteriormente ser usada por el depurador.

A continuación se describen, en el formato de los manuales de *UNIX*, las funciones que conforman el módulo de interface, se incluye esta descripción para dar un panorama de cómo está construido el editor.

NOMBRE

reset_txt

SINOPSIS

void reset_txt ()

FUNCION

inicia las variables adecuadas para que la función nextch() regrese el primer carácter del archivo de trabajo.

NOMBRE

display

SINOPSIS

void display (xx, yy) short xx, yy;

PARAMETROS

xx, yy columna y renglón del archivo de trabajo.

FUNCION

despliega, en la pantalla, la sección del texto indicada por xx, yy.

NOMBRE

nextch

SINOPSIS

int nextch ();

FUNCION

regresa el siguiente carácter, tiene un conjunto de banderas que le determinan de dónde debe leer el siguiente carácter, i) del archivo de trabajo, ii) del buffer de comandos del usuario o iii) buffer de evaluación de los *BREAK-POINTS*.

NOMBRE

send_tok

SINOPSIS

void send_tok (t, v) short t, v;

PARAMETROS

t es el número de token, definido por *YACC*. *v* es el valor que se desea asignar a este token

FUNCION

envía un token a *YACC*, esta función se emplea en el intérprete para comunicarse con *YACC*.

NOMBRE

setexp

SINOPSIS

void setexp (Badd, Eadd) textT Badd, Eadd;

PARAMETROS

Badd, Eadd, direcciones del principio y fin de una expresión

FUNCION

asigna las variables adecuadas para que la función nextch () mande el siguiente carácter a partir del primero de esta expresión.

Estructuras para representar el texto

Consideramos que un texto es un conjunto de caracteres almacenados en un archivo en disco o en memoria principal. Por ejemplo, el texto siguiente:

esto
es un fragmento
de texto.

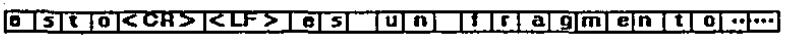
estaría almacenado como:

esto<CR><LF>es un fragmento<CR><LF>de texto.<EOF>

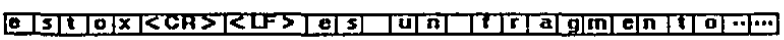
Los caracteres <CR> <LF> se emplean para separar una línea de otra, y el carácter <EOF> indica el fin del archivo de texto.

Las operaciones más comunes efectuadas por los editores son la inserción y eliminación de caracteres. Esto se ilustra en el siguiente ejemplo:

Supóngase el texto anterior representado en la memoria por un arreglo llamado buff, donde buff[0] es el primer carácter y buff[n-1] es el n-ésimo:



Al insertar el carácter "x" en buff[4], quedará:



Esto implica mover los caracteres de buff[4] en adelante, para hacer un lugar al nuevo carácter, la eliminación es análoga.

El problema de insertar o borrar una línea es semejante al problema anterior. Por ejemplo, si se desea insertar una línea más, inmediatamente después de la primera, en el texto anterior, entonces su representación en la memoria quedaría:

Nivel lógico	
Antes	Después
esto	esto
es un fragmento	línea nueva
de texto	es un fragmento
	de texto

Nivel físico:

Antes:

Después:

En un texto muy grande estas operaciones pueden generar un número excesivo de corrimientos de caracteres, si el texto está simplemente representado por una secuencia de éstos.

Por lo anterior, surgió la necesidad de diseñar una estructura para representar el texto que cumpliera con las siguientes características:

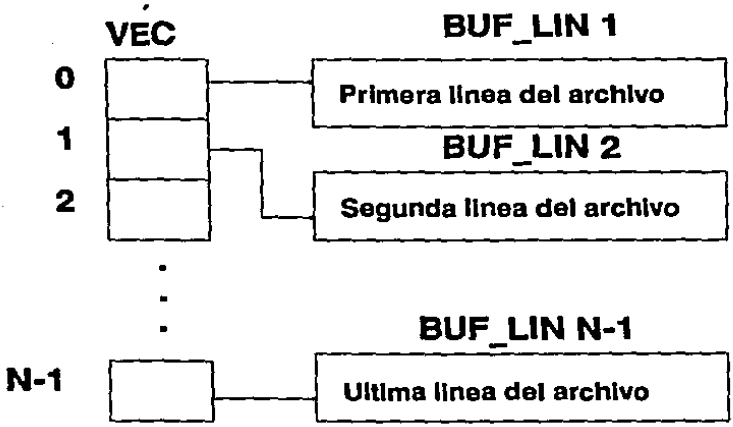
- Que refleje el efecto del corrimiento de caracteres, sin necesidad de hacerlo físicamente en todo el texto.
- Que sea propicia para hacer un manejo eficiente de memoria.
- Que permita realizar un despliegue adecuado del texto en la pantalla.
- Que facilite el acceso a cualquier parte del texto en forma inmediata.
- Que ocupe poca memoria.

Se analizaron varias alternativas de representación y se concluyó que eran dos las adecuadas.

La primera de ellas consideraba la opción de representar el texto en un conjunto de buffers doblemente ligados; cada buffer ocupado en forma inicial hasta la mitad de su capacidad. En este metodo cada inserción tiene el efecto de recorrer todos los caracteres contenidos en uno de estos buffers. Cuando uno de éstos se llena a un cierto nivel, se crea un buffer adicional y su contenido se reparte entre los dos en mitades.

Esta representación facilitaba el manejo de memoria virtual, es decir, la capacidad de editar un archivo más grande que la memoria principal; pero dificultaba el despliegue y el movimiento a través del archivo, este método está descrito en *[RICE72]*. Finalmente esta idea fue desechada.

La segunda, que fue la elegida, tiene la siguiente representación:



donde:

VEC es un arreglo, a través del cual es posible recorrer el archivo editado, es decir, el elemento i-ésima de *VEC* nos da la dirección de la i-ésimo línea del archivo.

Los buffers *BUFF_LIN* son de tamaño fijo y cada línea del archivo (texto entre dos [*CR*]) está contenido en uno de ellos, siguiendo la convención de poner un carácter nulo al final de la línea. Esto desperdicia un poco de espacio, pero simplifica bastante el funcionamiento y la implantación del editor.

La forma de insertar/borrar caracteres es como sigue:

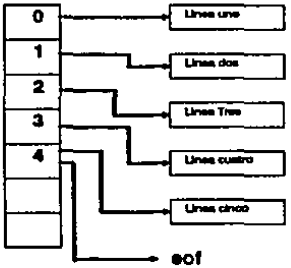
i) Si el carácter a insertar cabe en el buffer *BUF_LIN* correspondiente, solamente se inserta, recorriendo los caracteres en esa línea. El borrado de un carácter es similar, pasando al siguiente caso si la línea quedó vacía.

La forma de insertar/borrar líneas es como sigue:

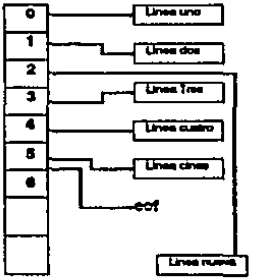
i) Si se quiere abrir una línea nueva entre dos ya existentes, j y j+1, se solicita un nuevo *BUF_LIN* por medio de un *alloc ()* y se lee el texto nuevo directamente en *BUF_LIN*, cuidando de no pasarnos nunca del número máximo de caracteres.

ii) En el arreglo *VEC* se recorren una posición a partir de la localidad j+1, todos los apuntadores, dejando un espacio para insertar en la posición j+1 el nuevo apuntador correspondiente a la línea recién insertada. Esto se ilustra en el siguiente ejemplo.

Supóngase que se desea insertar una nueva línea entre la 2 y 3 como se muestra en la figura siguiente.



El resultado sería:



cuando se trata de borrar una línea completa se hace lo contrario a la inserción, recorriendo los apuntadores del arreglo *VEC*, dejándolo de tal forma que siempre sea posible recorrer el archivo secuencialmente, y el renglón borrado se libera con la función *free ()*.

Cualquier otra modificación al texto se descompone en las primitivas anteriores.

La estructura *VEC* siempre está en memoria principal y determina el tamaño máximo del archivo que puede ser editado. Cada elemento del arreglo es un pointer que requiere de cuatro bytes, por lo que para poder editar un archivo de mil líneas *VEC* debe tener por lo menos mil apuntadores, esto implica reservar aproximadamente 4K bytes, más lo que ocupe cada renglón de texto.

Estos parámetros, como máximo el tamaño del arreglo *VEC* y otros parámetros importantes están reunidos en un archivo especial de los programas fuente del sistema con el objeto de configurarlo fácilmente.

La estructura de datos para ventanas está formada, también, por un arreglo de estructuras donde cada elemento corresponde a una ventana, estas se conforman de los siguientes elementos:

- Posición inicial de la ventana en la pantalla.
- Posición final de la ventana en la pantalla.
- Identificador del buffer desplegado.

2.4 Comandos del editor.

Los comandos de control se simbolizan con ^[car], lo cual significa oprimir simultáneamente la tecla marcada *control* y el carácter *car*; los comandos de escape se simbolizan como $[car], que significa oprimir la tecla *esc* y en seguida el carácter *car*.

Movimiento del cursor	
Comando	**Descripción**
^F	Mueve el cursor un carácter hacia adelante. Al llegar al final de línea, el cursor se mueve al principio de la siguiente línea.
^B	Mueve el cursor un carácter hacia atrás. Al llegar al inicio de la línea, el cursor se mueve al final de la línea anterior.
^N	Mueve el cursor una línea hacia abajo.
^P	Mueve el cursor una línea hacia arriba.
^A	Mueve el cursor al principio de una línea.
^E	Mueve el cursor al final de una línea.
$F	Mueve el cursor una palabra hacia adelante. Sólo avanza sobre letras y/o números.

Movimiento del cursor	
Comando	Descripción
$B	Mueve el cursor una palabra hacia atrás. Sólo retrocede sobre letras y/o números.
$<	Mueve el cursor al principio del texto.
$>	Mueve el cursor al final del texto.

Borrado	
Comando	Descripción
^D	Borra el carácter sobre el cursor.
^[del]	Borra el carácter anterior al cursor.
^K	Borra a partir de donde está el cursor hasta el final de la línea.
$D	Borra la palabra a partir de donde está el cursor.
$[del]	Borra la palabra hacia atrás de donde está el cursor.

Inserción	
Comando	Descripción
^I	Se inserta un tab (representado por blancos), donde está el cursor.
^J	Se inserta una línea debajo de donde está el cursor
^M	Se inserta una línea delante de donde está el cursor.

Búsqueda	
Comando	Descripción
^S	Busca una cuerda hacia adelante.
^R	Busca una cuerda hacia atrás.

Manejo de Pantalla	
Comando	Descripción
^V	Avanza una pantalla hacia adelante dejando parte del texto de la pantalla anterior en la parte superior.
$V	Retrocede una pantalla dejando parte del texto de la pantalla posterior en la parte inferior.

Manejo de Archivos	
Comando	Descripción
^X^F	Visita un archivo.
^X^S	Salva un archivo.
^X^Z	Se sale del ambiente.

Interface con otras herramientas del ambiente	
Comando	Descripción
^C^C	Compila el programa fuente.
^C^R	Ejecuta la función main del programa fuente.
^C^T	Prende el modo TRACE de rastreo

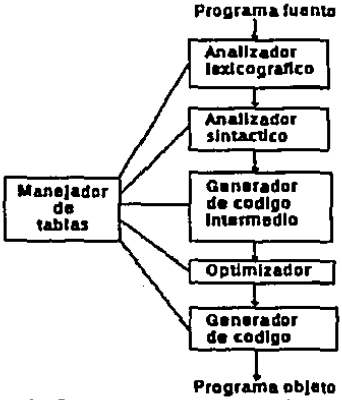
3 Compilador

Un traductor es un programa que toma como entrada un programa escrito en un lenguaje de programación (lenguaje fuente) y produce como salida el programa en otro lenguaje (lenguaje objeto). Si el lenguaje fuente es de alto nivel como C, Pascal o Fortran, y el lenguaje objeto es de bajo nivel, como ensamblador o lenguaje de máquina, entonces se dice que el traductor es un compilador. [Aho 77].

En el caso particular de *APC*, se traduce de *C* a un lenguaje de bajo nivel (código intermedio), el cual posteriormente será interpretado por el *intérprete-depurador*.

Este proceso de traducción es bastante complejo, por lo que para hacerlo se ha dividido en fases. Una fase toma como entrada una representación del programa fuente y produce como salida otra representación.

La primera fase, llamada análisis lexicográfico o *scanner*, separa los



caracteres del lenguaje fuente en grupos que lógicamente deben estar

juntos; estos grupos son llamados *tokens* o elementos lexicográficos, éstos son, podría decirse, las palabras del lenguaje, como por ejemplo: *while, if*, identificadores como *X* o *NUM*, operadores como < = o + y símbolos de puntuación como paréntesis o comas. La salida del análisis lexicográfico es una secuencia de tokens, que es pasado a la siguiente fase, análisis sintáctico o parser.

El parser agrupa tokens formando estructuras sintácticas, por ejemplo, los tres tokens siguientes:

A + B

podrían ser agrupados en una estructura sintáctica llamada *expresión* y éstas pueden combinarse para formar *statements* o instrucciones.

El generador de código intermedio *(CI)*usa las estructuras producidas por el parser para generar una secuencia de instrucciones simples. Existen varios tipos de CI. El más comúnmente usado es el llamado código de tres direcciones o *cuádruples*, que es el empleado en *APC*. La diferencia principal entre el código intermedio y el ensamblador es que el CI no necesariamente requiere especificar los registros que serán usados para cada operación.

Los módulos o fases de un compilador encargados de la generación del CI forman lo que se conoce como el *front end*. El *back end* son las fases que traducen a lenguaje objeto el CI.

El manejador de tablas o *book keeping* es una parte del compilador que sirve para almacenar información de los identificadores encontrados en el programa, la estructura empleada para este propósito es conocida como la tabla de símbolos.

El *optimizador* es una fase opcional, diseñada para mejorar el código intermedio logrando que el programa objeto corra más rápido y/o ocupe menos espacio en memoria. Su salida es un programa equivalente en CI. En *APC* no está incluída esta fase ya que no se busca la eficiencia en este aspecto.

El *generador de código* produce el código objeto, tomando en cuenta los registros del procesador en los cuales se llevarán a cabo las operaciones, así como las localidades físicas de la memoria en las que se almacenan los datos de las variables. Esta fase tampoco se implantó dentro de *APC*, en su lugar se implantó el intérprete-depurador.

El compilador identifica y analiza, por separado, tres tipos de construcciones sintácticas: las declaraciones, el control de flujo y las expresiones.

Cuando el compilador detecta declaraciones como, por ejemplo, de una variable, genera una entrada en la tabla de símbolos, donde se describen las características de la variable declarada. Cuando analiza el control de flujo, el compilador genera una tabla de código intermedio, posteriormente usada por el intérprete.

Al detectar expresiones se determina la dirección de dónde comienza y de dónde termina, con la finalidad de evaluarla directamente del texto. Estas direcciones constan de dos números, el renglón y la columna dentro del archivo de texto.

El compilador de *APC* se comporta de manera diferente dependiendo de lo que llamamos *modos de operación*, que pueden ser: i) cuando analiza texto o modo *statement* y ii) cuando evalúa expresiones.

En esta tesis únicamente se hablará de modo statement, que es el encargado de generar el código intermedio.

3.1 Gramática

"La especificación sintáctica de un lenguaje de programación se hace por medio de una notación llamada **Gramática libre de contexto** (Gramática para abreviar), también se llama BNF de *(Backus-Naur Form)*. Esta notación tiene muchas ventajas como método de especificación de la sintaxis de un lenguaje" [Aho 77]. Una gramática consta de cuatro componentes: terminales, no terminales, 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 no terminales son variables sintácticas que denotan conjuntos de cuerdas. Los no terminales 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 no terminal se distingue como el símbolo de inicio y 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 no terminales para formar cuerdas. Cada producción consiste en un no terminal seguido por una flecha, seguido por una cuerda de no terminales y terminales.

Como una convención, la producción para el símbolo de inicio se lista primero. Las producciones con el mismo no terminal en la izquierda pueden tener sus lados derechos agrupados, con los lados derechos alternativos separados por el símbolo | , el cual es la conjunción.

Gramática de APC

Cuando se inició el proyecto *APC*, no se tenía acceso a una descripción detallada de la gramática del lenguaje *C* y la fuente de información tradicional para estudiarlo era el libro *The C Programming Languaje*, escrito por los autores del lenguaje Brian Kernighan y Dennis Ritchie (Prentice-Hall, 1978).

Otra fuente de información del lenguaje era el compilador mismo *(PCC o Portable C Compiler)*, con su ayuda se podían determinar algunas características no documentadas, principalmente de la semántica del lenguaje.

En el año de 1981, se publicó en *ACM* un artículo llamado *C: Toward a Concise Syntactic Description*, de Patrick A. Fitzhorn y Gearold R. Johnson, del cual se tomó la primera gramática base para este trabajo.

Descripción sintáctica de K&R.

La descripción efectuada en [Kern 78] es inconsistente y contradictoria con respecto al PCC. Por citar algunos casos analicemos la siguiente definición de una función valida en C.

```
extern unigned long int VALIDA () {
    ... Cuerpo de la función
}
```

Como se puede comprobar, usando el propio *PCC*, la función es perfectamente válida, ahora analicemos el siguiente fragmento de la descripción *BNF*, publicada en *[Kern 78]*.

```
<Function definition> ::= <type specifier> <function declaration> <function body>
<type specifier>      ::= char
                          short
                          int
                          long
                          unsigned
                          float
                          double
                          struct
```

Resulta obvio que la función *VALIDA* () no puede producirse a partir de la descripción anterior. Ahora consideremos la siguiente función invalida:

```
int INVALIDA () [5] {
    ... Cuerpo de la función ..
}
```

De la definición anterior, *INVALIDA* es una función que regresa un arreglo de 5 enteros, pero las funciones en *C* sólo pueden regresar: apuntadores a (estructuras, uniones, arreglos o cualquier otro tipo, así como a funciones), tipos básicos y algunos compiladores, como el *PCC*, permiten que una función regrese estructuras. De ningúna manera se permite regresar arreglos. Pero sin embargo, en la descripción *BNF* de [*Kern 78*] se tiene que:

```
<function declarator> ::= <declarator> ( <parameter list> opt )
<declarator>          ::= identifier
                          ( <declarator> )
                          * <declarator>
                          <declarator> ()
                          <declarator> [ <constant expression> opt ]
```

Es claro que la función *INVALIDA* definida con anterioridad puede ser construida a partir de las reglas anteriores.

Por supuesto, el compilador *PCC* no acepta dicha construcción. De cualquier forma, no es posible usar la descripción de *K&R* para hacer un compilador de C.

Descripción sintáctica de F&J

La gramática presentada en *[Fitz 81]*, fue nuestro punto de partida para obtener la gramática usada en *APC*. Aunque quedaban por resolver

los siguientes problemas: i) al igual que la de *K&R*, tenía algunos errores y ii) no era una gramática del tipo *LALR*(1) que requeríamos para poder usar *YACC*.

Entre los errores que se corrigieron está el de no poder generar expresiones con subíndices, de la forma:

m[i] [j]

El proceso de transformación a LALR(1) fue muy laborioso y se hizo en paralelo con respecto a la corrección de errores, muchas veces al hacer un nuevo ajuste, se introducía un nuevo error y al corregir éste dejaba de ser LALR(1), al final de éste proceso, se comprobó experimentalmente que la gramática obtenida era aceptable. El mecanismo de prueba fue reunir una buena cantidad de programas fuentes recopilados de varios libros, entre ellos el fuente de un compilador y algunos programas de comunicaciones, alrededor de 70,000 líneas de código en total. Se implantó un analizador que simplemente revisaba la sintaxis empleando nuestra gramática y se usó para revisar aquellas 70,000 líneas de código. Se sabe que este experimento no demuestra de manera irrefutable que nuestra gramática es correcta, sin embargo se dio por buena, ya que el objetivo de este trabajo no era el de encontrar una gramática de *C*.

Gramática para el intérprete-depurador.

Con el objeto de aprovechar al máximo la memoria y hacer un uso más eficiente de *YACC*, se decidió tener una sola gramática que tenían que compartir el compilador y el intérprete. Esto implicó hacer algunos ajustes a la gramática, tomando en cuenta las siguientes consideraciones: i) la sección de la gramática que genera declaraciones, sólo el compilador la usa, por lo tanto no resultó modificada, lo mismo sucedió con la sección de control de flujo (if, while, for, etc), ii) la sección de expresiones tanto el intérprete como el compilador la usan, por lo que se definió una variable global que indicaba si la gramática la estaba usando el compilador o el intérprete, separando así las acciones semánticas de los dos módulos. Por lo que, toda acción semántica de una expresión tenía el siguiente código:

```
if (modo == COMPILER)  {
                           /* acciones semánticas del compilador */
                       }   else {
                           /* acciones semánticas del intérprete  */
                       }
```

iii) Finalmente, se agregó una sección más a la gramática con el objeto de interpretar el código de una función, a continuación se lista:

```
nvalue      :   ID (expL) { call 0; } code
            ;
code        :   /* vacia */
            |   gen_expL        { ret 0; }
            ;
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
            ;
```

3.2 Descripción del código intermedio

Se han definido nueve códigos de operación que reconoce el intérprete, los cuales son:

- *_IF* Código de operación condicional. Sus dos direcciones apuntan a una expresión, si es verdadera debe saltarse dos cuádruples y si es falsa se salta sólo uno.
- *_GOTO* Salto incondicional. De sus dos direcciones, sólo toma la primera, la cual interpreta como el índice del quadruple siguiente.
- *_RET* Código de retorno de una función. Sin regresar valor, no usa ninguna de sus direcciones.

- *_RETV* Código de retorno de una función. Regresa el valor que evalúa de la expresión apuntada por sus dos direcciones.
- *_SWITCH* Código de apoyo para la generación del statement switch. Sus dos direcciones apuntan a la expresión del switch y su valor se compara con las constantes del case.
- *_CASE* Código de apoyo para la generación del statement switch. Usa solamente su primera dirección y es el valor de la constante del case.
- *_DEF* Código de apoyo para la generación del statement switch. No usa ninguna de sus direcciones y se maneja básicamente como un código de relleno (consulte la parte correspondiente al switch en la generación de código.)
- *_E* Código de evaluación de una expresión. Sus direcciones apuntan a la expresión a evaluar.
- *_E_USER* Es un código similar al anterior, con la diferencia de que la expresión a evaluar la toma del buffer de usuario y no del texto del programa.

3.3 Tabla de símbolos

tabla de hash

Se usó el método de *hash* o de la función de dispersión para buscar rápidamente en las tablas de símbolos.

Este método requiere de un vector o arreglo de apuntadores, así como una función de dispersión que regresa un valor que calcula aplicando un algoritmo muy sencillo a la cuerda o nombre del símbolo en cuestión.

Al inicio del programa, el vector de apuntadores debe estar inicializado con *NULL* en cada uno de sus elementos, esto significa que la tabla de *hash* está vacía.

Cuando se inserta un símbolo nuevo: i) se calcula su valor *h* de dispersión utilizando la función destinada a este propósito; ii) se verifica que en el lugar indexado por *h* el vector de apuntadores contenga un *NULL*, si no hay *NULL* éste es buscado en los lugares siguientes hasta encontrarlo;, iii) se crea el nuevo símbolo y se asigna su dirección en el vector de apuntadores.

Cuando se busca un símbolo: i) se calcula su valor *h* de dispersión, ii) si en la localidad del vector indexada por *h* hay un *NULL*, entonces

no existe el símbolo buscado, por el contrario se compara el símbolo buscado, con el apuntado por el vector, si son iguales ya se encontró, si son diferentes se busca en la siguiente localidad y se repite esta operación hasta que se encuentra un *NULL*.

Las funciones relacionadas con la tabla de *hash* son:

NOMBRE

hash

SINOPSIS

short hash (s) /* función de dispersión */
*char * s;*

PARAMETROS

s es la cuerda de entrada a la función.

VARIABLES

_IDSIZE es el tamaño de un identificador
shash es el tamaño de la tabla de hash.

FUNCION

Regresa un valor entre 0 y *shash* en función a la cuerda de entrada *s*.

Tabla de símbolos

La tabla de símbolos es una estructura de datos en la que se almacenan cada uno de los nombres de las variables y funciones del programa fuente que se compila, así como algunas características asociadas a estos nombres.

En el caso de *APC* hay dos clases de tablas de símbolos, la global y la particular de cada función definida en el programa fuente. En la tabla de símbolos global se guardan las variables globales y los nombres de cada una de las funciones. La información almacenada en las tablas de símbolos particulares son: los parámetros de la función y las variables automáticas declaradas en dicha función.

El acceso a un símbolo cualquiera es por medio de las funciones *findsym* y *creatsym*, no importando si el símbolo está en la tabla global o particular.

Para lograr este efecto, el acceso a las tablas se hace por medio del apuntador *phash* que señala a la tabla de interés, en la variable *shash* se asigna el tamaño, ya que es diferente el de la tabla global que el de la particular.

En esta sección se describirán las funciones del compilador encargadas del manejo de la tabla de símbolos, así como su estructura.

NOMBRE:

findsym

SINOPSIS

*symT * findsym (s)*
*char * s;*

PARAMETROS

s es la cuerda correspondiente al símbolo buscado

VARIABLES

phash es un apuntador a la tabla de hash en uso, puede ser la correspondiente a la tabla de símbolos global o a la tabla de símbolos particular de una función.

_IDSIZE es el tamaño de un identificador shash y correspondeal tamaño de la tabla de hash.

FUNCION

Regresa la dirección de la tabla de símbolos donde encontró el símbolo buscado o un NULL si no lo encontró.

NOMBRE

creatsym

SINOPSIS

symT * creatsym (s)
char * s;

PARAMETROS

s es la cuerda de entrada a la función.

VARIABLES

nsym número actual de símbolos usados.
MAXSYM número máximo de símbolos permitidos.
phash mismo comentario de la función anterior.
symbol [] área de símbolos nuevos disponibles.

FUNCION

Crea una entrada en la tabla de hash para el símbolo correspondiente a la cuerda
si ya no queda espacio en la tabla aborta la ejecución del programa.

Formato de las tablas de símbolos

NOMBRE

symT

SINOPSIS

```
typedef struct {
  char  *sname;    /* NOMBRE                 */
  ushort satt;     /* ATRIBUTOS              */
  ushort ssize;    /* TAMAÑO                 */
  char   *slval;   /* VALOR IZQUIERDO        */
  char   *sdesc;   /* DESCRIPTOR             */
} symT;
```

FUNCION

Definición del tipo de la tabla de símbolos.

En la siguiente figura se muestra el formato de la tabla de símbolos.

Indice	Nombre	Atributos	Tamaño	Valor	Descriptor
0					
1					
2					
.					
.					
.					
MAXSYM					

Donde :

- **NOMBRE**

contiene un apuntador a un área de memoria en donde se almacena una cuerda de caracteres correspondiente al nombre del símbolo.

- **ATRIBUTOS**

es un campo de 16 bits organizado como se muestra a continuación:

modificadores												tipo			
5	5	4	4	3	3	2	2	1	1	0	0				

en los seis campos de dos bits se registra lo que denominamos modificadores y pueden ser:

POINTER, FUNCION, ARREGLO y NADA;

Estos modificadores se deben leer de derecha a izquierda y determinan exactamente lo que significa cada símbolo. En los cuatro bits menos significativos se almacena el tipo básico, por ejemplo:

*char * p ();*

esta declaración indica que p es una función que regresa un apuntador a carácter, sus atributos tienen:

NADA NADA NADA NADA POINTER FUNCION CHAR

int (f) ();*

en este caso se está declarando que f es un apuntador a una función que regresa un entero, sus atributos son:

NADA NADA NADA NADA FUNCION POINTER INT

- **TAMAÑO**

  Este campo tiene un significado diferente dependiendo el tipo del símbolo. Para los apuntadores se almacena el tamaño del objeto a que apunta, en los arreglos está el tamaño total del arreglo, considerando

  todas sus dimensiones y en las funciones el número de instrucciones generadas.

- **VALORI**

  O valor izquierdo (cuando un símbolo tiene la capacidad de almacenamiento); en este lugar se tiene un apuntador al área donde se almacena el valor de cada variable. En el caso de funciones, tiene la dirección del segmento de código correspondiente a ésta. Cuando el símbolo es el nombre de un arreglo, estará la dirección al principio de éste, en el área de memoria.

- **DESCRIPTOR**

Es un apuntador a un descriptor, éste sólo tiene sentido cuando se trata de un arreglo, una función o una declaración en la que existan dimensiones de arreglo. En la sección de la estructura de funciones y arreglos se describirá este campo detalladamente.

Las siguientes funciones son las encargadas del manejo de las tablas de símbolos:

NOMBRE

initsym

SINOPSIS

void initsym ();

VARIABLES

nsym	*número de símbolos creados*
nfunc	*número de funciones reconocidas*
stat_mode	*bandera de modo statment*
allocp	*apuntador a buffer de memoria disponible*
hglob []	*tabla de hash global*
symbol []	*buffer de símbolos*
fund []	*buffer de descriptor de funciones*
phash	*apuntador a la tabla de hash en uso*
shash	*tamaño de la tabla de hash en uso*
ccode	*apuntador a la tabla de código en uso*
nextquad	*índice de la tabla de código*
mfree	*índice de memoria usada en listas ligadas*
declarations	*indica si son globales o locales*
glob_offset	*aparta área de memoria para variables globales*

FUNCION

Inicializa las variables relacionadas con el manejo de los símbolos.

NOMBRE

settype

SINOPSIS

```
int settype (t,sym)
short t;
symT *sym;
```

PARAMETROS

t *tipo básico*

sym *apuntador al símbolo*

FUNCION

Asigna un tipo básico al símbolo apuntado por sym y regresa OK o ERROR.

NOMBRE

enter

SINOPSIS

*symT * enter (s)*

*char * s;*

PARAMETROS

s *apuntador al nombre del símbolo*

VARIABLES

newsym Bandera, se prende si el símbolo es nuevo, la variable where Indica si la tabla de hash que se usó fue la GLOBAL o la LOCAL.

FUNCION

Regresa la dirección del símbolo cuyo nombre es apuntado por el parámetro

s, si el símbolo no está registrado, lo crea.

Variables tipo básico.

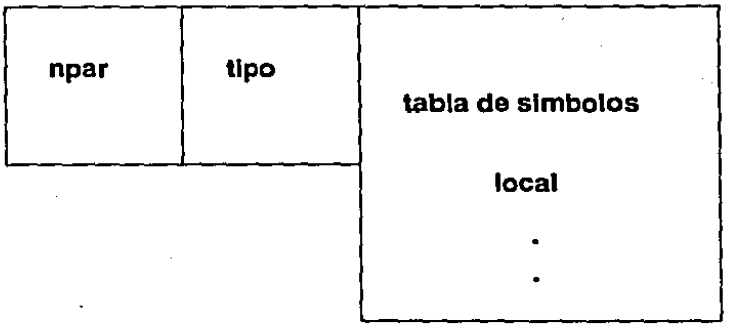
Una variable de tipo básico es aquella que no tiene los modificadores () función, [ ] arreglo o * apuntador en su declaración.

Cuando se detecta una declaración de este tipo de variables, simplemente se da de alta en la tabla de símbolos correspondiente, teniendo cuidado de dejar sus atributos en forma correcta.

Funciones

Cada vez que el parser reconoce una función, se genera en la tabla de símbolos global una *entrada*, en el campo *valori* se deja un apuntador al código de la función y en el campo *descriptor* está la dirección de éste, con la siguiente estructura:

Descriptor de funciones



Donde :

- **npar.-** *Es el número de parámetros definidos en la función.*
- **tipo.-** *Indica el tipo de dato que regresa la función.*
- **tabla de hash de símbolos.-** *Similar a la tabla de hash global, sólo que contiene las variables y parámetros internos de la función.*

Tabla de código de funciones

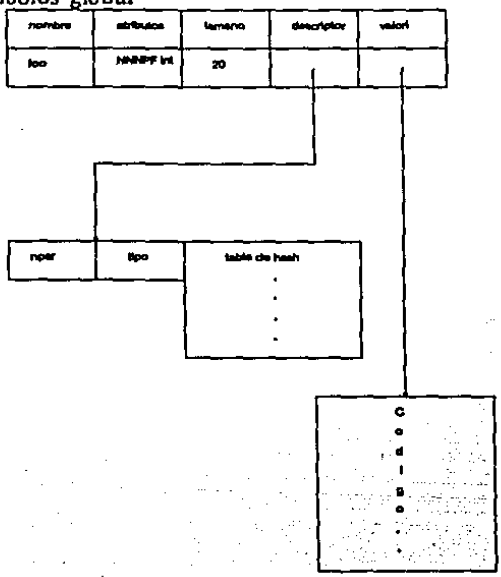
Esta es una tabla de tres columnas, la primera es el código de operación, la segunda y tercera es un apuntador al principio y final de una expresión por ejemplo:

Indice	Código po	Apuntador principio	Apuntador final	Comentarios
0	E	(12, 4)	(24, 5)	Evalúa la expresión
i	IF	(12, 5)	(15, 5)	Evalúa si es verdadera ejecuta i +1 si no i +2
j	GOTO	(k, 0)	(0, 0)	Ejecuta lo que está en índice k
k	RETV	(13, 8)	(30, 8)	Regresa como valor de la función la expresión
l	REV	(0, 0)	(0, 0)	Regresa de la función

Los apuntadores a las expresiones están compuestos del renglón y la columna en el archivo de trabajo.

La relación entre las estructuras que se emplean en la representación de las funciones se muestra a continuación:

Tabla de símbolos global



Arreglos

En el caso de reconocer un arreglo, el parser crea su entrada en la tabla de símbolos correspondiente (local o global) y los campos quedan con la siguiente información:

- *ssize*: tamaño del arreglo en bytes
- *sival*: lugar donde empieza el arreglo, como en el caso de una variable de tipo básico.
- *sdec*: apuntador al descriptor del arreglo.

Descriptor de arreglos

NOMBRE

arrdT

SINOPSIS

```
typedef struct {
tiny ndim;
ushort dim[4][2];
} arrdT;
```

FUNCION

Definición del tipo arrdT usado para el manejo del descriptor de los arreglos. Básicamente, este descriptor define un área de memoria con la siguiente estructura:

ndim				dim [i] [0]
				dim [i] [1]

Supongamos la siguiente declaración de un arreglo como sigue:

```
char mat [3][5][2];
```

entonces su descriptor sería:

3	3	5	2	-	dim [i] [0]
	10	2	1	-	dim [i] [1]

ndim *corresponde, como ya se mencionó, al número de dimensiones y en este ejemplo es 3.*

dim[i][0] *es donde se guarda el número de elementos de cada dimensión.*

dim[i][1] *contiene el tamaño del objeto al cual apunta el pointer correspondiente. Dado que la expresión:*

mat

es un pointer que apunta al primero de tres objetos de tamaño 10, así como la expresión:

mat [0]

es un pointer que apunta al primero de cinco objetos de tamaño 2 y la expresión:

mat [0][0]

es un pointer que apunta al primero de dos objetos de tamaño 1 y por ultimo la expresión:

mat [0][0][0]

es el primer objeto almacenado en la matriz.

Apuntadores

Los apuntadores se tratan de la misma forma que las variables de tipo básico descritas con anterioridad, con excepción de las involucradas con arreglos, en cuyo caso se crea un descriptor con las mismas características del descriptor de arreglos.

3.3 Análisis léxico.

Elementos lexicográficos de C.

Alfabeto

i) Cincuenta y dos caracteres alfabéticos.

{ A .. Z a .. z }

ii) Veintinueve caracteres gráficos.

{ ! + " # = { % } ^ [, &] . * \ < (| >) ; / - : ? _ ' }

iii) Diez dígitos decimales

{ 0 .. 9 }

iv) El carácter blanco.

{ }

v) El carácter de fin de línea.

{ LF }

vi) Seis caracteres de formateo, correspondientes al código ASCII: *Back Space, Horizontal Tab, Vertical Tab, Form Feed, y Carriage Return.*

{ BS HT VT FF CR }

Comentarios

Un comentario, en un texto fuente de C, empieza con los caracteres /* y termina con */. Los comentarios pueden tener cualquier número de caracteres y son equivalentes a escribir un carácter blanco.

Tokens

Los operadores de un solo carácter en *C* son :

{ ! % ^ & * - + = ~ | . < > / ? }

Operadores compuestos :

{ -> + + - - < < > > < = > = = = != && || + = - = * = / = % = < < = > > = ^ = | = }

Separadores :

{ ( ) [ ] { } , ; : }

Nota: Algunos de estos separadores se emplean también como operadores, como es el caso del carácter correspondiente a la coma, así como en forma compuesta los operadores () y [].

Identificadores

Un identificador es una secuencia de letras, dígitos y el carácter de subrayar. Para indicar esto en forma más adecuada, se emplea la nomenclatura de las expresiones regulares descrita en *<LEX>*

identificador:: = primer_carácter { siguiente_carácter }*.

primer_carácter :: = letra | subrayado

siguiente_carácter :: = letra | subrayado | dígito

letra :: = A .. Z a .. z

subrayado :: = _

dígito :: = 0 .. 9

Palabras reservadas

Los siguientes identificadores son usados como palabras reservadas en *C* y no deben usarse como identificadores dentro de un texto fuente.

auto	else	long	typedef
break	enum *	register	union *
case	extern	return	unsigned
char	float	short	void *
continue	for	sizeof	while
default	goto	static	
do	if	struct	
double	int	switch	

Nota : las palabras reservadas marcadas con * no son estándar y además no están consideradas en esta versión.

Constantes

Se pueden distinguir cuatro clases de constantes en C.

constante ::= constante_entera
| constante_flotante
| constante_carácter
| constante_cuerda

Constantes enteras

constante_entera ::= decimal
| octal
| hexadecimal

decimal ::= nocero_dígito { dígito }* { marca_long }?

octal ::= 0 { dígito_*octal* }* { marca_long }?

hexadecimal:: = marca_hex {dígito_hexa} + { marca_long }?

dígito ::= 0 .. 9

nocero_dígito ::= 1 .. 9

dígito_octal ::= 0 .. 7

dígito_hexa ::= 0 .. 9 A .. F a .. f

marca_long ::= l | L

marca_hex ::= 0x | 0X

Constantes de punto flotante

constante_flotante ::=secuencia exponente |dígito2 secuencia

exponente ::= (e | E) { + | - }? secuencia

dígito2 ::= secuencia . | secuencia . secuencia |. secuencia

secuencia ::= { dígito } +

Constantes tipo carácter

constante_carácter ::= *carácter*

carácter ::= carácter_imprimible | carácter_escape

carácter_imprimible ::= { *ASCII*(32) .. *ASCII*(44) } u { *ASCII*(46) .. *ASCII*(91) } u { *ASCII*(93) .. *ASCII*(126) }

Nota : La notación *ASCII*(x) significa el carácter correspondiente a la tabla ASCII en el lugar x en decimal, así que el conjunto descrito como caracteres imprimibles agrupa desde el carácter blanco hasta el ~, excepto los caracteres ' y \ .

carácter_escape ::= \ código_escape

código_escape ::= mnemónico | escape_numérico

mnemónico ::= n | t | b | r | f | v | \ | ' | "

escape_numérico ::= dígito_octal { dígito_octal { dígito_octal }? }?

Constantes tipo cuerda

constante_cuerda ::= " { carácter }* "

carácter ::= carácter_imprimible | carácter_escape.

Implantación

Dado que el proceso de programar scanners está bien entendido, se han construido herramientas que los generan automáticamente a partir de una descripción de los tokens que deben aceptar. Uno de estos generadores es el programa *LEX*, el cual se usó en esta tesis.

Los scanners generados por *LEX* no son muy eficientes por lo que se emplean comúnmente para desarrollar prototipos rápidos de un proyecto.

La especificación que usa *LEX* es sumamente simple, de hecho es un conjunto de expresiones regulares seguidas de la acción semántica que se debe ejecutar en el momento de reconocerlas.

Un problema que se detectó fué, que el scanner generado por *LEX* lee los caracteres de entrada por medio de la función *GETC* lo que implica leer siempre de un archivo, o bien de la terminal, pero en nuestro caso se requería los leyera del editor. La solución a este problema fué modificar el programa fuente y substituir *GETC* por una función

de interface con el editor *nextchar ()*. Otro cambio que se hizo al programa fué para guardar la dirección del texto antes de reconocer el siguiente token, introduciendo la función *save_txt ()*, esto debido a que muchas veces es necesario saber dónde empieza un token y no dónde termina.

3.4 Análisis sintáctico y acciones semánticas

Para hacer la generación del código intermedio se emplearon las siguientes estructuras de datos y funciones:

NOMBRE

quadT y codeT

SINOPSIS

```
typedef struct {
    tiny op; /*     código de operación  */
    long a1; /*     dirección primera    */
    long a2; /*     dirección segunda    */
} quadT, * codeT;
```

FUNCION

Definición de los tipos *quadT*, estructura para representar código de tres direcciones (*cuádruples* [Aho79]) y *codeT*, que es de tipo apuntador a estructuras *quadT* usado en el manejo del código intermedio.

NOMBRE

struct list

SINOPSIS

```
struct list {
    short       indx;
    struct list *nxt;
};
```

FUNCION

declaración de la estructura para el manejo de listas de índices.

NOMBRE

memcode, ccode, nextquad

SINOPSIS

quadT memcode [*MAXCODE*];

codeT ccode = memcode;

short nextquad;

FUNCION

memcode área en la memoria para guardar los segmentos de código intermedio. *ccode* apuntador al segmento de código intermedio en uso. *nextquad* índice del siguiente *cuádruple*.

NOMBRE

makelist

SINOPSIS

struct list * makelist (index)

short index;

PARAMETROS

index el índice del *cuádruple* que contendrá la lista.

FUNCION

Crea una nueva lista de índices, regresando su dirección o *NULL* si ya no queda más memoria. Cada índice representa un *cuádruple* o código de tres direcciones.

NOMBRE

merge

SINOPSIS

struct list * merge (p1, p2)
struct list *p1, *p2;

PARAMETROS

p1, p2 Apuntadores a dos listas de índices diferentes.

FUNCION

Toma las listas apuntadas por *p1* y *p2* y las concatena, regresando un apuntador a la lista concatenada.

NOMBRE

backpatch

SINOPSIS

void backpatch (p, i)
register struct list * p;
short i;

PARAMETROS

p Apuntador a una lista de índices.
i índice.

VARIABLES

ccode Apuntador al segmento de código en uso.

FUNCION

Hace que cada uno de los *cuádruples* contenidos en la lista apuntada por *p* tome el índice *i* en su dirección *a1*. Expresado en *C* quedaría:

```
for (; p; ccode[p->indx].a1 = i, p = p->nxt);
```

NOMBRE

gen

SINOPSIS

void gen (code, ad1, ad2)

char code;

long ad1, ad2;

PARAMETROS

code Código de operación {*IF, GOTO, RET, RETV, SWITCH, CASE, DEF, E, E_USER*} ad1, ad2 direcciones.

VARIABLES

ccode, apuntador al segmento de código en uso.

nextquad, índice del siguiente *cuódruple* sobre el segmento de código

FUNCION

crea un nuevo *cuódruple* en el índice *nextquad*, asignándole *code*, *ad1*, y *ad2*; además incrementa *nextquad*

A continuación se describe el proceso en la generación de código intermedio relacionado al control de flujo del programa.

La mecánica que se empleará en esta descripción será mostrar una parte de la gramática correspondiente (con la misma sintaxis de *YACC*) y las acciones semánticas más importantes en un pseudocódigo tipo *C*.

A cada uno de los símbolos gramaticales, *YACC* permite asociarles un valor que puede usarse una vez que se ha reconocido la regla.

Las reglas de la gramática de *statements*, tienen asociados tres diferentes tipos de valores: índices, listas y apuntadores a expresiones, aunque hay algunas reglas que no tienen ningún valor asociado.

Una expresión del tipo '$$= 33' significa que a esa regla se le está asociando el valor 33, o también se puede decir que la regla regresa 33.

Una expresión del tipo '$n' hace referencia al valor asociado con el símbolo n-ésimo de la regla.

Las acciones semánticas de una regla están delimitadas por los caracteres {}.

Comencemos por la primera parte de la gramática de *statements*.

```
1    M_nextQuad:
2                        {
3                            $$ = nextquad;
4                        }
5                        ;
```

M_nextQuad, está definido como símbolo no terminal vacío y sirve únicamente como una marca que proporciona el valor del índice del siguiente *cuádruple*.

Se ha usado como convención que cuando el nombre de una regla termina con *L* indica que es una lista, en este caso es una lista de *statements*.

```
6    statL:      stat
7            |   statL M_nextQuad stat
8            {
9                backpatch ($1, $2);
10               $$ = 3;
11           }
12           ;
```

Cuando en una regla no se especifica acción semántica, *YACC* asume la acción { $$ = $1; }, que es el caso de la primera parte de *statL* (línea 6).

Los valores asociados a los símbolos *statL* y *stat* son la lista de *cuádruples* con código *GOTO* que tienen pendiente la dirección del *statement* que sigue en su secuencia de ejecución.

Una vez que se reconoce la regla del renglón 7, se arregla su lista de pendientes con la llamada a la función *backpatch*, haciendo que sus *GOTO* tengan la dirección del siguiente *statement*, en este caso es la entrada a *stat* renglón 7.

```
13   stat:          compound_stat
14                | loop
15                | switch_stat
16                | cond_stat
17                | action
18                | null
19                | expresión ';'
20                {
21                    gen (_E, $1.Beg, $1.End);
22                    $$ = NULL;
23                }
24                ;
```

Como se observa del fragmento anterior, *stat* es un símbolo no terminal donde agrupan todos los casos de *statements* del lenguaje, y regresa la lista que cada uno de ellos le proporciona por medio de la acción semántica por omisión de *YACC*. A continuación se describen las acciones que se llevan a cabo con cada una de los statements agrupados en la regla del renglón 13.

```
25   M_begin:
26                {
27                    stat_mode + +;
28                    kill_bterr 0;
29                }
30                ;
31   M_end:       .
32                {
33                    stat_mode --;
34                }
35                ;
```

La marca de principio de *statement M_begin* (línea 25), es un símbolo no terminal vacío, que se usa para manejar un contador, el cual indica

el nivel de anidamiento de bloques de *statement*. Cuando este contador es cero, significa que se estan analizando las declaraciones del símbolo *internalDdec* (renglón 36) esto es importante ya que la función *enter* no debe dar de alta símbolos cuando *stat_mode* > *0* , es decir, cuando analiza *statements*.

Debido a que *YACC* lee un token adelantado para poder determinar así el estado de su autómata, y que el scanner es el encargado de instalar en la tabla de símbolos los identificadores, es necesario hacerle un ajuste, ya que se pudo dar de alta un símbolo indebidamente. La función *kill_bterr ()* hace este ajuste.

```
36  compound_stat:      '{' InernalDdec M_begin statL M_end '}'
37                  {
38                      $$ = $4
39                  }
40                  ;
```

La regla de la línea 36 simplemente regresa la lista proporcionada por el símbolo *statL*.

```
41  M_while:
42              {
43                  $$ = nextquad;
44                  gen (_IF, 0, 0);
45                  gen (_GOTO, 0, 0);
46                  if (loopi++ >= MAXLEVEL){
47                          yyerror ("While demasiado anidado");
48                                  YYERROR;
49                  }
50              }
51              ;
52  M_do:
53                      {
54                          $$ = nextquad;
```

```
55                          if (loop++ >= MAXLEVEL) {
56                              yyerror ("Do demasiado anidado);
57                                  YYERROR;
58                          }
59                      }
60                      ;

61  M_for:
62                      {
63                          $$= nextquad;
64                          gen (_E, 0, 0);
65                          gen (_IF, 0, 0);
66                          gen (_GOTO, 0, 0);
67                          if (loop++ >= MAXLEVEL){
68                              yyerror ("For demasiado anidado");
69                                  YYERROR;
70                          }
71                      }
72                      ;
```

Las marcas *M_while, M_do y M_for* se consideran inmediatamente después de reconocer las palabras reservadas *while, do* y *for*, respectivamente, y son usadas para generar la primera parte del código de cada statement, regresan el índice del siguiente *cuádruple*. Además controlan la variable *loopl* que indica el nivel de anidamiento, ya que los *statements break* y *continue* actúan en el contexto del *loop* donde están definidos.

Generación del código de la regla *loop* (renglón 14):

```
73  loop:       WHILE M_while '(' expresión ')' M_nextQuad stat
74          {
75              ccode[$2].a1 = $4.Beg;
76              ccode[$2].a2 = $4.End;
77              backpatch ($7, $2);
78              backpatch (cont[loopl], $2);
```

```
79              $$ = merege (brks[loopl], makelist ($2+1));
80              gen (_GOTO, $2, 0);
81              loopl--;
82          }
```

El código generado para un statement *while* tiene el siguiente templete:

INDICE	CODIGO	DIR1	DIR2	REFERENCIA EN LA GRAMATICA
m	IF	exp.Beg	exp.End	renglón 44, 75, 76
m +1	GOTO	n + 1*	————	renglón 45, 79 * pendiente
CODIGO DEL STAT DEL RENGLON 73				
las salidas de este stat, hacen un GOTO al *cuádruple* con índice *m*				renglón 77
los *break* de este stat, se forman en la lista de salidas pendientes que harán un GOTO al *cuádruple* de n + 1				renglón 79
los *continue* de este stat, hacen un GOTO a *m*				renglón 78
n	GOTO	m	————	renglón 80
n + 1	????	???	————	primer código del siguiente stat

Por ultimo, se decrementa el contador de nivel de anidamiento de loops (renglón 81), esta regla regresa su lista de salidas pendientes para que sean arregladas en el siguiente statement, (renglón 79).

A continuación se presentan los fragmentos de gramática correspondientes a los statemets *do* y *for*, que siguen los mismos lineamientos que el *while* que se detalló anteriormente.

```
83          |   DO M_do stat M_nextQuad WHILE '('expresión ')' ';'
84          {
85              backpatch ($3, $4);
86              backpatch (cont[loopl], $4);
87              gen (_IF, $7.Beg, $7.End);
88              $$ = merege (brks[loopl], makelist (nextquad));
89              gen (_GOTO, 0, 0);
```

```
90              gen (_GOTO, $2, 0);
91              loopl--;
92          }
```

El templete de la generación de código del *do* es:

INDICE	CODIGO	DIR1	DIR2	REFERENCIA EN LA GARMATICA
m				
CODIGO DEL STAT DEL RENGLON 83				
las salidas de este stat, hacen un GOTO al *cuádruple* con índice *n*				renglón 85
los *break* de este stat , se forman en la lista de salidas pendientes que harán un GOTO al *cuádruple* n + 3				renglón 88
los *continue* de este stat, hacen un GOTO a *n*				renglón 87
n	_IF	exp.Beg	exp.End	renglón 87
n + 1	GOTO	n + 3*	———	renglón 88, 89 * pendiente
n + 2	GOTO	m	———	renglón 90
n + 3	????	???	———	primer código del siguiente stat

Generación del código de la instrucción *for*:

```
93          |   FOR M_for '(' oexp ';' oexp ';' oexp ')' stat
94          {
95              ccode[$2].a1 = $4.Beg;
96              ccode[$2].a2 = $4.End;
97              ccode[$2 + 1].a1 = $6.Beg;
98              ccode[$2 + 1].a2 = $6.End;
99              $$ = merge (brks[loopl], makelist ($2 + 2));
100             backpatch (cont[loopl], nextquad);
101             backpatch ($10, nextquad);
102             gen (_E, $8.Beg, $8.End);
103             gen (_GOTO, $2 + 1, 0);
104             loopl--;
105         }
```

```
106                ;
```

El templete de esta instrucción es:

INDICE	CODIGO	DIR1	DIR2	REFERENCIA EN LA GRAMATICA
m	E	oexp.Beg	oexp.End	renglón 64, 95, 96
m +1	IF	oexp.Beg	oexp.End	renglón 65, 97, 98
m +2	GOTO	n +1 *	————	renglón 66, 69 * pendientes
CODIGO DEL STAT DEL RENGLON 93				
las salidas de este stat, hacen un *GOTO* al *cuádruple* con índice *n*				renglón 101
los *break* de este stat, se forman en la lista de salidas pendientes que harán un *GOTO* al *cuádruple n +2*				renglón 99
los *continue* de este stat hacen un GOTO a *n*				renglón 100
n	E	oexp.Beg	oexp.End	renglón 102
n +1	GOTO	m +1	————	renglón 103
n +2	????	???	————	primer código del siguiente stat

Generación del código de la instrucción *switch* (renglón 15) :

```
107    M_sw:
108                {
109                    swit1 + +; loop1 + +;
110                    $$ = nextquad;
111                    gen (_SWITCH, 0, 0);
112                    nextcase[swit1] = 0;
113                }
114                ;
```

El *switch* lleva dos controles del nivel de anidamiento, el primero por medio de la variable *swit1* (renglón 109) que sirve para manejar los *case* y *default*. El segundo control, por medio de la variable *loop1* (renglón 109), se encarga del manejo de los *break* dentro del contexto del switch. La marca *M_sw* es reconocida cuando el analizador sintáctico detecta la palabra reservada *switch*, entonces genera la primera parte del templete del código de este *statement*, (renglón 111), también se inicializa

la variable que contiene la dirección del siguiente case de este nivel de anidamiento, (renglón 112).

```
115   M_case:
116                 {
117                     $$ = nextquad;
118                     gen (_CASE, 0, 0);
119                     gen (_GOTO, 0, 0);
120                 }
121                 ;
```

La marca *M_case* es reconocida después de haber detectado la palabra reservada *case* y genera dos *cuádruples*, que se deben manejar de la siguiente manera: el primero, *_CASE* (renglón 118), contendrá en su primera dirección una constante que corresponde a la constante especificada en el programa fuente; el intérprete compara el resultado de evaluar la expresión del *cuádruple _SWITCH* (renglón 111) con la constante de *cuádruple _CASE* (renglón 118), si son diferentes, el *_GOTO* de *cuádruple* generado en el renglón 119 lo lleva al siguiente *case*, si son iguales el intérprete no considera el *_GOTO* y ejecuta las instrucciones correspondientes a ese *case*. Esta marca regresa el índice del *cuádruple* _CASE (renglón 117).

```
122   M_default:
123                 {
124                     $$ = nextquad;
125                     gen (_DEF, 0, 0);
126                 }
127                 ;
```

La marca *M_default* se reconoce inmediatamente después de la palabra reservada, *default*, la cual genera un *cuádruple* con el código de operación *_DEF* ignorado por el intérprete, continuando la ejecución al siguiente *cuádruple*, regresa el índice del *cuádruple* generado.

```
128   switch_stat:  SWITCH M_sw '(' expresión ')' '{'caseL M_nextQuad default '}'
129                 {
130                     ccode[$2].a1 = $4.Beg;
131                     ccode[$2].a2 = $4.End;
132                     if (_default) backpatch ($7, $8 + 1);
```

```
133                 else backpatch ($7, $8);
134                 $$ = merge ($9, brks[loopl]);
135                 loopl--; switl--;
136             {
137             ;
```

La regla del renglón 128 contiene todos los elementos que conforman la sintaxis del *statement switch*, el analizador sintáctico reconoce esta regla cuando detecta la última llave }, en este momento el templete de este statement está prácticamente completo y sólo hace falta hacer algunos cambios, como son decrementar las variables del nivel de anidamiento y regresar su lista de salidas pendientes.

```
138 caseL:          case_stat
139             |
140             {
141                 backpatch ($1, $2+2);
142                 $$ = $3;
143             }
144             ;
145 case_stat::     CASE M_case CONST ':' statL
146             {
147                 ccode[$2].a1 = $3.crval;
148                 if (nextcase[switL])
149                 ccode[nextcase[switl]].a1 = $2;
150                 nextcase[switl] = $2 + 1
151                 $$ = merge ($5, makelist (nextquad));
152                 gen (_GOTO, 0, 0);
153             }
154             ;
155 default:
156             {
157                 ccode[nextcase[switl]].a1 = nextquad;
158                 $$ = makelist (nextquad);
159                 _default = 0;
160             }
161             |   DEFAULT M_default ':' statL
162             {
163                 ccode[nextcase[switl]].a1 = $2;
164                 $$ = $4;
165                 _default = 1;
166             }
```

167 ;

Temple generado por el *statement switch*:

INDICE	CODIGO	DIR1	DIR2	REFERENCIA EN LA GRAMATICA
a	SWITCH	exp.Beg	exp.End	renglón 111, 130, 131
a +1	CASE	cte	———	renglón 118, 147
a +2	GOTO	b	———	renglón 119, 148, 149
CODIGO DEL STAT RENGLON 145				•
las salidas de este stat, hacen un GOTO al *cuádruple* con índice *b +2*				renglón 141
los *break* de este stat, se forman en la lista de salidas pendientes que harán un GOTO al *cuádruple d +m* •				renglón 134, • pendientes
a +n	GOTO	b +2	———	renglón 152, 141
b	CASE	cte	———	renglón 118, 147
b +1	GOTO	c	———	renglón 119, 148, 149
b +2:				
CODIGO DEL STAT DEL RENGLON 145				
las salidas de este stat, hacen un GOTO al *cuádruple* con índice *c +2*				renglón 141
los *break* de este stat, se forman en la lista de salidas pendientes que harán un GOTO al *cuádruple d +m* •				renglón 134, • pendientes
c	CASE	cte	———	renglón 118, 147
c +1	GOTO	d	———	renglón 119, 148, 149
c +2:				
CODIGO DEL STAT DEL RENGLON 145				
las salidas de este stat, hacen un GOTO al *cuádruple* con índice *d +1*				renglón 141
los *break* de este stat, se forman en la lista de salidas pendientes que harán un GOTO al *cuádruple d +m* •				renglón 134, • pendientes
c +n	GOTO	d	———	renglón 152, 141
d	DEFAULT		———	renglón 125
d +1:				

INDICE	CODIGO	DIR1	DIR2	REFERENCIA EN LA GRAMATICA
CODIGO DEL STAT DEL RENGLON 161				
las salidas de este stat, hacen un GOTO al *cuádruple* con índice $d + m$				renglón 164, *134*
los *break* de este stat, se forman en la lista de salidas pendientes que harán un *GOTO* al *cuádruple* $d + m$				renglón 134, * pendientes
d + m				

Generación de código de la regla *cond_stat* (renglón 16) :

```
168  M_if:
169              {
170                  $$ = nextquad;
171                  gen (_IF, 0, 0);
172                  gen (_GOTO, 0, 0);
173              }
174              ;

175  M_else:
176              {
177                  $$ = makelist (nextquad);
178                  gen (_GOTO, 0, 0);
179              }
180              ;
```

Las marcas M_if y M_else dejan una parte del templete del código de los dos tipos del statement condicional *if*.

```
181  cond_stat:      IF M_if '(' expresión ')' stat
182                  {
183                      ccode[$2].a1 = $4.Beg;
184                      ccode[$2].a2 = $4.End;
185                      $$0 merge (makelist ($2 + 1), $6);
186                  }
187                  |  IF M_if '(' expresión ')' stat M_else ELSE M_nextQuad stat
188                  {
189                      ccode[$2].a1 = $4.Beg;
190                      ccode[$2].a2 = $4.End;
```

```
191                 ccode[$2 + 1].a1 = $7-> indx + 1;
192                 $$ = merge ($6, merge ($7, $10));
193   }
194   ;
```

El templete de statement *if* es:

INDICE	CODIGO	DIR1	DIR2	REFRENCIA EN LA GRAMATICA
a	IF	exp.Beg	exp.End	renglón 171, 183, 184
a + 1	GOTO	b *	———	renglón 172, 185, * pendiente
CODIGO DEL STAT DEL RENGLON 181				
las salidas de este stat, se forman en la lista de salidas pendientes que harán un GOTO al *cuádruple b*				renglón 185
b	????	???	———	primer código del siguiente stat

El templete del statement if con else es:

INDICE	CODIGO	DIR1	DIR2	REFRENCIA EN LA GRAMATICA
a	IF	exp.Beg	exp.End	renglón 171, 189, 190
a + 1	GOTO	b + 1	———	renglón 172, 185, * pendiente
CODIGO DEL STAT DEL RENGLON 181				
las salidas de este stat, se forman en la lista de salidas pendientes que harán un GOTO al *cuádruple c*				renglón 192
b	GOTO	c *	———	renglón 178, 192, * pendiente
b + 1:				
CODIGO DEL STAT DEL RENGLON 187 CORRESPONDIENTE AL ELSE				
las salidas de este stat, se forman en la lista de salidas pendientes que harán un GOTO al *cuádruple c*				renglón 192
c	????	???	———	primer código del siguiente stat

Generación de código de la regla *action* (renglón 17) :

```
195   action:     break ';'
196               {
197                   if (loopl) {
198                   brks[loopl] = merge (brks[loopl],
199                   makelist (nextquad));
```

```
200                 gen (_GOTO, 0, 0);
201                 $$ = NULL;
202                 } else {
203                 yyerror ("Break fuera de lugar");
204                 YYERROR;
205                 }
206             }
```

El statment *break*, (renglón 195) primeramente comprueba que esté dentro de un loop, (renglón 197), y lo anexa a la lista de breaks pendientes (Renglones 198, 199), genera su código *goto* 200 y regresa un valor *NULL*. Si se está dentro del contexto de un loop, marca el error correspondiente.

```
207   |   CONTINUE ';'
208   {
209       if (loopl) {
210       cont[loopl] = merge (cont[loopl],
211       makelist (nextquad));
212       gen (_GOTO, 0, 0);
213       $$ = NULL;
214       } else {
215       yyerror ("Continue fuera de lugar");
216       YYERROR;
217       }
218   }
```

El statment *continue* se genera en forma análoga del *break*.

```
219   |   RETURN ';'
220   {
221       gen (_RET, 0, 0);
222       $$ = NULL;
223   }
224   |   RETURN expresión ';'
225   {
226       gen (_RETV, $2.Beg, $2.End);
227       $$ = NULL;
228   }
229   ;
```

El statement *return* sin expresión genera un código espacial *RET* (renglón 221), y el que regresa una expresión genera otro código *RETV* el cual contiene la dirección de la expresión que debe regresar, (renglón 226).

```
230  null:          ';'
231             {
232                             $$ = NULL;
233             }
234             ;
```

El statement ';' vacío no genera código y simplemente regresa un *NULL*.

```
235  oexp:
236          {
237              $$.Beg = NULL;
238              $$.End = NULL;
239          }
240          |   expresión
241          {
242              $$ = 1;
243          }
244          ;
```

La regla *oexp* (renglón 235), cuando es vacía regresa una dirección de expresión con *NULL*. Si hay expresión (renglón 240), regresa su dirección.

3.5 Funciones de la biblioteca de C.

Todas aquellas funciones de la biblioteca estándard de entrada y salida son reconocidas por el analizador lexicográfico y regresan un *token* especial denominado *BLTIN*.

En la gramática se contempló por separado la llamada de una función normal (definida por el usuario) y una de la biblioteca, estas últimas son ejecutadas en código de máquina y las primeras son interpretadas del código intermedio.

CONCLUSIONES

Uno de los objetivos de este proyecto fue el de diseñar e implantar un prototipo de *APC*. Dicho prototipo debería probar la validez de las aportaciones originales.

APC mostró que algunas de las proposiciones son correctas, como por ejemplo: El método de evaluar las expresiones directamente del texto facilitó bastante algunos aspectos del depurador, pero hace que sea lenta la interpretación. Una posible alternativa, si se quiere seguir con esta idea, es tener un cache de expresiones ya precompiladas, para que sólo la primera vez que se evalúe una expresión, se efectúe el proceso de compilación y en las siguientes se use la expresión del cache.

Otra alternativa sería desechar la idea y tener cada expresión en código intermedio, ésto agilizaría la interpretación y complicaría la depuración.

Por otro lado, en su versión inicial no se consideraron todos los tipos básicos de *C*, como estructuras, uniones y flotantes; sin embargo, *APC* maneja hasta seis modificadores mismos ya que es capaz de manejar el *pcc* [JOHN 79] que es el compilador oficial de *UNIX*. En lo relacionado al esquema elegido para manejar los tipos, ha resultado eficiente y práctico. En una versión futura se piensan incluir todos los tipos básicos.

Para lograr una mayor eficiencia se ha considerado la generación de un analizador lexicográfico más eficiente.

Un resultado inesperado en este proyecto fue el haber generado una tesis de licenciatura, que usó varias de las funciones básicas del compilador, así como la misma gramática.

Otra de las limitaciones que se impusieron a este proyecto fue la de no incluir las instrucciones para el preprocesador de *C*, que consideramos sería de bastante utilidad incluir en una versión futura.

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

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 Feiler 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, n\'um. 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, n\'um. 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. n\'um. 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, n\'um. 3. septiembre 1982.

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

RUN/C Age of Reason Co.

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, n\'um. 9, 563-573, septiembre 1981.

VERM83 Vermaak D., PROGEN: A Programming Environment, SIGPLAN Notices, vol. 18, n\'um. 3, marzo 1983.

Gramática de C

```
c_pro       :   /* */
            |   M00 sc Ddec ';' c_pro
            |   M00 fun_def c_pro
            |   usr_exp
            ;
/*
*               definición de funciones
*/              ,
fun_def     :   sc Tspec FdecR Mfu00 par_dec Mfu01 com-
                pound_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         :   /* */
                |   sc Tspec par_def ';' par_dec
                ;
par_def         :   DdecR
                |   DdecR ',' par_def
                ;
/*
*                   declaraciones
*/
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
*/
statL       :   stat
            |   statL M03 stat
            ;
stat        :   compound_stat
            |   switch_stat
            |   cond_stat
            |   loop
            |   action
            |   null
            |   expresion ';'
            ;
compound_stat : '{' internalDdec M01 statL M02 '}'
            ;
M01         :   /* */
            ;
M02         :   /* */
            ;
M03         :   /* */
            ;
N00         :   /* */
            ;
switch_stat :   SWITCH M08 '(' expresion ')' '{' caseL M03 de-
                fault '}'
            ;
caseL       :   case_stat
            |   caseL M03 case_stat
            ;
case_stat   :   CASE M09 CONSTANT ':' statL
```

```
                ;
default         :   /* empty */
                |   DEFAULT M10 ':' statL
                ;
M08             :   /* */
                ;
M09             :   /* */
                ;
M10             :   /* */
                ;
cond_stat       :   IF M04 '(' expresion ')' stat
                |   IF M04 '(' expresion ')' stat N00 ELSE M03 stat
                ;
M04             :   ;
loop            :   WHILE M05 '(' expresion ')' M03 stat
                |   DO M06 stat M03 WHILE '(' expresion ')' ';'
                |   FOR M07 '(' oexp ';' oexp ';'  oexp ')' stat
                ;
M05             :   /* */
                ;
M06             :   /* */
                ;
M07             :   /* */
                ;
action          :   BREAK ';'
                |   CONTINUE ';'
                |   GOTO ID ';'
                |   LABEL stat
                |   RETURN ';'
                |   RETURN expresion ';'
                ;
null            :   ';'
                ;
```

```
oexp        :   /* */
            |   expresion
            ;
/*
*               expresiones
*/
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
            |   M_RETV expjr ETX
            |   M_GOTO CONSTANT
            |   M_CASE CONSTANT
            |   M_RET
            |   M_DEF CONSTANT
            ;
gen_expL    :   gen_exp
            |   gen_expL gen_exp
            ;
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 exp
                |   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
                ;
/*
*               tipos
*/
Tspec           :   T
```

```
                ;
T               :   INT
                |   CHAR
                ;
sc              :   /* empty */
                |   STATIC
                ;
```