UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

COLEGIO DE CIENCIAS Y HUMANIDADES

UNIDAD ACADEMICA DE LOS CICLOS

PROFESIONAL Y POSGRADO

INSTITUTO DE INVESTIGACIONES

EN MATEMATICAS APLICADAS

Y EN SISTEMAS

EDITOR ORIENTADO A LOS LENGUAJES PARA MICROCOMPUTADORAS

TESIS QUE PARA OBTENER EL GRADO DE

M A E S T R O

EN CIENCIAS DE LA COMPUTACION

PRESENTA

CRISTOBAL JUAREZ CASTELLANOS

México,D.F. 1985

AGRADECIMIENTOS

Agradezco ampliamente el apoyo y asesoría invaluable proporcionada por mi asesor Dr. Raúl Medina Mora, sin la cual no hubiera sido posible éste trabajo. También agradezco el apoyo prestado para la elaboración de ésta tesis a mis compañeros del Departamento de Computación, en especial al personal de la FOONLY (Gilberto Becerril, Rodrigo Siguenza, Carlos Palomino y Ricardo Espriella), así como a las secretarias Elvia Velázquez y Alma Rosa Rodríguez. Agradezco especialmente a mis compañeros Pedro Rafael Márquez, Luz Marina Quiroga y Francisco Nava, por la revisión de la tesis, así como por sus críticas y opiniones. Finalmente quiero agradecer al Dr. Alejandro Buchmann por el apoyo que me ha brindado, así como por las opiniones y críticas a este trabajo.

Tabla de Contenido

Lista de Figuras

Lista de Tablas

CAPITULO 1

Introducción

1.1 Identificación del Problema.

Uno de los principales objetivos de la ingeniería de software es aumentar la productividad así como la calidad de la programación, para lo cual se diseñan e implantan herramientas que auxilien a los programadores en el desarrollo de sus programas. En el proceso de desarrollo de un programa podemos identificar 3 fases: edición, compilación y depuración. En un medio ambiente de programación tradicional cada una de estas fases se lleva a cabo como un proceso independiente, utilizando herramientas desasociadas.

Dado que los procesos de edición y compilación se encuentran separados y que el compilador realiza el proceso de

análisis sintáctico de programas completos, los errores sintácticos cometidos por el programador al momento de editar su programa solo son conocidos hasta el momento de la compilación. Este proceso es sumamente lento, además de que distrae al programador de la escencia del problema a resolver al obligarlo a pensar en términos del lenguaje en lugar de pensar en términos del problema a resolver, disminuyendo la productividad del programador.

En los últimos años se han realizado grandes esfuerzos para crear herramientas de software que permitan resolver este problema, tales como el ALOE[1] [MEDINA-MORA 81], el sistema SYNTHESIZER [TEITELBAUM 81] o el sistema SUPPORT [ZELKOWITZ 84]. Otro aspecto sumamente importante, es el desarrollo tecnológico en cuanto a hardware se refiere, dado lo cual, en los últimos años ha habido un gran auge de las microcomputadoras. Este auge nos enfrenta al problema de crear herramientas para el desarrollo de software para microcomputadoras.

1.2 Editores Orientados a los Lenguajes.

Un programa no es una estructura textual, un programa es una composición jerárquica de estructuras computacionales, por lo tanto, su edición, ejecución y seguimiento deben realizarse en un medio ambiente con las herramientas propias

---

[1] A Language Oriented Editor

de estas estructuras y cuyo enfoque ponga el acento en esta concepción.

Un editor orientado a los lenguajes es un editor generado a partir de una descripción gramatical, es decir, no es un editor creado exprofeso para un lenguaje en particular [MEDINA-MORA 81]; algunas de sus características mas relevantes son:

- En la descripción de la gramática se define una parte muy importante de la interfaz con el usuario. Los nombres de los operadores del lenguaje son los nombres de los comandos de construcción del ALOE.
- La edición estructurada o constructiva se realiza a todos los niveles del lenguaje.
- Capacidad de procesamiento de contexto sensitivo a través de rutinas de acción asociadas a cada construcción del lenguaje.

Un editor orientado a los lenguajes es un editor dirigido por sintaxis, es decir, es un editor que entiende la sintaxis del lenguaje para el cual se generó. Con lo cual se garantiza en todo momento que el programa que se está editando es sintácticamente correcto; aumentado de esta manera la productividad del programador.

1.3 Objetivo.

El objetivo de este trabajo es desarrollar e implantar un EDitor Orientado a los LEnguajes para Microcomputadoras (EDOLEM) o editor estructurado para microcomputadoras, así como investigar las características y limitaciones de este sistema implantado en una

microcomputadora.

1.4 Trabajos Relacionados.

La creación de herramientas para el desarrollo de software ha sido una constante en la historia de la computación, sin embargo, tradicionalmente estas herramientas se encuentran separadas. El objetivo del desarrollo de herramientas de software, es la realización, mantenimiento y depuración de programas en forma mas flexible y eficiente. Es de este objetivo de donde surgió la idea de conjuntar y desarrollar en un sistema integrado las herramientas tradicionales de software. Algunos de los principales trabajos en este campo se presentan a continuación:

- El sistema EMILY [HANSEN 71] fue el primer esfuerzo para crea un editor dirigido por sintaxis. Este sistema está dirigido por menú, en el cual el usuario construye un programa a partir de seleccionar una producción BNF [BACKUS 59]. En las producciones BNF se incluye la representación concreta del lenguaje. Este sistema no es un sistema integrado, ya que produce como salida un archivo de texto para su compilación por separado.

- El sistema SYNTHESIZER [TEITELBAUM 81] se desarrolló originalmente para el lenguaje PL/CS, que es un subconjunto de PL/I. Este es un sistema híbrido que al nivel de las proposiciones manipula los programas en forma estructurada, mientras que al nivel de las expresiones la manipulación se realiza en forma textual; para lo cual cuenta con dos tipos de elementos: formas y frases; donde una forma es un patrón de caracteres y signos de puntuación predefinidos, y una frase es una secuencia arbitraria de símbolos.

- El sistema SUPPORT [ZELKOWITZ 84] es un editor dirigido por sintaxis que intenta evitarle al programador el pensar en términos de notación posfija, para lo cual implanta un sistema de análisis y construcción híbrido. Este sistema cuenta

con un analizador sintáctico que recibe como entrada el nodo no-terminal actual y el texto a analizar y regresa el árbol que representa al segmento de programa analizado. En este sistema el usuario puede cambiar el modo de operación, pasando del modo de construcción de árbol a modo texto y viceversa. Este sistema proporciona un medio ambiente integrado para el desarrollo y ejecución de programas en Pascal [JENSEN 75].

- El sistema SYNED [HORGAN 84] es un editor basado en el lenguaje. Este sistema está basado en dos conceptos: comandos interpretables y transacciones, a partir de las cuales manipula una relación bidireccional entre el árbol sintáctico y la representación textual. SYNED es un editor para el lenguaje de programación C, aunque plantea la generación de editores para lenguajes tipo ALGOL utilizando el generador de analizadores sintácticos YACC.

- El ALOE [MEDINA-MORA 81] es un editor orientado a los lenguajes. Este es un sistema generatriz dirigido por tablas. En este sistema se puede generar un editor para un lenguaje a partir de una descripción de la gramática del lenguaje. En el ALOE la interacción del usuario con el sistema es en términos de la estructura del lenguaje y no en términos textuales.

Cada uno de los sistemas mencionados anteriormente tienen sus ventajas y desventajas, sin embargo, para realizar una evaluación objetiva se tendría que contar con estadísticas del tiempo utilizado para el desarrollo de un programa con cada uno de estos sistemas. Dado que no se cuenta con estadísticas, analizaremos las diferencia de enfoque de estos sistemas. Existen dos características básicas que diferencian a estos sistemas:

1. Es un sistema generatriz o no.
2. La manipulación es estructurada o es híbrida.

Se entiende como sistema generatriz, al sistema que puede generar un editor para un lenguaje determinado a partir de una descripción de la gramática, sin que tenga que rescribirse el sistema. De los sistemas mencionados anteriormente, solamente el ALOE [MEDINA-MORA 81] y SYNTHESIZER [TEITELBAUM 81] son sistemas generatrices. Aunque el sistema SYNED [HORGAN 84] plantea la posibilidad de generación de editores utilizando el generador de analizadores sintácticos YACC, este sistema no es en el sentido estricto de la palabra un sistema generatriz.

En un sistema híbrido al nivel de las proposiciones la edición se lleva a cabo en forma estructurada mientras que al nivel de las expresiones se realiza en forma textual, esta forma de operación pretende evitarle al usuario pensar al nivel de las expresiones en notación polaca. En un sistema estructurado la aplicación de los comandos de construcción pone el énfasis en la estructura del programa que se está editando, es decir, se tiene una edición estructurada a todos los niveles, incluyendo las expresiones. Esta característica de ninguna forma es una desventaja, ya que si bien es cierto que al nivel de las expresiones el usuario tiene que pensar en notación posfija, también es cierto que esto permitiría al usuario pensar en su programa en forma estructurada.

Dado que existe una gran diversidad de lenguajes de programación y que estos tienen un desarrollo natural, el poder generar un editor en forma fácil y flexible para

cualquier lenguaje o para otra versión del mismo lenguaje es un aspecto sumamente importante, ya que esto ahorra tiempo en la implantación de un editor para un lenguaje o en las modificaciones al existente. En cuanto a la facilidad de editar programas en forma textual al nivel de las expresiones (asumiendo esta como la forma natural) en lugar de editar en forma estructurada, es un aspecto que está todavía en proceso de experimentación, además de que técnicamente no es un problema difícil el asignar un pequeño analizador sintáctico de expresiones. Bajo estas consideraciones, en el trabajo aquí desarrollado se decidió utilizar un esquema generatriz con edición estructurada a todos los niveles.

1.5 Organización de la Tesis.

Un objetivo de este trabajo de tesis es mostrar las características y limitaciones de la implantación de un EDitor Orientado a los Lenguajes para Microcomputadoras, llamado EDOLEM; pero el objetivo principal es mostrar las motivaciones que llevaron a tomar ciertas decisiones. En otras palabras, el objetivo de esta presentación es transmitir la experiencia (con sus aciertos y errores) obtenida a lo largo de la implantación de este sistema.

Bajo la perspectiva anterior, en el capítulo 2 se presentan las características de la interfaz con el usuario, en el capítulo 3 se describe el proceso de generación de un editor así como las características y el formato de descripción de la gramática de un lenguaje específico para su

procesamiento, en el capítulo 4 se proporciona una descripción detallada del manejo de memoria y un análisis de cuales fueron los motivos que llevaron a utilizar un esquema de manejo de memoria y no otro, el capítulo 5 describe la representación interna del árbol sintáctico que manipula el EDOLEM, y en el capítulo 6 se hace una evaluación de los resultados obtenidos en la implantación de este sistema. En el apéndice A se proporciona una descripción detallada de los comandos de edición, los cuales son comunes a todos los EDOLEM; en el apéndice B se describen los comandos del esquema de despliegue; en el apéndice C se describe detalladamente el proceso de descripción de la gramática de un lenguaje así como la generación de un editor para dicho lenguaje; finalmente, en el apéndice D se muestra la descripción de la gramática del lenguaje de programación Pascal [JENSEN 75].

CAPITULO 2

Interfaz con el Usuario

En un sistema interactivo uno de los módulos de mayor importancia es el módulo de interfaz o comunicación entre el sistema y el usuario, gran parte de la aceptación y utilización de un sistema se decide en las facilidades que éste proporcione al usuario, por lo tanto, un buen sistema debe contar con una interfaz flexible y de fácil uso. Algunos de los objetivos más importantes de la interfaz con el usuario de un ALOE [MEDINA-MORA 81] son:

- Permitir que la interacción del usuario con el editor sea en términos de las estructuras de sus programas.
- Minimización de esfuerzos.- El usuario debe emplear el mínimo de esfuerzo para indicar la acción que desea realizar.

- Flexibilidad.- La interfaz debe ser lo suficientemente flexible como para satisfacer las necesidades de usuarios nóveles y usuarios expertos.

- Actualización de despliegue inmediata.- En un medio ambiente interactivo el usuario debe conocer el estado actual del sistema, esto se lleva a cabo actualizando el despliegue después de cada interacción.

- Retardo mínimo.- El tiempo de retardo en un medio ambiente interactivo debe ser pequeño, de forma tal que el tiempo de respuesta sea adecuado. El tiempo de retardo es aceptable sí el tiempo máximo de respuesta varía entre uno y tres segundos.

2.1 Edición Dirigida por Sintaxis.

Un editor orientado a los lenguajes para microcomputadoras (EDOLEM), es un editor dirigido por sintaxis, es decir, es un editor que entiende la sintaxis del lenguaje para el cual se generó. Este es un sistema de construcción de estructuras de árbol, que llamaremos árbol sintáctico. El usuario construye un programa insertando formas que representan las construcciones del lenguaje, donde cada forma corresponde a un nodo de cierto tipo en el árbol. Es a través de estas formas que el EDOLEM conoce en cada punto del programa (nodo del árbol sintáctico) cuales operadores son sintácticamente correctos, permitiéndole al usuario la aplicación de un operador solo cuando éste sea válido.

En el EDOLEM se cuenta con tres grandes tipos de nodos: nodos terminales, que corresponden a los operadores terminales del lenguaje(variables, constantes, etc.); nodos no-terminales que corresponden a los operadores no-terminales del lenguaje(operadores aritméticos, operadores condicionales,

etc.) y los cuales pueden ser de orden[2] fijo, con un número fijo de descendientes o de orden variable, es decir con una lista de descendientes; y los meta-nodos[3], que son nodos intermedios no expandidos que representan al conjunto de descendientes cuya aplicación es sintácticamente correcta.

Como se muestra posteriormente en el capítulo 3, en la descripción de la gramática se define gran parte de la interfaz con el usuario a través de definir el esquema de despliegue de cada operador del lenguaje; es decir, la representación concreta del lenguaje es manipulada por el EDOLEM. Esto es; los signos de puntuación, separadores, formateo del programa, etc, se realizan en forma automática, permitiéndole al usuario interactuar con el editor en términos de la estructura del lenguaje y evitando de esta forma los errores sintácticos, tipográficos, de puntuación, etc.

Un aspecto muy importante en la interfaz con el usuario, es el despliegue del estado actual del desarrollo del programa y/o la posición en el árbol sintáctico en donde se encuentra posicionado el programador, a este fin se utiliza el cursor. En algunos sistemas, como en Cornell [TEITELBAUM 81]

---

[2] ORDEN.- El orden de un nodo está dado por el número de descendientes que tiene(ej. un nodo de orden tres tiene tres descendientes)

[3] Se denomina meta-nodo a un nodo que se remplaza por medio de un comando de construcción.

se utiliza un cursor de un solo caracter, como se muestra en la figura 2-1.a. Sin embargo, en algunos casos un cursor de este tipo puede ser ambiguo. Otros sistemas, como el ALOE [MEDINA-MORA 81], utilizan un cursor de área, es decir, la pantalla se pone en inversa en todo el texto que representa al nodo donde está posicionado el programador, figura 2-1.b; este tipo de cursor proporciona una visión más clara durante el proceso de construcción y edición de un programa, por lo cual en el EDOLEM se utiliza un cursor de área.

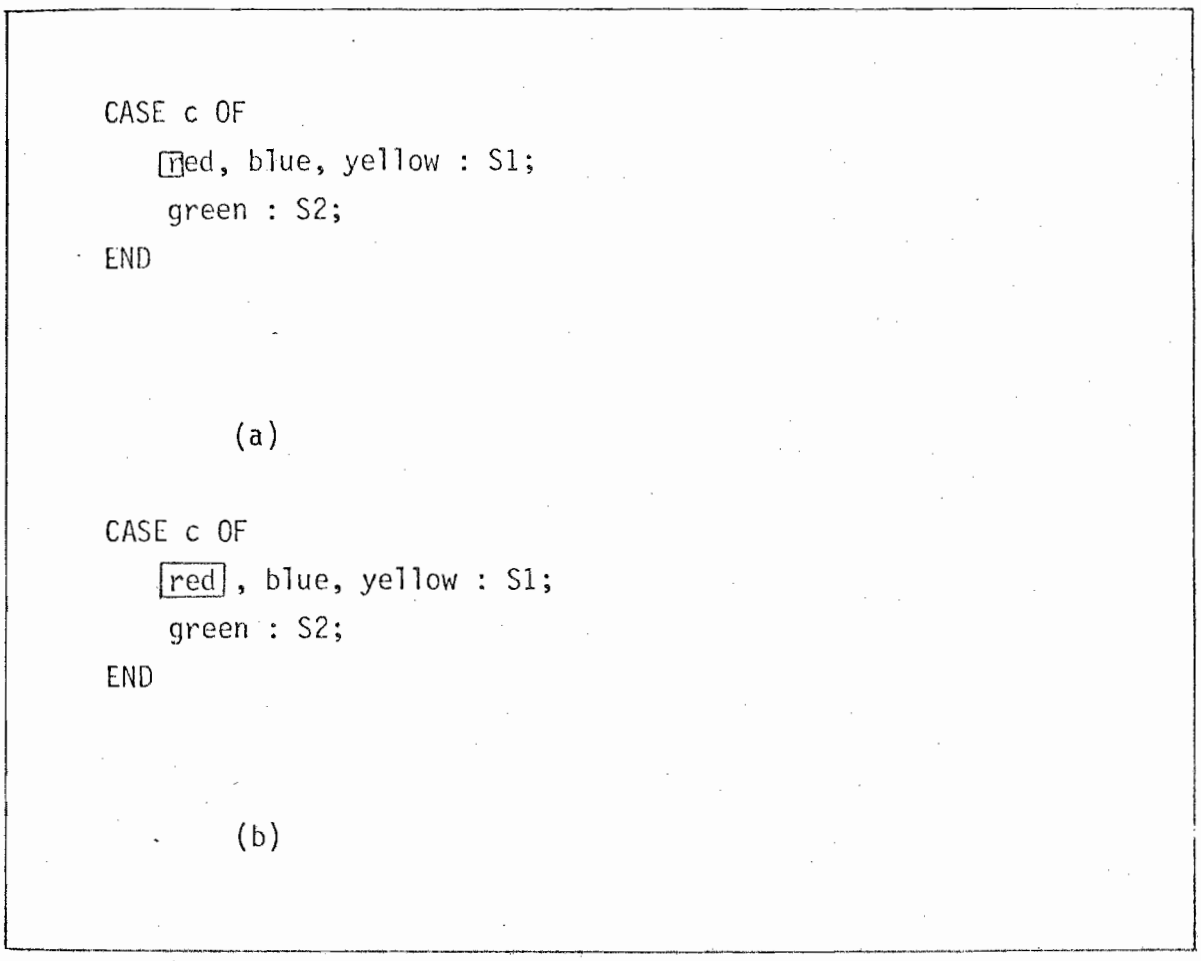


Figura 2-1: Tipos de cursor.- a) de caracter, b) de área.

En la figura 2-2 se muestra un ejemplo de las características de la interfaz de un EDOLEM descritas anteriormente; por ejemplo, en la figura 2-2.a nos encontramos en el meta-nodo "statement", representado como "[<statement>]", donde se indica un meta-nodo encerrando el nombre entre los símbolos "<" y ">"; el meta-nodo "statement" indica que en este nodo se debe aplicar un operador que sea un "statement". En la figura 2-2.b se muestra la aplicación del operador CASE, donde la representación concreta de esta proposición condicional la manipula el EDOLEM a través de los esquemas de despliegue, es decir, las palabras CASE, OF y END, el signo de puntuación ":", así como el formateo (la indentación) se despliegan automáticamente, los meta-nodos <expresion>, <lista de etiquetas> y <statement> son las formas que representan al conjunto legal de operadores que se pueden aplicar a cada uno de los tres descendientes de este operador no-terminal.

2.2 Comandos de un Edolem.

Cada EDOLEM está constituido por dos tipos de comandos: comandos de construcción (comandos del lenguaje) y comandos de edición. Los comandos de construcción son comandos que dependen del lenguaje para el cual se generó el EDOLEM, los comandos de edición son comandos independientes del lenguaje, por lo cual son comunes a todos los EDOLEM.

La forma de aplicación de los comandos de construcción y los comandos de edición es diferente, como se

```
BEGIN .
    <statement>
END

     (a)

BEGIN
    CASE <expression> OF
        <lista de etiquetas> : <statement>
END

     (b)
```

Figura 2-2: Ejemplo de las características de la interfaz.

muestra más adelante, sin embargo, en ambos casos el usuario solamente requiere aplicar la cadena de caracteres mínima tal que el comando no sea ambiguo.

2.2.1 Comandos de Construcción.

Como se mencionó previamente, los comandos de construcción se definen en la descripción de la gramática del lenguaje, esto es, los nombres de los comandos de construcción son los nombres de los operadores del lenguaje. Durante el proceso de construcción después de la aplicación de un comando, el cursor se mueve del nodo actual al siguiente meta-nodo en el árbol (definido en preorden), por ejemplo, en la figura 2-3 se describe una secuencia de la aplicación de los comandos de construcción utilizando la gramática del

lenguaje de programación Pascal [JENSEN 75], en la primera columna se muestra el comando que proporciona el usuario, en la segunda columna el despliegue y en la tercera la construcción del árbol sintáctico interno.

Como puede apreciarse en la figura 2-3, la aplicación de los comandos de construcción se lleva a cabo escribiendo el nombre del comando que se desea aplicar; para algunos comandos esta forma de aplicación es adecuada (tales como un IF, WHILE, FOR, etc) sin embargo, para otros comandos, tales como MULTIPLICA, SUMA, etc., es más claro (además de práctico) aplicar el símbolo que represente la operación deseada, por ejemplo "*", "+", etc. Esta facilidad se obtiene a través de permitir la definición de sinónimos en la descripción de la gramática, por ejemplo, en la figura 2-3.b el usuario aplica ">" que es el sinónimo del comando MAYOR o en la figura 2-3.e aplica ":=" en lugar del comando ASIGNA.

Dado que uno de los objetivos de la interfaz con el usuario es proporcionarle a éste una información exacta del estado actual del programa, después de cada interacción se actualiza el despliegue. Sin embargo, en la figura 2-3 es aparente que el usuario deberá aplicar uno por uno los comandos de construcción, actualizándose el despliegue después de la aplicación de cada comando. Esta forma de operación, que para un programador novel sería aceptable, para un programador experto no lo sería. Es aquí donde se presenta el aspecto de la flexibilidad de la interfaz. Para satisfacer a ambos

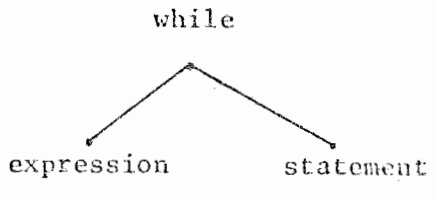
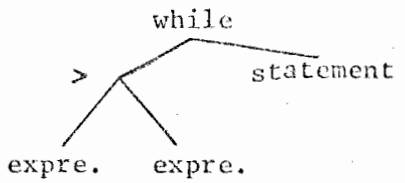
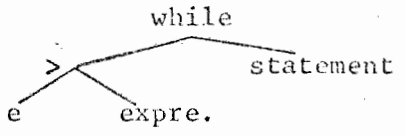
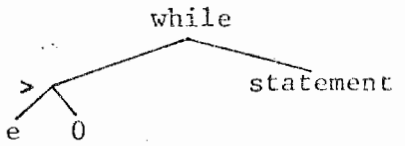
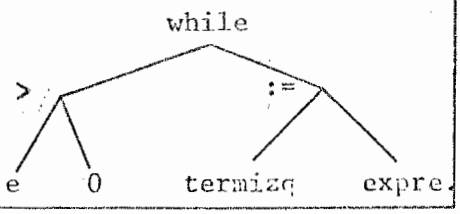
while	WHILE < expression > DO < statement >	
>	WHILE < expression > <expression> DO < statement >	
e	WHILE e > < expresion > DO < statement >	
0	WHILE e > 0 DO < statement >	
:=	WHILE e > 0 DO < termizq > := <expression> ;	

Figura 2-3: Ejemplo de aplicación de los comandos de construccion.

programadores, el EDOLEM le permite al usuario aplicar solamente un comando o aplicar una secuencia de comandos. En la figura 2-4 se muestra la misma secuencia de comandos del

ejemplo mostrado en la figura 2-3, pero aplicados simultáneamente.

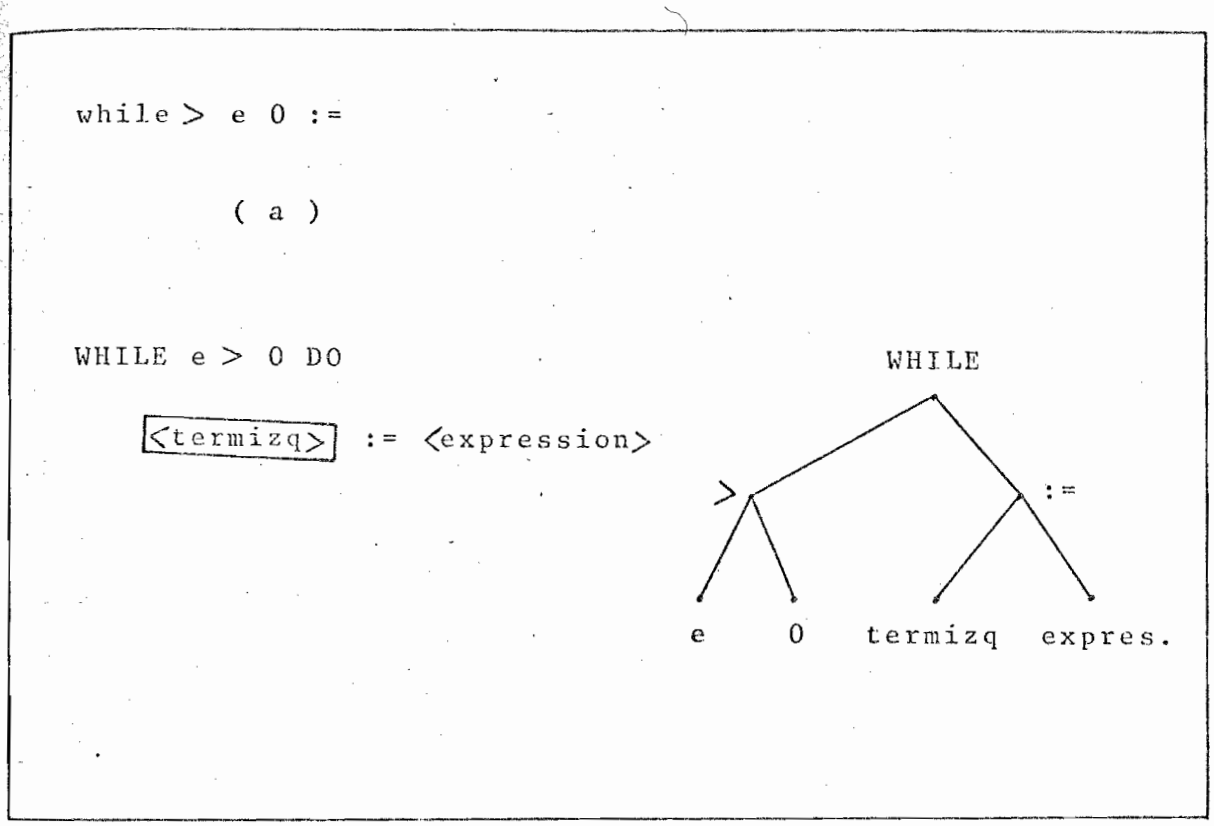


Figura 2-4: Ejemplo de aplicación de una secuencia de comandos de construcción.

2.2.2 Comandos de Edición.

Los comandos de edición son comandos independientes del lenguaje, tales como movimiento del cursor, comandos de ayuda, comandos de manejo de árbol, etc. Como se mencionó anteriormente, la forma de aplicación de estos comandos es diferente de la forma de aplicación de los comandos de construcción. Con el objeto de permitir que en la definición de la gramática se pueda utilizar un conjunto amplio de

cadenas de caracteres en la definición de los comandos de construcción, los comandos de edición se pueden aplicar por medio del nombre o por medio de caracteres de control; en el caso de utilizar el nombre, éste debe ir precedido de un punto ".", en el caso de utilizar el sinónimo éste se aplica directamente. Por ejemplo, si se quiere pedir ayuda se puede aplicar el comando ".HELP" o "^x"(donde ^x significa CONTROL-x).

Un EDOLEM cuenta con cinco tipos de comandos de edición: comandos de movimiento del cursor, comandos de búsqueda, comandos de manejo de árboles, comandos de ayuda y otros comandos. En el apéndice A se describe detalladamente cada uno de los comandos.

2.2.2.1 Comandos de Movimiento del Cursor.

Este tipo de comandos le permite al programador moverse a través de su programa (árbol sintáctico). Los movimientos del cursor no se realizan siguiendo la representación textual del programa como en los editores de texto, sino de acuerdo a la estructura del programa, es decir, el programador se mueve en el árbol sintáctico. A continuación se describen los movimientos del cursor disponibles en el EDOLEM.

↓ Mueve el cursor al primer descendiente legal del nodo actual de acuerdo con el esquema de despliegue.

↑ Mueve el cursor al padre del nodo actual.

→ Mueve el cursor al siguiente descendiente del

padre del nodo actual de acuerdo con el esquema de despliegue, si no existe otro descendiente se mueve al hermano del padre del nodo actual recursivamente.

← Mueve el cursor al hermano anterior del nodo actual de acuerdo con el esquema de despliegue, sí no existe, se mueve al hermano anterior del padre del nodo actual recursivamente.

.HOME Mueve el cursor a la raíz del árbol.

.BACK Si el comando anterior fue un movimiento del cursor, regresa este a la posición anterior, en caso contrario no tiene ningun efecto.

2.2.2.2 Comandos de Busqueda.

Otros comandos muy importantes de movimiento del cursor, son los comandos de búsqueda. En este sistema se cuenta con dos comandos de búsqueda, los cuales de describen a continuación.

.FIND Busca en el árbol una variable o una constante hacia adelante.

.RFIND Busca en el árbol una variable o una constante hacia atrás.

2.2.2.3 Comandos de Manejo de Arboles.

Los comandos de manipulación de árboles son de gran importancia, dado que a través de ellos se permite la edición de las estructuras de los programas, es decir, es a través de estos comandos que se modifica la estructura de un programa. En esta versión del EDOLEM se cuenta con un conjunto básico de este tipo de comandos. En el apéndice A se describe detalladamente cada uno de estos comandos.

.APPEND Si el nodo actual es un elemento de una lista, inserta un meta-nodo al final de la lista.

.PREPEND Si el nodo actual es un elemento de una lista, inserta un meta-nodo al inicio de la lista.

.EXTEND Extiende una lista (de la cual el nodo actual es un elemento) insertando delante del nodo actual.

.BEXTEND Extiende una lista (de la cual el nodo actual es un elemento) insertando detrás del nodo actual.

.DELETE Borra un subárbol.

2.3 Organizacion de la Pantalla.

Dado que la pantalla de despliegue con que se cuenta manipula 24 renglones por 80 columnas, es necesario organizarla en áreas de trabajo específicas. En la figura 2-5 se muestra la organización que hace el EDOLEM de la pantalla, en donde podemos apreciar que se tiene 21 renglones para el despliegue del programa (los renglones 1 a 21), un renglón (el renglón 22) para la captura de los comandos del usuario, un renglón (el renglón 23) para el "prompt" del sistema y un renglón para los mensajes. También se cuenta con un área de ayuda, la cual se sobrepone al área de despliegue del programa a partir de la columna 54.

2.4 Modos de Operación.

En el EDOLEM se cuenta con dos modos de operación: modo principiante y modo experto. El objetivo de tener estos dos modos de operación es que el sistema sea funcionalmente adecuado tanto para los programadores noveles como para los programadores expertos.

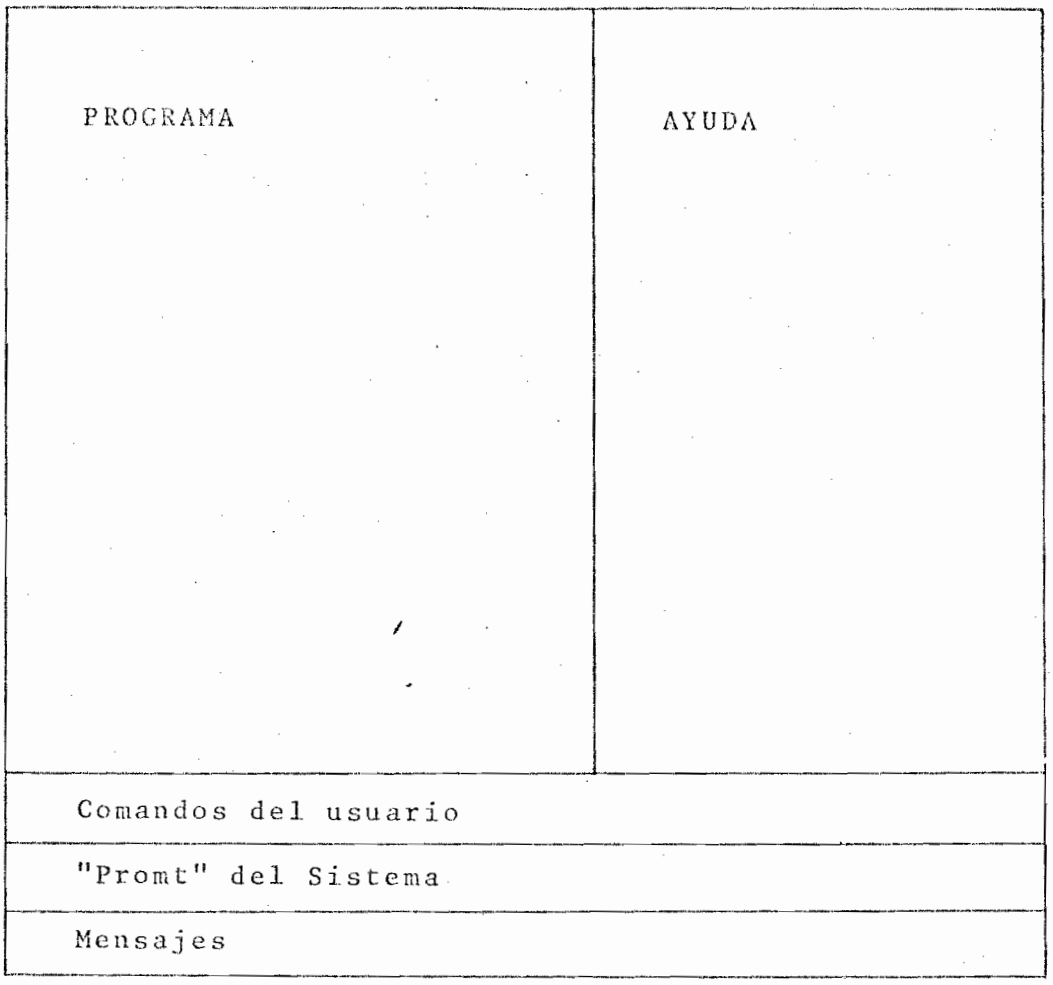


Figura 2-5: Organización de la pantalla.

En el modo de operación principiante, después de cada actualización del despliegue, en el área de ayuda descrita en la sección anterior se despliegan los comandos que puede aplicar el programador; esto es, si el nodo actual es un meta-nodo, el EDOLEM despliega los comandos de construcción

cuya aplicación es legal en dicho nodo.

En el modo de operación experto el área de ayuda solo se despliega cuando el programador solicita la ayuda explícitamente.

CAPITULO 3

Generación del Editor

3.1 Introducción.

En este capítulo se describe el proceso de generación de un editor para un lenguaje determinado, así como las características y detalles estructurales que conforman un EDOLEM.

En un EDOLEM podemos identificar dos fases: generación y edición. En la fase de generación se le describe al sistema la gramática del lenguaje, mientras que en la fase de edición se utiliza el sistema generado para crear y/o modificar programas o cualquier otra estructura. En las figuras 3-4 y 3-5 se muestra un subconjunto de la gramática de Pascal para ejemplificar la descripción de la gramática así

como el proceso de construcción.

La función de la fase de generación es precisamente obtener los comandos de construcción para la gramática proporcionada por el realizador[4], así como proporcionar las características que permitan una edición sintácticamente correcta. La descripción de la gramática para un EDOLEM está constituída por tres tipos de primitivas: los operadores terminales, no-terminales y clases. Estos operadores forman una estructura de dos niveles: En el primer nivel se definen los operadores del lenguaje (terminales y no-terminales), mientras que en el segundo nivel se definen las clases de operadores del lenguaje. Los operadores terminales forman las hojas del árbol sintáctico (ej. una constante, una variable), es decir, nodos que no tienen descendientes. Los operadores no-terminales forman un conjunto de nodos, cuyo orden puede ser fijo o variable. Cuando se aplica un operador no-terminal, al crearse éste se crean los meta-nodos correpondientes a cada uno de sus descendientes. El segundo nivel corresponde a las clases; la función de éstas es definir el conjunto legal de operadores (terminales o no-terminales) que pueden remplazar un meta-nodo.

El EDOLEM manipula estructuras de árbol, que

---

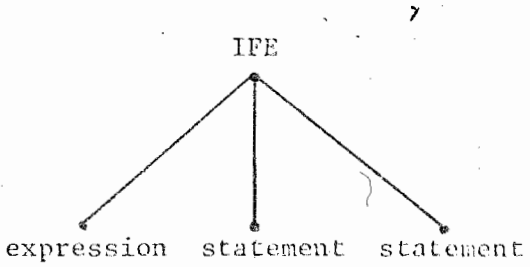
[4] Denominaremos realizador al que genera un editor para un lenguaje determinado a través de describir la gramática

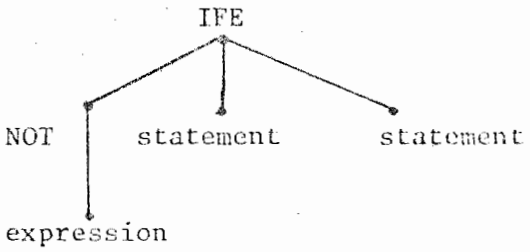
denominamos como árbol sintáctico; éste se construye a través de los comandos proporcionados por el usuario y de acuerdo a la definición de la gramática del lenguaje para el cual se generó el editor; por ejemplo, utilizando el subconjunto de la gramática de Pascal mostrado en las figuras 3-4 y 3-5, aplicamos el operador no-terminal IF-ELSE (denotado como IFE), este nodo es de orden tres; es decir, tiene tres descendientes, denotados como "expression", "statement" y "statement". Estos descendientes son meta-nodos que nos indican la clase de comandos que se pueden aplicar en dichos meta-nodos. Una vez aplicado el comando IFE, figura 3-1.a el cursor del sistema se posiciona en el primer descendiente, en el cual aplicamos el comando NOT, como se muestra en la figura 3-1.b, el cual es válido dado que se encuentra dentro de la clase "expression", como puede verse, este operador tiene a su vez un descendiente, el cual debe pertenecer a la clase "expression" y en el cual aplicamos finalmente el identificador "error", como se muestra en la figura 3-1.c.

3.2 Esquemas de Despliegue.

El EDOLEM representa internamente las construcciones del lenguaje en forma de árbol (tanto en la manipulación como en el almacenamiento), por lo cual, el programador posiciona el cursor, inserta y borra subárboles (que pueden estar constituídos únicamente por un nodo); sin embargo, generalmente el usuario está acostumbrado a interactuar en términos textuales y no de estructuras arboresentes. En este

Comando	Despliegue	Construcción Interna

IFE	IF < expression > THEN
	< statement >
	ELSE
	< statement >
		

NOT	IF NOT < expression > THEN
	< statement >
	ELSE
	< statement >
		

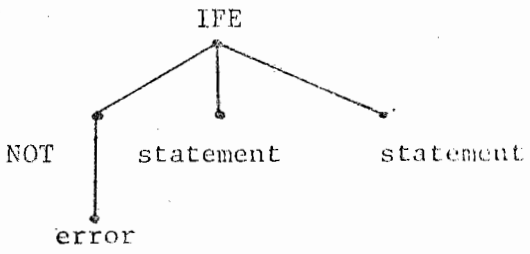
ERROR	IF NOT ERROR THEN
	< statement >
	ELSE
	< statement >
		

Figura 3-1: Ejemplo de construcción de un programa.

sentido es necesario contar con una interfaz que transforme de la representación interna a una representación textual para que el usuario pueda leer su programa. Esta interfaz se proporciona a través de un conjunto de esquemas de despliegue definidos en la descripción del lenguaje.

Todos los operadores deben tener asociado un esquema de despliegue, el cual proporciona la descripción del texto a utilizar, el formato de despliegue y la forma en que se deben

desplegar los descendientes (para nodos no-terminales) del nodo. Las cadenas de caracteres proporcionadas en el esquema de despliegue y que no están precedidas por una "@" se despliegan literalmente, solo las secuencias que empiezan con "@" se tratan de manera diferente. La forma en que se despliegan los operadores terminales y no-terminales es diferente. En el apéndice B se proporciona una lista completa y detallada de los comandos de formateo. Algunos de los comandos de formateo son:

@n Inserta una nueva línea en el despliegue.

@+ Incrementa el nivel de indentación (tiene efecto en el siguiente "@n").

@- Decrementa el nivel de indentación (tiene efecto en el siguiente cambio de línea-"@n"-).

para los operadores terminales los comandos de despliegue son:

@c Despliega el valor de una constante o una cadena.

@s Despliega el nombre de una variable, tomándolo de la tabla de símbolos.

Para los operadores No terminales los comandos de despliegue son:

@<n> Despliega el n-ésimo descendiente recursivamente. Este comando es aplicable solamente para operadores de orden fijo, donde los descendientes se numeran de uno en adelante.

<pr>@O<t>[@q<po>][@e<s>]
Recorre los operadores lista para su despliegue. La "@O" indica que cada elemento de la lista se recorre en orden, la cadena "<t>" se utiliza para separar la lista de

elementos, esta se termina al encontrar "@q" o el fin del esquema de despliegue.

En la figura 3-2.a se muestra el despliegue del operador no-terminal de orden 3 IFELSE, cuyo esquema de despliegue es "IF @1 THEN@+@n@2@-@nELSE@+@n@3@-", donde las palabras IF, THEN y ELSE se despliegan literalmente; "@1", "@2" y "@3" indican el primer, segundo y tercer descendiente respectivamente, "@+" incrementa el nivel de indentación, "@-" decrementa el nivel de indentación y "@n" inserta un cambio de línea en el despliegue. En la figura 3-2.b se muestra el despliegue del operador no-terminal de orden variable VARDECLS, cuyo esquema de despliegue es "VAR@+@n@O;@n@q@-", donde la palabra VAR se despliega literalmente; "@+", "@-", y "@n" tienen el mismo significado que en el caso anterior; "@O" indica el despliegue de elementos de la lista en orden separados por ";" y "@q" indica el fin del despliegue de la lista.

3.3 Descripción de la Gramática.

Dado que el EDOLEM es un sistema orientado a los lenguajes o estructuras, para generar un editor es necesario describir la gramática del lenguaje o estructura para la cual se quiere hacer la generación. Chomsky clasificó los lenguajes en cuatro tipos: lenguajes numerables recursivamente (tipo 0), lenguajes de contexto sensitivo (tipo 1), lenguajes libres de contexto (tipo 2) y lenguajes regulares (tipo 3) [CHOMSKY 59]. De éstos el tipo de lenguaje más utilizado como un modelo para

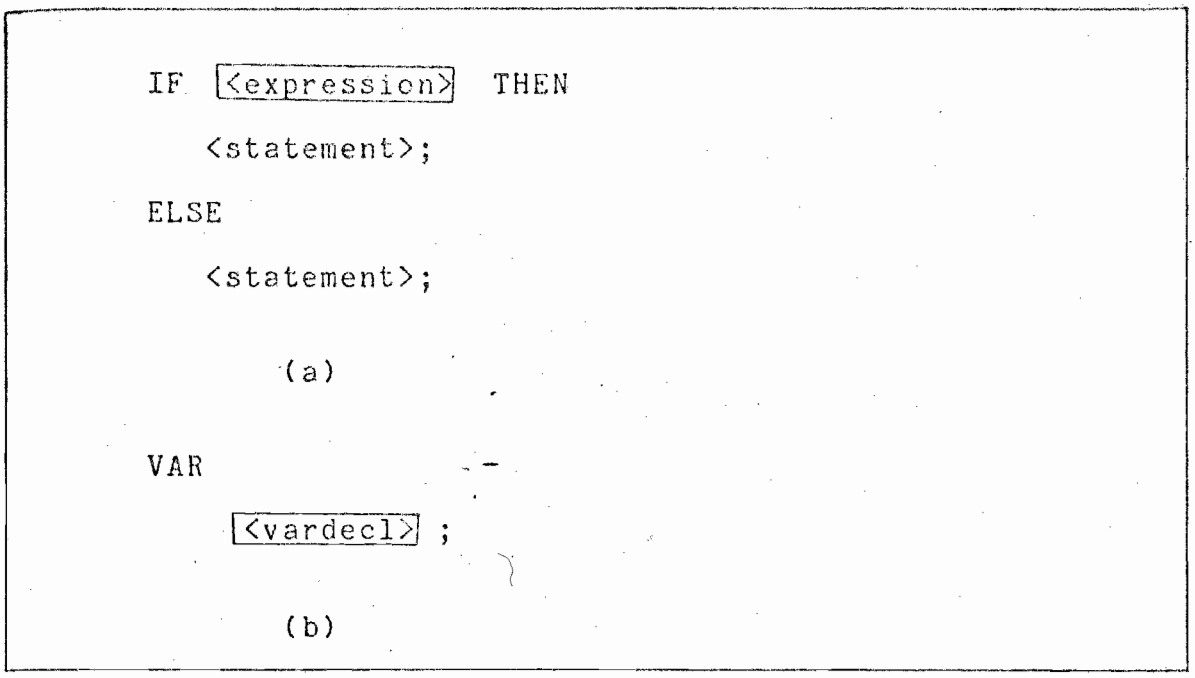


Figura 3-2: Ejemplo de los esquemas de despliegue.

los lenguajes de programación es el lenguaje libre de contexto. El formalismo BNF es una notación para gramáticas libres de contexto [NICHOLLS 75].

Algunas de las características más importantes de la descripción de la gramática en los ALOES [MEDINA-MORA 81] son:

- La edición estructurada o constructiva se realiza a todos los niveles del lenguaje.
- Capacidad de procesamiento de contexto sensitivo a través de rutinas de acción asociadas a cada construcción del lenguaje.
- En la descripción de la gramática se define una parte muy importante de la interfaz con el usuario. Los nombres de los operadores del lenguaje son los nombres de los comandos de construcción del ALOE.

Dado que el formalismo BNF es una estructura jerárquica, la utilización de éste en la descripción de

gramáticas impacta fuertemente la interfaz con el usuario, obligándolo a aplicar los operadores intermedios, tales como: "factor", "term", etc. En el ALOE [MEDINA-MORA 81] se propone un formalismo diferente, el cual tiene una estructura plana, es decir, todos los operadores de una clase se encuentran en el mismo nivel de jerarquía. En el EDOLEM se utiliza esta última forma de definición de la gramática.

3.3.1 El Nombre del Lenguaje.

Para una gramática determinada, el procesador de gramáticas genera un archivo de tablas del lenguaje. La importancia del nombre del lenguaje radica en dos aspectos fundamentales: el primero es que a través de él se le indica al EDOLEM de donde debe obtener los operadores del lenguaje y sus características, el segundo es que a través de él se evita que un programa editado con un EDOLEM generado para un lenguaje determinado y una versión determinada de dicho lenguaje, se intente editar y/o modificar utilizando un EDOLEM generado para otro lenguaje o para otra versión del mismo lenguaje. Esto implica que dos o más versiones del mismo lenguaje no deberán tener el mismo nombre, entendiéndose que tienen el mismo nombre si el nombre y extensión de los archivos son iguales.

Es importante señalar que el realizador de un EDOLEM para un lenguaje deberá cambiar el nombre del lenguaje cuando se genere un EDOLEM para otra versión del lenguaje, dado que si se intenta editar un archivo editado con un EDOLEM generado

para otra versión se tendrán inconsistencias.

Existen ciertas modificaciones a la descripción de una gramática que no producen inconsistencias, tales como cambio en el sinónimo de un operador o cambios en el esquema de despliegue, sin embargo la adición o borrado de un operador producirá cambios en la tabla de operadores por lo que estas modificaciones si producen inconsistencias.

3.3.2 Operadores Terminales.

La segunda parte de la descripción de la gramática la constituye la descripción de los operadores terminales, esta descripción, consta del nombre del operador, tipo de operador, rutinas de acción, sinónimo y el esquema de despliegue. Por ejemplo, de la figura 3-4 tenemos el operador terminal IDENT que es del tipo variable, no tiene sinónimo y su esquema de despliegue es "@s" cuya función es desplegar el nombre de una variable, tomándolo de la tabla de símbolos.

Los operadores terminales pueden clasificarse en tres tipos: estáticos, constantes y variables. Los operadores estáticos son aquellos que representan alguna palabra reservada del lenguaje, tales como el tipo de una variable (ej. en la gramática de Pascal las palabras reservadas BOOLEAN, ENTERO, etc.). Los operadores constantes son cadenas de caracteres ASCII, tales como los comentarios en un programa. Los operadores variables son aquellos que introducen automáticamente el nombre de las variables en una tabla de

símbolos (ej. en la figura 3-1 la variable "error" se inserta en la tabla de símbolos).

3.3.3 Operadores No-terminales.

Como tercera parte de la descripción se tiene los operadores no-terminales, cuya estructura es: nombre del operador, nombre(s) de la(s) clase(s) de su(s) descendiente(s), rutina de acción asociada, sinónimo y esquema de despligue. Por ejemplo de la figura 3-4 se tiene el operador IF-ELSE, el cual tiene tres descendientes, donde cada uno de ellos debe pertenecer a la clase: "expression", "statement" y "statement" respectivamente, su sinónimo es "IFE" y el esquema de despliegue es "IF @1 THEN@+@n@2; @-@nELSE@+@n@3;", donde "@1", "@2", "@3" indican el primer, segundo y tercer descendiente respectivamente, "@+" incrementa el nivel de indentación y "@n" inserta un cambio de línea en el despliegue. Después de aplicar el operador IF-ELSE, el despliegue en la pantalla queda como se muestra en la figura 3-3.

```
IF <expression> THEN
   <statement>;
ELSE
   <statement>;
```

Figura 3-3: Ejemplo de despliegue de un operador no-terminal.

3.3.4 Clases.

Finalmente tenemos las clases que no son más que conjuntos de operadores, cuya descripción consta del nombre de la clase y el nombre de los operadores que pertenecen a ella. Por ejemplo de la figura 3-4 tenemos que a la clase "expression" pertenecen los operadores no-terminales "EQUAL", "NOT" e "IDENT". Es decir, como se muestra en la figura 3-1, después de aplicar el operador IF-ELSE, el cual tiene tres descendientes, el cursor se posiciona en el primero, en donde es legal aplicar los operadores que pertenecen a la clase "expression", la aplicación de cualquier otro operador constituirá un error. De esta manera se garantiza que el programa que esté siendo editado sea siempre sintácticamente correcto.

3.3.5 Precedencia.

Los operadores no-terminales deben tener un valor de precedencia asociado a ellos; ésto se utiliza para proporcionar los paréntesis en forma automática en el esquema de despligue. Es importante señalar que esto es necesario en el despliege del texto en la pantalla para que no se induzca al usuario a confusión, dado que el árbol interno nunca es ambiguo. Cuando no se especifica en la descripción de la gramática la precedencia del operador no terminal, se toma una precedencia por omisión que es la precedencia más alta. En el apéndice C se presenta un ejemplo.

3.3.6 Rutinas de Acción.

Estas rutinas se utilizan básicamente para realizar pruebas semánticas. En esta versión no se tienen disponibles estas rutinas. Se incluyen estas rutinas en el formato de la descripción de la gramática para que en versiones posteriores del sistema se tenga la posibilidad de definir y utilizar dichas rutinas.

3.3.7 Sinónimo.

Cada operador terminal o no-terminal puede tener un sinónimo asociado. La función del sinónimo es la de permitir al usuario teclear comandos en forma más simple, por ejemplo, de la figura 3-4, en lugar de teclear la palabra "EQUAL" el usuario puede teclear "=".

3.4 Funcionamiento del Procesador de Gramáticas.

Para obtener un editor para un lenguaje determinado, al procesador de gramáticas se le debe proporcionar una descripción del lenguaje, que se puede crear utilizando un editor de texto o un EDOLEM: La gramática puede expresarse en términos de sí misma; es decir, si se genera un EDOLEM para describir gramáticas, éste se puede utilizar para crear o modificar las descripciones de gramáticas. Con lo cual no hay manera de producir gramáticas incorrectas y las tablas del lenguaje se producen automáticamente utilizando los esquemas de despliegue.

El procesador de gramáticas consta de dos pasos. En

```
{subpas.v1}
/* terminales */
{
IDENT           = {v}
                  |
                  | "'"
                  | "@s";
IDENTFILE       = {v}
                  |
                  | "'"
                  | "@s";
INTEGER         = {s}
                  |
                  | ""
                  | "INTEGER";
BOOLEAN         = {s}
                  |
                  | ""
                  | "BOOLEAN";
TRUE            = {s}
                  |
                  | ""
                  | "TRUE";
FALSE           = {s}
                  |
                  | ""
                  | "FALSE";}
/* Operadores no-terminales */
{
PROGRAM         = proghead decl compoundstate
                  |
                  |
                  | ""
                  | "@1@n@2:@n@3.";
PROGHEAD        = ident fileidents
                  |
                  |
                  | ""
                  | "PROGRAM @1@2:@n";
FILEIDENTS      = fileident
                  |
                  |
                  | ""
                  | "(@0, @q)";
VARDECLS        = vardecl
                  |
                  |
                  | ""
                  | "VAR@+@n@0:@n@q@-";
VARDECL         = idents type
                  |
                  |
                  | ""
                  | "@1 : @2";
```

Figura 3-4: Ejemplo de Descripcion de la gramatica.

```
IDENTS          = ident
                  |
                  |
                  | ""
                  | "@0, ";
ASSIGNSTATE     = leftside expression
                  |
                  |
                  | ":="
                  | "@1:= @2";
ADD             = expression expression
                  | (11)
                  |
                  | "+"
                  | "@1 + @2";
MULTIPLY        = expression expression
                  | (12)
                  |
                  | "*"
                  | "@1 * @2";
AND             = expression expression
                  | (12)
                  |
                  | ""
                  | "@1 AND @2";
NOT             = expression
                  | (13)
                  |
                  | ""
                  | "NOT @1";
EQUAL           = expression expression
                  | (10)
                  |
                  | "="
                  | "@1 = @2";
COMPOUNDSTATE   = statement
                  |
                  |
                  | ""
                  | "BEGIN@+@n@0:@n@q@-@nEND@eBEGIN@nEND";
IFELSESTATE     = expression statement statement
                  |
                  |
                  | ""
                  | "IF @1@+@nTHEN@+@n@2@-@nELSE@+@n@3@-@-";}
/* classes */
{
proghead        = PROGHEAD;
ident           = IDENT;
fileidents      = FILEIDENTS;
fileident       = IDENTFILE;
decl            = VARDECLS;
type            = INTEGER BOOLEAN;
vardecl         = VARDECL;
idents          = IDENTS;
compoundstate   = COMPOUNDSTATE;
statement       = ASSIGNSTATE COMPOUNDSTATE IFELSESTATE;
leftside        = IDENT;
expression      = EQUAL ADD MULTIPLY AND NOT IDENT TRUE FALSE;}
```

Figura 3-5: Ejemplo de Descripcion de la gramatica.

el primer paso crea una tabla de símbolos, la tabla de símbolos contendrá el nombre de los operadores del lenguaje sin diferenciar su tipo. En el segundo paso se analiza la descripción de los operadores terminales, no-terminales y las clases de operadores respectivamente y finalmente genera las tablas del lenguaje, las cuales se almacenan en el archivo abierto en el primer paso.

3.5 Algoritmo del Procesador de Gramáticas.

A continuación se presenta el algoritmo del procesador de gramáticas, en el cual, los puntos 1 a 3 constituyen el primer paso y el resto el segundo paso.

1. Abre el archivo de descripción de la gramática.
2. Lee el nombre del lenguaje.
3. Genera la tabla de símbolos.
4. Analiza los operadores terminales.
5. Analiza los operadores no-terminales.
6. Analiza las clases.
7. Si no existe error genera el archivo con las tablas del lenguaje.
8. FIN.

3.6 Como Generar un Editor.

En las figuras 3-4 y 3-5 se muestra un subconjunto de la gramática del lenguaje de programación Pascal, para ejemplificar como se debe describir la gramática. Con el fin de que el realizador pueda realizar comentarios, éstos se pueden colocar antes del nombre del lenguaje o de la

descripción de los operadores terminales, no-terminales o las clases de operadores. Los comentarios se deben escribir de la siguiente forma: primero la secuencia "/*", que le indica al procesador de gramáticas que lo que sigue es un comentario, este se salta el texto hasta que encuentra la secuencia "*/". En el apéndice C se dá un ejemplo más detallado de como generar un EDOLEM.

CAPITULO 4

Manejo de Memoria

4.1 Introducción.

La memoria en una computadora está organizada en forma jerárquica, comprendiendo al menos dos niveles de jerarquía: la memoria principal y la memoria secundaria. Dado que la manipulación de la información se realiza en la memoria principal, cuando la información a manejar rebasa el tamaño de ésta, es necesario contar con un mecanismo de transferencia entre la memoria principal y la memoria secundaria, provocando el problema de determinar a cada momento en que nivel de memoria debe residir la información. Para resolver este problema se cuenta con un mecanismo que le crea al usuario la ilusión de que el espacio de memoria que emplea es mayor que

la memoria principal disponible y se conoce como esquema de memoria virtual [DENNING 70].

El esquema de memoria virtual utiliza un mecanismo de transformación (o mapeo de direcciones) para transformar las direcciones utilizadas por el programador a las correspondientes direcciones de memoria física. El problema fundamental es la distinción entre las direcciones de programa (direcciones usadas por el programador) y las direcciones físicas a las que se mapean las direcciones virtuales. Las direcciones virtuales se conocen como espacio direccionable mientras que las direcciones físicas se conocen como espacio de memoria [LISTER 79]. Si las direcciones virtuales las denotamos por N y el espacio de memoria por M, estonces el mapeo lo podemos denotar por:

$$f : N \dashrightarrow M$$

El manejo de memoria generalmente ha sido función del sistema operativo, el cual le permite al usuario ejecutar sus programas sin la obligación de conocer en que parte de la memoria se carga y ejecuta su programa. El manejo de memoria virtual se ha implantado en máquinas multiusuario, en las cuales se pueden tener varias tareas corriendo al mismo tiempo y es necesario garantizar que dos o más tareas corriendo al mismo tiempo no se interfieran; sin embargo en máquinas monousuario la mayoría de los sistemas operativos no contemplan este esquema.

Otro de los problemas que se presenta cuando se tiene poco espacio de memoria, es que para programas que excedan este espacio es necesario implantar un esquema que permita tener una parte del programa ejecutándose en memoria principal y la otra en memoria secundaria; uno de los esquemas utilizados para resolver este problema es el manejo de "overlay"[5] [WATSON 70].

4.2 Manejo de Memoria Virtual.

Existen tres métodos para el manejo de memoria virtual. Cada método agrupa la información en bloques, en donde cada bloque está constituido por direcciones contiguas. El primer método (segmentación) organiza el espacio direccionable en bloques de longitud variable, el segundo método (páginación) organiza el espacio direccionable en bloques de longitud fija, por último, el tercer método (páginas segmentadas) combina los dos primeros métodos [DENNING 70].

4.2.1 Método de Segmentación.

Como se mencionó anteriormente, este método organiza el espacio direccionable en bloques de longitud variable, denominados segmentos. Un segmento es un agrupamiento de información (una rutina, un registro de datos,etc.) que se

---

[5] PROGRAMA CON "OVERLAY".- es un programa segmentado en el cual el segmento que se está ejecutando ocupa la misma área de memoria que el segmento previamente ejecutado [KURZBAN 75].

puede tratar como una unidad; cada segmento tiene asociada una longitud. Ademas de la longitud del segmento, para poder referenciarlo y utilizarlo se requieren otros datos que caracterizan al segmento; esto se tiene en un descriptor del segmento; sus componentes son: el nombre del segmento (se utiliza para identificarlo), el estado (nos indica si el segmento se encuentra en memoria principal o no), la longitud, la dirección base en memoria principal y la dirección base en memoria secundaria. Considerando que la longitud de los segmentos sea un múltiplo de 2 y una longitud máxima de segmento de 128 bytes, la dirección base en memoria secundaria se representa por 2 Bytes, la dirección base en memoria principal por 12 bits, la longitud del segmento por 3 bits y el estado de un segmento por 1 bit (está presente en memoria principal o no), el descriptor de un segmento tendrá un tamaño de 4 Bytes, como se muestra en la figura 4-1.

```
struct segmento              /* Tabla de segmentos      */
{
   int dbs;                  /* Dir. base Mem. secundaria*/
   unsigned desc_dbp : 12;/* Dir. base Mem. Principal */
   unsigned desc_lon : 3; /* longitud del segmento    */
   unsigned desc_sta : 1; /* estado                   */
}
```

Figura 4-1: Tabla de descriptores de segmentos.

4.2.2 Método de Paginación.

En este método, al igual que en el método de segmentación, se tienen bloques de información, solo que la

longitud de estos bloques es fija. La información que se requiere para manipular la información en este esquema es: nombre, estado, la base en memoria principal y la base en memoria secundaria, cuyas funciones son las mismas que en el método de segmentación. Dado que se tiene una tabla de páginas físicas y una tabla de páginas virtuales, el descriptor de una página ocupará 8 bytes, como se muestra en la figura4-2.

```
struct pagf                  /* tabla de páginas físicas */
{
    unsigned desp_mf : 14;/* apuntador memoria física  */
    unsigned desp_omd : 2;/* ocupada, modifica         */
    char frec;             /* frecuencia de uso         */
    char dirs;             /* dirección memoria virtual */
}

                    ( a )

struct pagv             /* tabla de páginas virtuales */
{
    int ptrpf;             /* apuntador memoria física */
    unsigned pv_fsp : 10;/* apuntador espacio libre  */
    unsigned pv_lfr : 5; /* tamaño del espacio libre */
    unsigned pv_mse : 1; /* está en memoria principal*/
                         /* o no                     */
}

                    ( b )
```

Figura 4-2: Estructura de las tablas de manejo de memoria virtual. a).- tabla de páginas físicas. b).- tabla de páginas virtuales.

4.2.3 Selección del Método de Manejo de Memoria.

En este trabajo se utiliza el método de páginación, el cual se seleccionó despues de hacer un análisis de costos, por lo cual a continuación presentamos una descripción breve de dicho análisis. Los costos están formados por dos aspectos:

costo de memoria y costo de tiempo. Empezaremos por analizar el costo de tiempo.

Si denotamos el tiempo total de transporte de un bloque de información por T(z), el tiempo de arranque del motor que hace girar el disco por ta, el tiempo que tarda en dar una revolución por tr, el tiempo promedio de posicionamiento por tp, el tiempo de transferencia de un bloque por tt, el número de bytes a transferir por z y el número de bytes por pista por w, tendremos:

$$T(z) = ta + tp + tr/2 + tt. \quad \text{----------(4.1)}$$

$$tt = tr * z/w. \quad \text{----------(4.2)}$$

sustituyendo 4.2 en 4.1 tenemos:

$$T(z) = ta + tp + tr(1/2 + z/w). \quad \text{---(4.3)}$$

Donde, para la microcomputadora que utilizamos, tenemos:

ta = 250 mseg (máximo).

tp = 100 mseg.

tr = 200 mseg.

Por lo que, finalmente tenemos:

$$T(z) = 450 + 200 * z/w. \quad \text{----------(4.4)}$$

donde: w = 2304 Bytes.

Por último, considerando que un segmento promedio tiene un tamaño de 32 Bytes y tomando como base un tamaño de página de 1024 Bytes (1 KB), los tiempos de transporte para un segmento y para una página son:

Tsegmento = T(32) = 452.78 mseg.

Tpágina = T(1024) = 538.89 mseg.

Sin embargo, la transferencia de información entre la memoria principal y la memoria secundaria se realiza por sectores de 128 bytes. Dado que el proceso de construcción y edición de un programa es dinámico y aleatorio, no se puede garantizar que los segmentos que se requieran se encuentren en el mismo sector, ya que para que ésto sea posible, se tendría que reacomodar periódicamente los segmentos. Por lo tanto el tiempo de transferencia de un segmento es:

Tsegmento = T(128) = 461.11 mseg.

Por lo tanto, el costo de transferencia de un segmento es 14.45 % menor que el costo de transferencia de una página, tabla 4-1.

Considerando que se pudiera contar con un disco duro (tal como el SA706/712 de Shugart), con las siguientes características:

ta = 0 mseg (el motor del disco siempre está encendido)

tp = 99 mseg (promedio)

tr = 8.4 mseg (promedio)

w = 8192 Bytes

el tiempo de transferencia de una página de 1 Kbyte y el tiempo de transferencia de un sector, sería:

T(1024) = 104.25 mseg.

T(128) = 103.33 mseg.

Con lo cual el costo de velocidad se reduciría en un 517 % y 446.24 % respectivamente.

Analizaremos ahora los costos de memoria, para lo cual partiremos de las mismas consideraciones anteriores de tamaño de un segmento y tamaño de una página. Como se mencionó anteriormente, el descriptor de un segmento tiene un tamaño de 4 Bytes y el descriptor de una página tiene un tamaño de 8 Bytes (figura 4-2), como se muestra en la tabla 4-1.

En la tabla 4-1 se muestra los costos de utilizar segmentación o páginación, en donde podemos apreciar que para una página de 1024 Bytes, el costo en memoria de las tablas para el manejo de cada una de las páginas es 0.77 % de la información útil. Por otro lado, considerando un segmento promedio de 32 Bytes, el costo de las tablas para el manejo de cada uno de los segmentos es el 12.5 % de la información útil. Por otro lado, si consideramos un segmento promedio de 128

Bytes, el costo sería del 3.2 %.

Método	tiempo de transporte mseg.	costo de memoria %
Segmentación (32 Bytes)	452.78	12.5
Segmentación (128 Bytes)	452.78	3.2
Paginación	538.89	0.77

Tabla 4-1: Costos de manejo de memoria virtual.

Del análisis anterior podemos observar que el método de segmentación tiene un costo menor en tiempo que el método de páginación, pero tiene un costo mayor en memoria. Dada esta situación es necesario escoger alguna alternativa. En este trabajo se decidió utilizar el método de páginación por ser su costo en memoria mucho menor que el costo del método de segmentación y ser el costo de memoria crucial mientras que la diferencia del costo en tiempo es secundaria.

Una vez seleccionado el método de páginación es necesario determinar el tamaño de la página a utilizar. Del análisis anterior se induce que el aumento del tamaño de la página aumenta el costo en tiempo y disminuye el costo en memoria. Por ejemplo; considerando 1 KB como tamaño base de una página, y que incrementamos éste a 2 KB, los costos son:

Costo de tiempo:

T(1024) = 538.89 mseg.

T(2048) = 728 mseg.

Costo de memoria (en por ciento):

M(1024) = 0.77 %

M(2048) = 0.39 %

De los cálculos anteriores se puede apreciar que el costo en tiempo se incrementa en 35 %, mientras que el costo en memoria se decrementa en 50 %. Sin embargo, dado que el espacio de memoria es pequeño, para que el método sea funcional se requiere tener un número mínimo de páginas. En nuestro caso, como se muestra al final del capítulo (figura 4-7), el espacio de memoria es de 4 KB. Bajo estas consideraciones se decidió utilizar páginas de 1 KB.

4.2.4 Realización del Esquema de Memoria Virtual.

Dado que el esquema a realizar es el método de páginación, surge un aspecto más a considerar, que es cómo controlar y manipular el espacio libre en una página. Este control se puede establecer a través de dos métodos, a saber; se puede utilizar un arreglo binario del mapa de memoria o una lista ligada de bloques libres.

4.2.4.1 Arreglo Binario del Mapa de Memoria.

En este método se utiliza un bit que nos indica si un bloque de memoria está ocupado o libre, es decir, un 1 nos

indica que el bloque está ocupado y un 0 que está libre. Donde el tamaño del bloque está determinado por las características de la aplicación.

4.2.4.2 Lista de Bloques Libres.

En este método se tiene un apuntador al primer bloque libre y un contador que nos indica el tamaño del bloque. En el bloque se construye a su vez un apuntador al siguiente espacio libre y su contador correspondiente, y así sucesivamente; finalmente, el último bloque tendrá un apuntador nulo (figura 4-3).

4.2.4.3 Selección del Método.

Para poder seleccionar alguno de estos métodos tenemos que partir nuevamente del análisis del costo de utilizar uno ú otro. Considerando que el acceso a disco se realiza por sectores (128 Bytes), y tomando este tamaño como el tamaño de un bloque, el costo en memoria del esquema de un arreglo binario del mapa de memoria sería del 1 % de la información útil. Dado que el proceso de construcción y edición de un programa es dinámico y aleatorio, y que en un bloque se podría tener varios nodos del árbol sintáctico, un bloque se marcaría como ocupado solo cuando esté lleno, por lo cual se tendría que compactar el árbol cuando se libere espacio dentro de un bloque marcado como ocupado, lo cual nos consumiría tiempo de procesamiento. Por otro lado, utilizando el esquema de lista ligada, en la tabla de descripción de la página virtual se tiene un apuntador al inicio del espacio

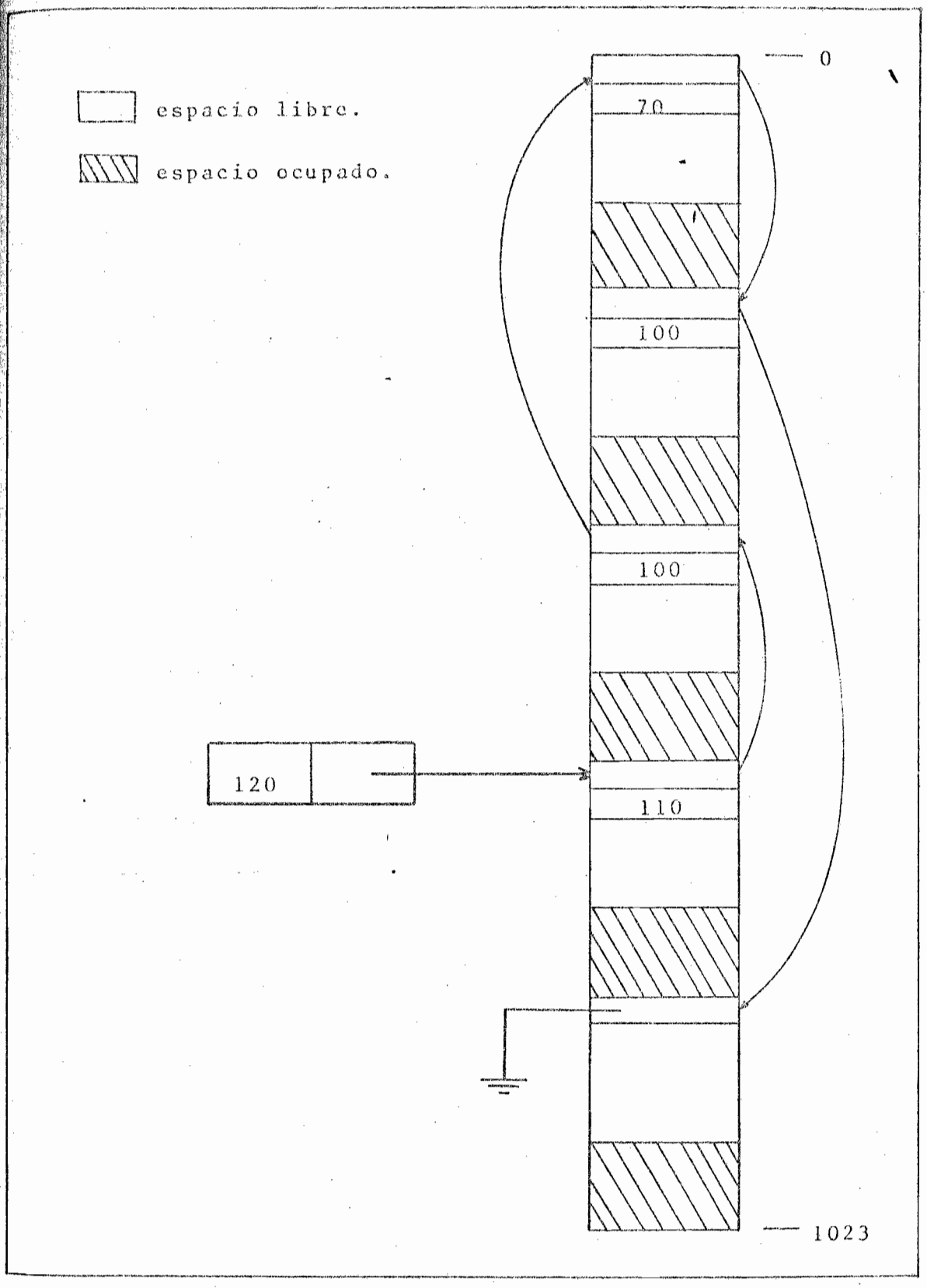


Figura 4-3: Manejo de espacio libre en una página.

libre y un contador de espacio libre, mientras que la lista se construye sobre la misma página, en este último caso se utilizan 4 Bytes, que para una página de 1 Kbyte corresponde al 0.37 % de la información útil. Sin embargo, cuando el tamaño del espacio libre al que se apunta en el descriptor de la página es menor que el espacio solicitado, se tiene el problema de buscar espacio en una lista que se encuentra en memoria secundaria ("trashing"), este problema se resuelve reacomodando la lista cuando la página se encuentra en memoria principal, de forma tal que en el descriptor de la página se apunta al mayor espacio libre. Se decidió utilizar este último método dado que su costo de memoria es bajo y no se tiene el problema de compactación.

Por último, se debe tener un sistema de mapeo, el cual, además de transformar las direcciones virtuales a direcciones físicas, debe establecer una política de asignación de espacio. Básicamente existen tres estrategias de mapeo [LISTER 79]:

1. Remplazar la página mas antigua (FIFO)[6]

   Este es el algoritmo mas simple y su costo consiste solo en "recordar" la secuencia de carga de las páginas.

2. Remplazar la página menos recientemente usada

---

6 First-In-First-Out

(LRU)[7].

> En este algoritmo se asume que la probabilidad de referenciar una página es proporcional al intervalo de tiempo entre la última referencia a la página y el momento actual [HABERMANN 76].

3. remplazar la página menos usada (LFU)[8].

> Este algoritmo asume que la probabilidad de referenciar una página está correlacionada con la frecuencia de uso de la página [HABERMANN 76].

Estudios de simulación [COLIN 71] mostraron las diferencias de rendimiento de los tres algoritmos (en términos del número de transferencias durante la ejecución de un número de tareas) que varía de acuerdo al tipo de tareas. El algoritmo (a) fué el peor en general [LISTER 79]. La dificultad inherente a escoger entre el método (b) y el (c) reside en que no es posible tener un conocimiento preciso de la probabilidad de referenciar una página y el rendimiento depende de la aplicación. En este trabajo se utiliza el método de remplazar la página menos usada, asumiendo que la página cuya frecuencia de uso es mayor, tiene mayor probabilidad de ser requerida; dado que la mayoría de los nodos que se requieren residirían en la misma página.

---

[7] Least Recently Used

[8] Least Frequently Used

Finalmente, la realización de este método se lleva a cabo mediante los siguientes componentes:

- Una tabla de páginas físicas.- En esta tabla se tiene la siguiente información: Nombre, dirección física de inicio, direccion virtual de inicio, frecuencia de uso, estado de la página (ocupada, modificada, no modificada), como se muestra en la figura 4-2.a.

- Una tabla de páginas virtuales.- Con la siguiente información: nombre, dirección física, apuntador a el espacio libre, longitud del espacio libre, estado (presente en memoria principal o nó).figura 4-2.b.

- Una rutina de mapeo y remplazo de páginas.- Dada una dirección virtual, esta rutina verifica si ésta tiene una dirección física asociada o no, en caso de no tenerla libera una página física, y por último, en ambos casos regresa la dirección física asociada.

4.2.4.4 Algoritmo de Manejo de Memoria Virtual.

1. Calcula la página virtual.
2. Verifica si existe la página en memoria principal.
3. No existe la página en memoria principal.
    a. Verifica si existe una página libre en memoria principal.
    b. Verifica si la página a utilizar se ha modificado.
    c. Mueve la página a remplazar a memoria secundaria.
    d. Mueve la página a memoria principal.
    e. Inicializa frecuencia de uso.
4. Existe la página en memoria principal.
    a. Incrementa la frecuencia de uso.

5. Calcula la dirección física.
6. Regresa la dirección física.

4.3 Manejo de las Tablas del Lenguaje.

Otro aspecto importante a considerar, es que el EDOLEM tiene que manejar las tablas de la gramática del lenguaje. Estas tablas ocuparán también un espacio de memoria, el cual dependerá del tamaño del lenguaje. Existen dos alternativas para este manejo:

1. Manejar las tablas en disco.
2. Manejar las tablas en memoria.

Dado que nuestro sistema se encuentra limitado en memoria, se analizó la posibilidad de manejar las tablas en disco. Sin embargo, el tener las tablas en disco representa un problema de velocidad, ya que el sistema se encuentra constantemente interactuando con ellas. Por lo cual se realizó un análisis de costos para tratar de encontrar alguna alternativa que nos permitiera tener un costo bajo en memoria y un costo aceptable en tiempo de respuesta.

Para este análisis partimos de las características de dichas tablas. Como puede verse en la figura 4-4, para cada operador del lenguaje la información a manejar es: índice en la tabla de operadores, nombre del operador, orden del operador, tipo de operador, precedencia, apuntador a la rutina de acción, apuntadores a las hojas o ramas, sinónimo, número de esquemas de despliegue y el esquema de despliegue; mientras

que por otro lado, cada elemento clase esta constituído por: índice en la tabla de operadores, nombre de la clase, número de operadores de la clase, tipo de operador y apuntadores a los operadores de la clase.

```
struct tnodeinf        /* tabla de operadores    */
{
    int infopt;        /* índice                 */
    char operator[15];/* nombre del operador     */
    char arity;        /* orden                  */
    char terminal;     /* tipo de nodo           */
    char precedence;   /* precedencia            */
    int semantic;
    char sons[];       /* apunt. a descendientes */
    char synonym[3];   /* sinónimo del operador  */
    char numofunpsch; /* No. esq. despliegue     */
    char *unpsch1;     /* Esquema de despliegue  */
};
```

(a)

```
struct strclass         /* tabla de clases       */
{
    int opt;            /* índice                */
    char printname[15];/* nombre de la clase     */
    char numofelements;/* No. Miembros           */
    char terminal;      /* tipo                  */
    int members[];      /* Miembros de la clase  */
};
```

(b)

Figura 4-4: tablas de descripción de los operadores del lenguaje.

Considerando que en promedio, el descriptor de cada operador ocupa 50 Bytes y utilizando como ejemplo la descripción de la gramática de pascal estandar que se muestra en el apéndice D, el espacio que ocuparía la tabla del lenguaje sería de 12.5 Kbytes. Tomando en cuenta que el acceso a disco se realiza por sectores y que el tamaño de

éstos es de 128 Bytes, tenemos dos alternativas para el manejo de las tablas en disco: almacenar la información de cada operador por sector o almacenar las tablas en la misma forma que en memoria principal. Si almacenamos cada operador por sector, el costo en disco del almacenamiento aumentaría en un 54 %; sin embargo, esto nos permitiría tener un acceso directo a la información, con lo cual el acceso a la información sería mas rápido. Para este acceso se establece una estructura de datos, que consiste en tener una tabla reducida de operadores en memoria que contiene el índice del operador, que para este caso es el número de sector, el nombre del operador y el sinónimo del operador, como se muestra en la figura 4-5.a.

Si almacenamos cada operador en la misma forma que en memoria principal, el costo de almacenamiento en disco sería el mismo costo que si se almacenara en memoria principal, pero en la estructura de datos para manejar las tablas sería necesario tener un apuntador a la dirección en disco del operador, y un indicador del tamaño del operador, como se muestra en la figura 4-5.b. Además habría que calcular a partir del apuntador el sector donde empieza el operador y si éste ocupa parte de otro sector, sería necesario realizar dos lecturas a disco para conformar el operador en memoria y dado que el 33 % de los operadores estarían almacenados una parte en un sector y la otra en el siguiente, se haría muy lento el acceso.

Despues de analizar los costos de utilizar un método

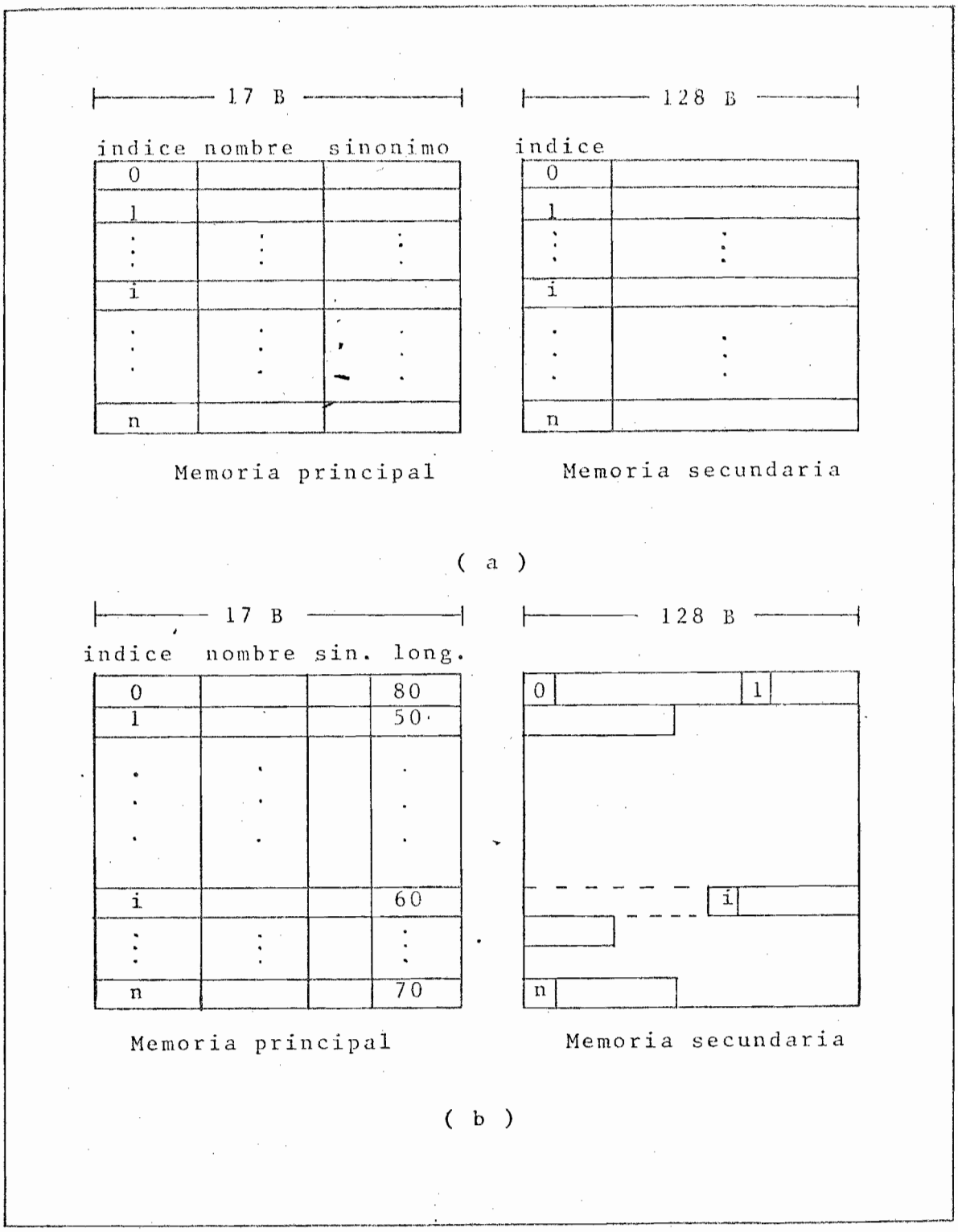


Figura 4-5: Manejo en disco de las tablas del lenguaje.

u otro se decidió experimentar la funcionalidad del primero, obteniéndose un funcionamiento muy pobre (en lo que respecta a velocidad), con tiempos de respuesta sumamente altos.

Finalmente se decidió manejar las tablas en memoria, reservándose para este efecto un espacio de memoria de 12.5 Kbytes. Este espacio nos permite manejar las tablas del lenguaje de programación Pascal [JENSEN 75], las cuales están constituídas por 190 operadores y ocupa un espacio de 8.8 Kbytes. Por lo cual se podría generar un EDOLEM para un lenguaje cuya gramática fuera aproximadamente 20 % mayor que la gramática de Pascal estandar.

4.4 Manejo de Overlays.

Finalmente, en el aspecto del manejo de memoria, nos encontramos frente al problema de la dimensión del EDOLEM [9], cuyo tamaño es mayor que la memoria principal disponible. La microcomputadora utilizada en este trabajo tiene un tamaño de memoria de 64 Kbytes, de los cuales el sistema operativo utiliza 15 Kbytes, quedando solamente 49 Kbytes para el desarrollo de programas. La magnitud del EDOLEM es de 57 Kbytes, de los cuales 54 Kbytes son de código y 3 Kbytes de variables globales del sistema. Por lo tanto, es indispensable realizar un manejo de "overlays", que permita la utilización del sistema.

---

9 Nos referimos al tamaño del programa ejecutable

Considerando que las variables globales estarán residiendo permanentemente en memoria, el espacio disponible se reduce a 46 Kbytes. Si ademas de ésto se toma en cuenta que se tienen rutinas recursivas y que los parámetros de las rutinas se pasan a través del "stack", tenemos que reservar un área para este propósito, en este caso se reserva un área de "stack" de 1 Kbyte. Como se mencionó anteriormente, las tablas del lenguaje ocupan 12.5 Kbytes. Por último, dado que el sistema es interactivo con el usuario, se reserva un espacio para la edición y construcción de los programas del usuario, que virtualmente es de 64 Kbytes, pero físicamente es de 4 Kbytes. Por lo tanto, el espacio de memoria disponible para el programa se ha reducido a 28.5 Kbytes.

Nuevamente, el problema de espacio de memoria, nos remite al problema de velocidad de respuesta del sistema. Por lo tanto, se debe realizar un análisis de como se realiza el manejo de los "overlays". En el esquema aquí utilizado se tiene una parte del programa residiendo en memoria principal permanentemente ("NUCLEO") y dos áreas de memoria para colocar las rutinas a utilizar ("overlays"). A esta decisión se llegó después de hacer una clasificación de las rutinas, estableciéndose un conjunto de rutinas de interpretación de comandos de edición y un conjunto de rutinas de propósito general (manejo de listas, rutinas de despliegue, etc). Quedando las áreas de "overlay" organizadas de la siguiente forma: En el área OV1 se cargan y ejecutan las rutinas de

interpretación de comandos y en el área OV2 las rutinas de propósito general, como se muestra en la figura 4-6.

Dado que el manejo de memoria virtual y el manejo de los "overlays" hacen que la respuesta del sistema sea más lenta y que una vez cargado un conjunto de rutinas, éstas se quedan residiendo en memoria principal hasta que el espacio sea ocupado por otras rutinas, se cuenta con un par de banderas, una por cada área de "overlay", que nos indican que rutinas están presentes en memoria principal, para que en caso de utilizarse éstas, no se haga un nuevo acceso a disco.

Bajo estas consideraciones, la distribución del espacio de memoria física, queda establecido como se muestra en la figura 4-7.

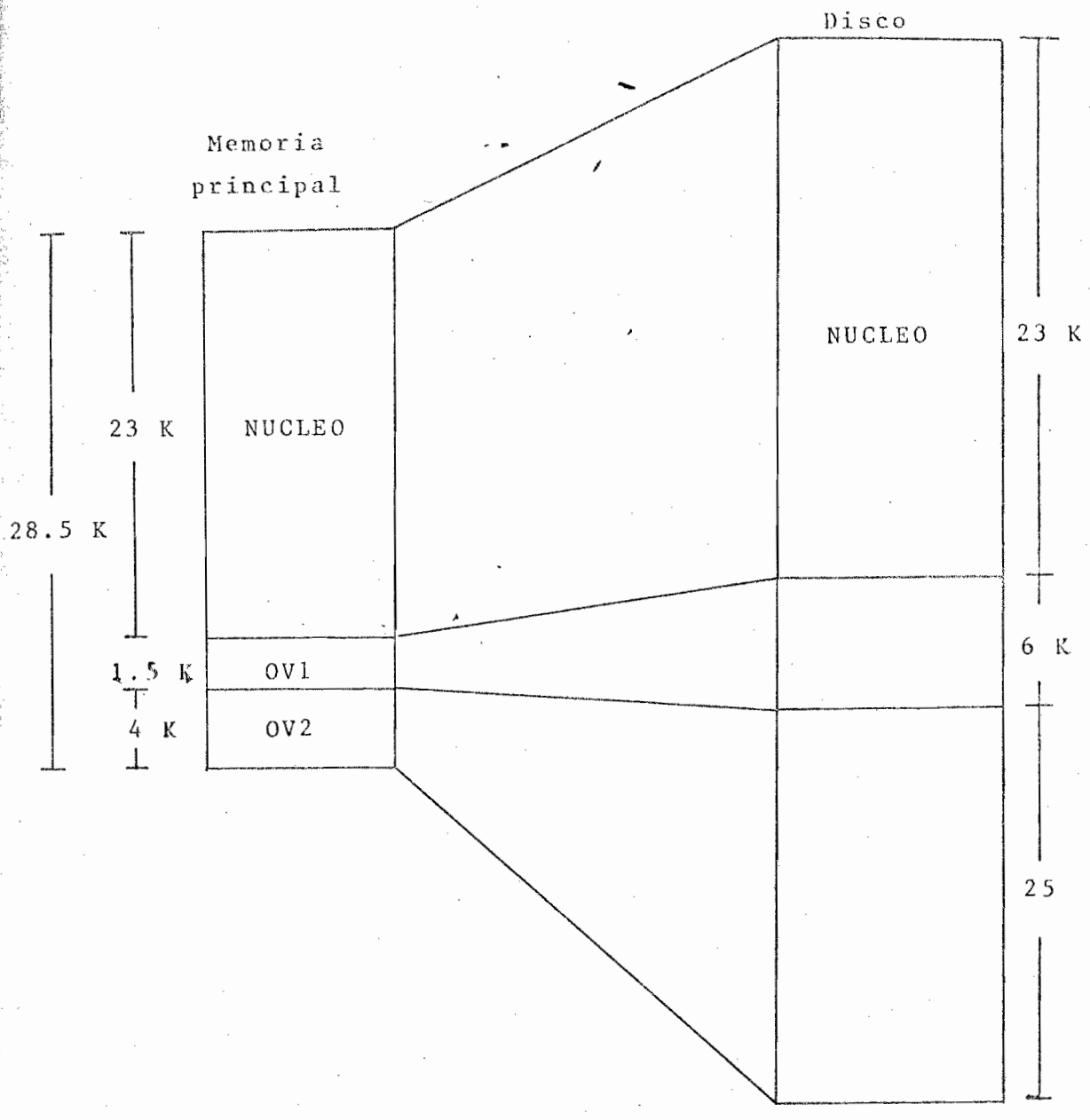


Figura 4-6: Manejo de overlays.

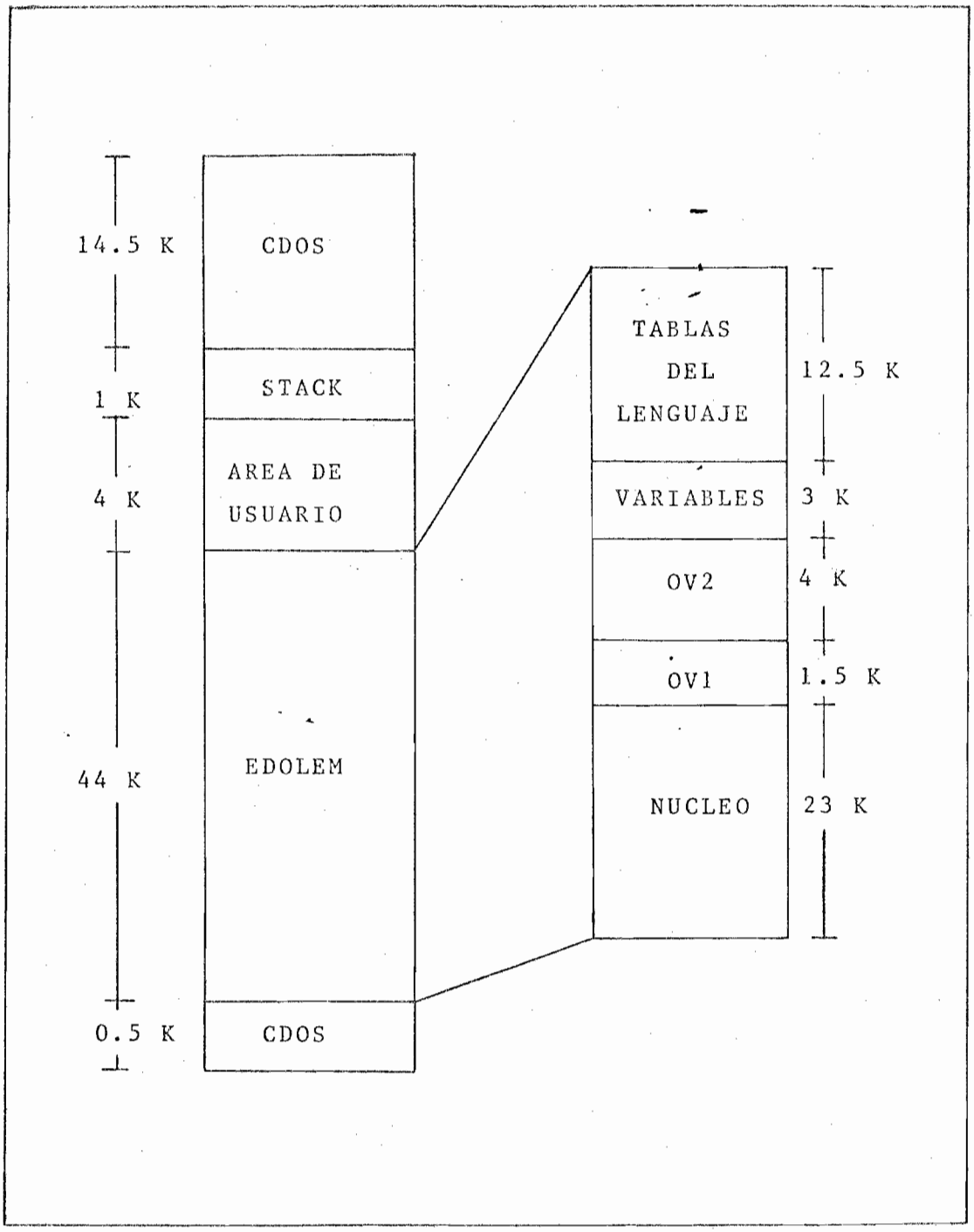


Figura 4-7: distribución del espacio de memoria.

CAPITULO 5

Representación Interna

5.1 Introducción.

Como se ha mencionado en los capítulos precedentes, el EDOLEM construye y manipula estructuras de árbol; las cuales están constituidas por diferentes tipos de nodos. En un EDOLEM se cuenta con tres tipos de nodos: nodos no-terminales, nodos terminales, y meta-nodos. En este capítulo se describe la forma en que se manipula y representa internamente cada tipo de nodo asi como sus características.

5.2 Nodo Genérico.

Todos los nodos del árbol tienen dos campos en común: el campo OPT, donde OPT es la entrada a la tabla de operadores del lenguaje en donde se encuentra la descripción del nodo

correspondiente; y el campo FATHPTR que es un apuntador al padre del nodo. En la figura 5-1 se muestra estos campos comunes a todos los nodos.

```
struct tnodeb
{
     int opt;              /*identificador operador*/
     struct tnode *fathptr;/* apuntador al padre   */
}
```

Figura 5-1: Características comunes a todos los nodos.

5.3 Nodos No-terminales

Los nodos no-terminales, como se mencionó en el capítulo 3, pueden ser de orden fijo o de orden variable, es decir; cada nodo no-terminal puede tener un número fijo de descendientes o una lista de descendientes, por ejemplo; de la gramática de Pascal mostrada en el apéndice D, el operador IF-ELSE está representado por un nodo de orden fijo (orden 3) como se muestra en la figura 5-2. Por otro lado, el operador BLOCK esta representado por un nodo de orden variable (lista) como se muestra en la figura 5-3. La forma en que se representan y manipulan los nodos de orden fijo y los nodos de orden variable es diferente, como se puede ver en las figuras 5-4 y 5-5.

5.3.1 Nodos de Orden Fijo.

Los nodos de orden fijo son aquellos que tienen un número fijo de descendientes. En este tipo de nodos los

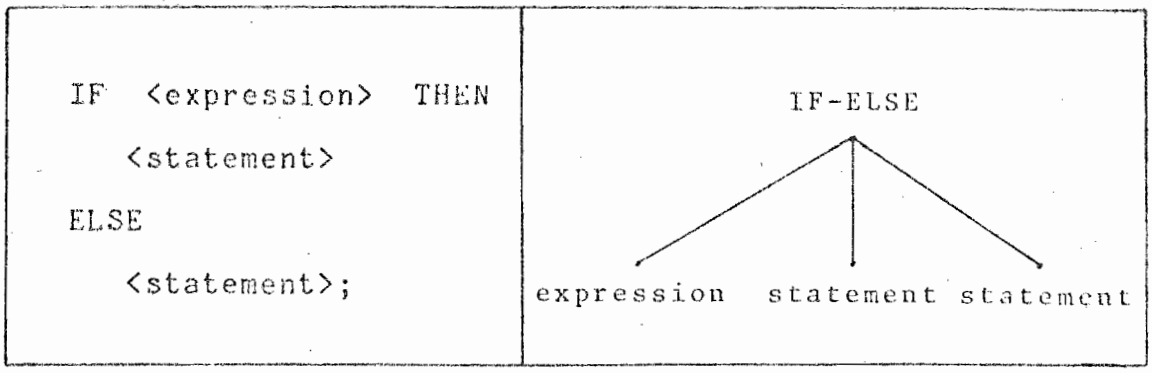


Figura 5-2: Ejemplo de un nodo no-terminal de orden fijo.

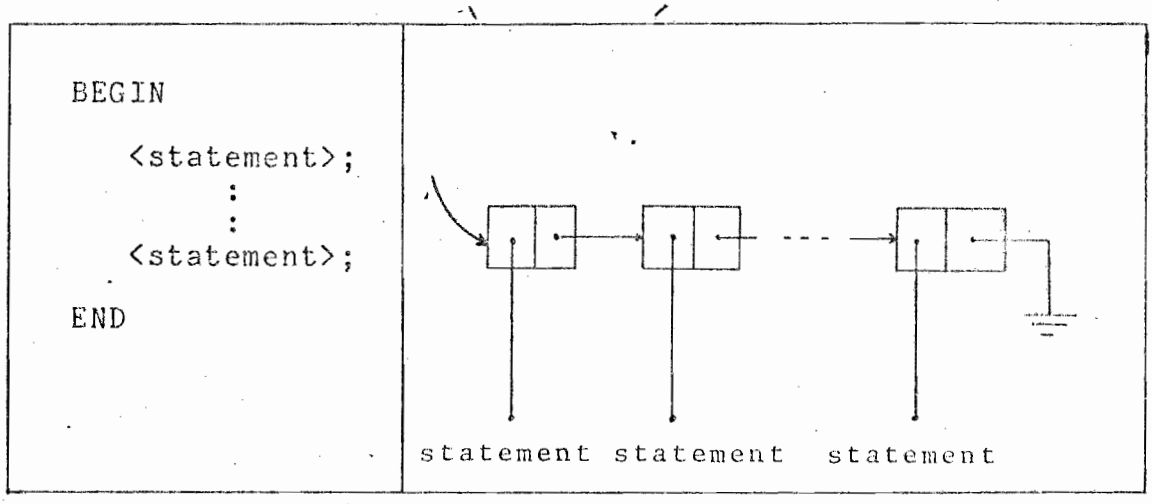
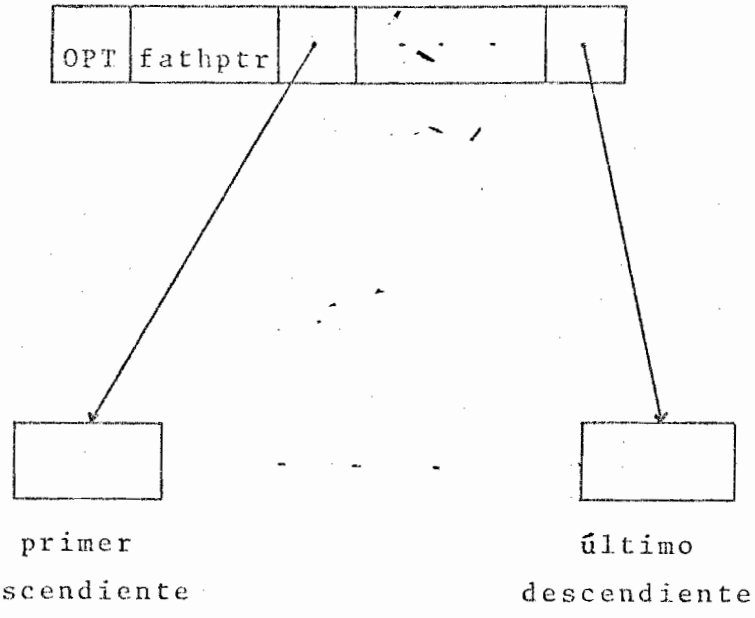


Figura 5-3: Ejemplo de un nodo no-terminal de orden variable.

```
struct tnode
{
      int opt;
      struct tnode *fathptr;
      struct tnode *sonsptr[];/* apuntadores a los */
                              /* descendientes     */
}
```

(a)



(b)

Figura 5-4: Descripción de un nodo no-terminal de orden fijo.

apuntadores a los descendientes nunca pueden apuntar a descendientes nulos, por lo cual, al crear este nodo se crean meta-nodos que representan la clase a que debe pertenecer cada uno de los descendientes, por ejemplo; en la figura 5-2, el nodo IF-ELSE tiene tres descendientes. En la figura 5-4 se

muestran los campos que constituyen un nodo de orden fijo, en el cual, ademas de los campos comunes a todos los nodos se tiene los apuntadores a los descendientes, representado por el campo SONSPTR.

5.3.2 Nodos de Orden Variable.

Los nodos de orden variable tienen un conjunto de descendientes indeterminado, los cuales se manipulan a través de una estructura de lista ligada (que puede ser una lista vacía), es decir, en los nodos de orden variable se tiene un apuntador a una lista de descendientes, éste apuntador es una cabeza de lista. En este tipo de nodos se crean nodos lista a través de los cuales se ligan los descendientes, como se muestra en la figura 5-5.b. En la figura 5-5.a se muestra los campos de los nodos lista, estos son: NEXTPTR que apunta al siguiente elemento de la lista y NODE que apunta al descendiente propiamente dicho. Como puede apreciarse en la figura 5-5.b, los nodos lista son solamente nodos intermedios, ya que los descendientes del nodo no apuntan a la lista, sino a la cabeza de dicha lista, que es el operador (nodo) padre.

5.4 Nodos Terminales.

Los nodos terminales, como su nombre lo indica, forman las hojas del árbol y por lo tanto no tienen descendientes. Estos nodos están clasificados en tres tipos: estáticos, constantes y variables.

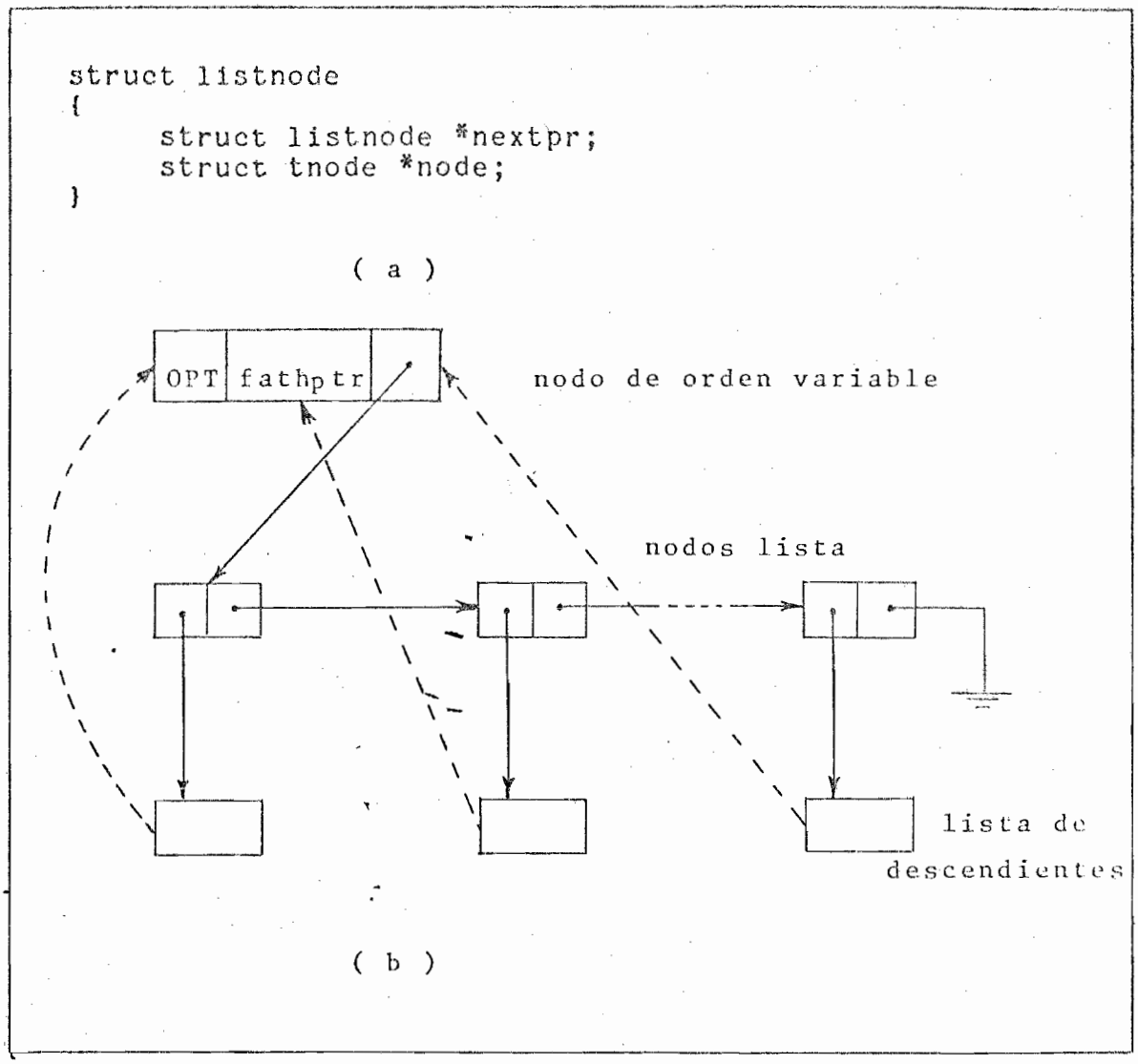


Figura 5-5: Descripción de un nodo no-terminal de orden variable.

5.4.1 Nodos Estáticos.

Estos nodos representan palabras reservadas del lenguaje, tales como el tipo de una variable(ej. REAL, ENTERO, etc.), por lo cual la estructura de estos esta constituida por los elementos básicos comunes a todos los nodos, como se muestra en la figura 5-6.

```
struct tnodes
{
     int opt;
     struct tnode *fathptr;
}
```

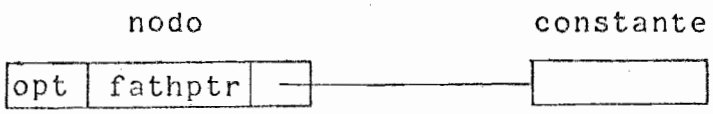
Figura 5-6: Representación de un nodo terminal estático.

5.4.2 Nodos Constantes.

Los nodos constantes se utilizan para manipular y representar a los operadores constantes del lenguaje, tales como comentarios, constantes enteras, reales, etc., por lo cual, los nodos de este tipo, ademas de tener los campos comunes a todos los nodos tienen el campo NOMBRECT, que es un apuntador a la constante en cuestión; como se muestra en la figura 5-7.

```
struct tnodec
{
     int opt;
     struct tnode *fathptr;
     char *nombrect;        /* apuntador a la constante */
}
```

(a)



(b)

Figura 5-7: Representación de un nodo terminal constante.

5.4.3 Nodos Variables.

Los nodos variables se utilizan para representar y manejar las variables del usuario, para lo cual, ademas de los campos comunes a todos los nodos tiene el campo LLAVE, el cual es un apuntador a la variable en la tabla de símbolos correspondiente; como se muestra en la figura 5-8. Es importante señalar la diferencia entre los nodos constantes y los variables, los nodos constantes apuntan al nombre o cadena de caracteres mientras que los nodos variables apuntan a la tabla de símbolos donde se inserta la variable correspondiente.

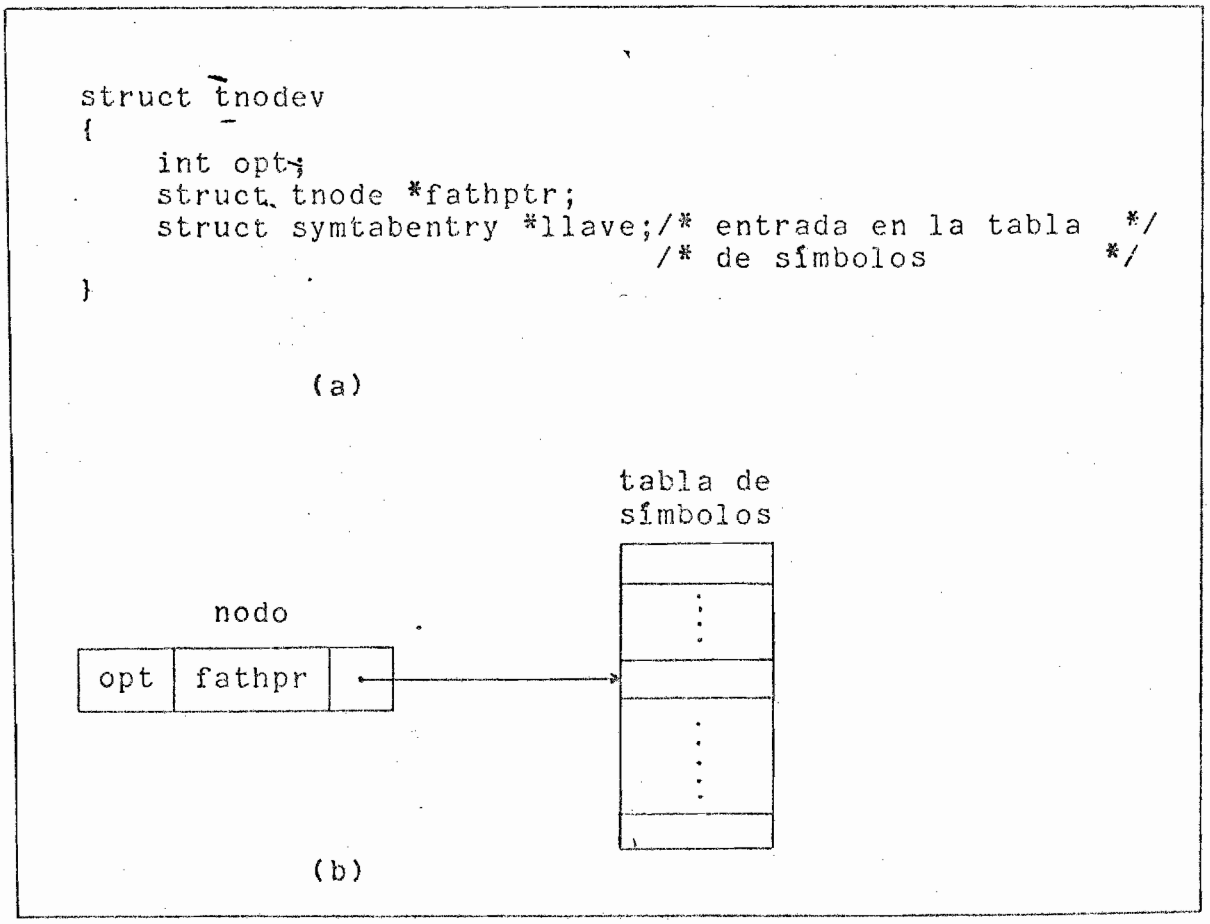


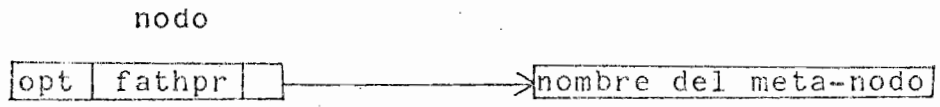
Figura 5-8: Representación de un nodo terminal variable.

5.5 Meta-nodos.

Los meta-nodos son nodos temporales que se remplazan por medio de comandos de construcción, el remplazo de estos se realiza de acuerdo a la descripción de la gramática, es decir, antes de remplazarlo se verifica que el comando aplicado pertenezca a la clase indicada por el meta-nodo, dado esto, este tipo de nodos estan formados, ademas de los campos comunes a todos los nodos por el campo METANOM, que es un apuntador al nombre del meta-nodo o nombre de la clase representada por el meta-nodo; como se muestra en la figura 5-9.

```
struct tnodem
{
    int opt;
    struct tnode *fathptr;
    char *metanom;      /* nombre del meta-nodo */
}
```

(a)



(b)

Figura 5-9: Representación de un meta-nodo.

CAPITULO 6

Evaluación de Resultados

6.1 Introducción.

Los objetivos de este capítulo son:

- Dar una visión de las perspectivas de desarrollo y utilidad de los editores orientados a los lenguajes para microcomputadoras.
- Transmitir la experiencia obtenida en el desarrollo de este trabajo de tesis a través de analizar los aciertos y los errores cometidos en la implantación del EDOLEM.

Considerando estos objetivos, la evaluación que se hará consiste de tres puntos:

- Analizaremos las características de la interfaz con el usuario.
- Se hará una comparacion entre un editor dirigido por

sintaxis y un editor de texto.

- Analizaremos las características del equipo de cómputo utilizado, asi como las estrategias de diseño e implantación que se llevaron a cabo en este trabajo.

6.2 Interfaz con el Usuario.

Como se mencionó en el capítulo 3, un EDOLEM está constituido por dos tipos de comandos: comandos de construcción y comandos de edición; donde los comandos de construcción dependen del lenguaje para el cual se generó el EDOLEM, mientras que los comandos de edición son comunes a todos los EDQLEM.

6.2.1 Comandos de Construcción.

Uno de los aspectos que mas impacta en la utilización de un sistema generatriz es la facilidad que éste proporcione al implantador en el proceso de generación. En este sentido el EDOLEM presenta dos aspectos muy importantes; primero, dado que la gramática puede expresarse en términos de sí misma, se puede generar un EDOLEM para crear o modificar la descripción de gramáticas, con lo cual no existe la posibilidad de producir gramáticas incorrectas; segundo, en la descripción de la gramática se cuenta con un campo que le permite al implantador definir un sinónimo asociado al operador.

Los comandos de construcción los proporciona el implantador de un EDOLEM para un lenguaje específico, por lo tanto, el nombre de los comandos de construcción corresponde a los nombres de los operadores del lenguaje definidos por el

implantador.

En algunos editores dirigidos por sintaxis, tales como el sistema SYNTHESIZER [TEITELBAUM 81] o el sistema SUPPORT [ZELKOWITZ 84], la edición de los programas se realiza en forma híbrida; por ejemplo, en el sistema SYNTHESIZER [TEITELBAUM 81], las expresiones se insertan como texto y el editor realiza el análisis sintáctico del texto, mientras que el resto del programa se inserta en forma constructiva; por otro lado, en el sistema SUPPORT [ZELKOWITZ 84], se tienen dos modos de operación (que podríamos llamar modo de construcción y modo texto) y el usuario puede cambiar el modo de operación.

En el EDOLEM la aplicación de los comandos de construcción pone el énfasis en la estructura del programa que se está editando, es decir, se tiene una edición estructurada a todos los niveles, incluyendo las expresiones. En cuanto a la facilidad de editar programas en forma textual al nivel de las expresiones (asumiendo ésta como la forma natural) en lugar de editar en forma estructurada, es un aspecto que está todavía en proceso de experimentación; es decir, uno de los objetivos de un editor estructurado es investigar el impacto de esta forma de edición en el desarrollo de programas, considerando además que al EDOLEM siempre se le podrá crear un analizador sintáctico para las expresiones, si éste fuera necesario.

6.2.2 Comandos de Edición.

En el EDOLEM se cuenta con un conjunto de comandos de edición que le permiten al usuario manipular su programa; en el apéndice A se proporciona una lista completa de estos comandos y se explica cada uno de ellos. Dos comandos muy importantes son: .CLIP y .INSERT; el comando .CLIP le permite al usuario borrar un subárbol y almacenar éste en un área aparte para posteriormente insertarlo en otra parte del programa por medio del comando .INSERT; por ejemplo, en la figura 6-1.a nos encontramos en el nodo ":=", en el cual se aplica el comando .CLIP (figura 6-1.b), posteriormente se posiciona el cursor en el nodo "<statement>" (figura 6-1.c) para finalmente aplicar el comando .INSERT (figura 6-1.d).

Por otro lado, existen algunas características de gran importancia en una buena interfaz, que el EDOLEM no tiene, tales como la transformación de un operador a otro o el cambio de precedencia en la evaluación de una expresión. Por ejemplo, en la figura 6-2.a se desea cambiar el operador IF por el operador IF-ELSE, como se muestra en la figura 6-2.b; en el EDOLEM esta transformación no se puede realizar directamente. Para realizar esta transformación el usuario tendrá primero que almacenar el subárbol ASIGNA, posteriormente borrar el subárbol IF, aplicar el operador IF-ELSE y finalmente insertar el subárbol ASIGNA. De la misma forma, en la figura 6-3.a si el usuario desea cambiar la precedencia y realizar primero la suma y despues la

```
BEGIN
    w := x;
    <estado>;
    while i > 0 DO
        BEGIN
            i := i DIV 2;
            w := SQR(w);
        END
END
```

a).- Programa original.

```
BEGIN
    w := x;
    <estado>;
    while i > 0 DO
        BEGIN
            i := i DIV 2;
        END
END
```

b).- Aplicación de .CLIP

```
BEGIN
    w := x;
    <estado>;
    while i > 0 DO
        BEGIN
            i := i DIV 2;
        END
END
```

c).- Movimiento al lugar de inserción.

```
BEGIN
    w := x;
    w := SQR(w);
    while i > 0 DO
        BEGIN
            i := i DIV 2;
        END
END
```

d).- Aplicación de .INSERT

Figura 6-1: Aplicación de los comandos .clip y .insert.

multiplicación, tendría que guardar el subárbol MULTIPLICA, borrar el subárbol SUMA, insertar el subárbol MULTIPLICA

borrar el nodo z y aplicar el operador SUMA, o simplemente borrar el subárbol SUMA e insertar nuevamente la expresión para obtener el resultado mostrado en la figura 6-3.b.

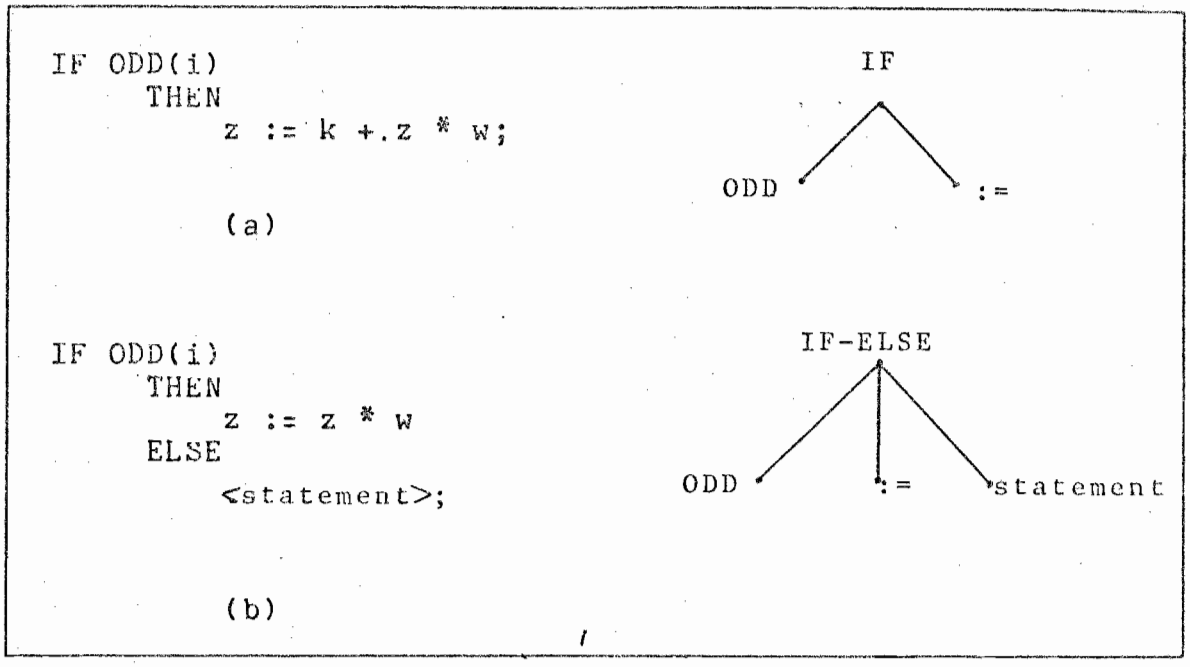


Figura 6-2: Tranformacion de un operador a otro operador.

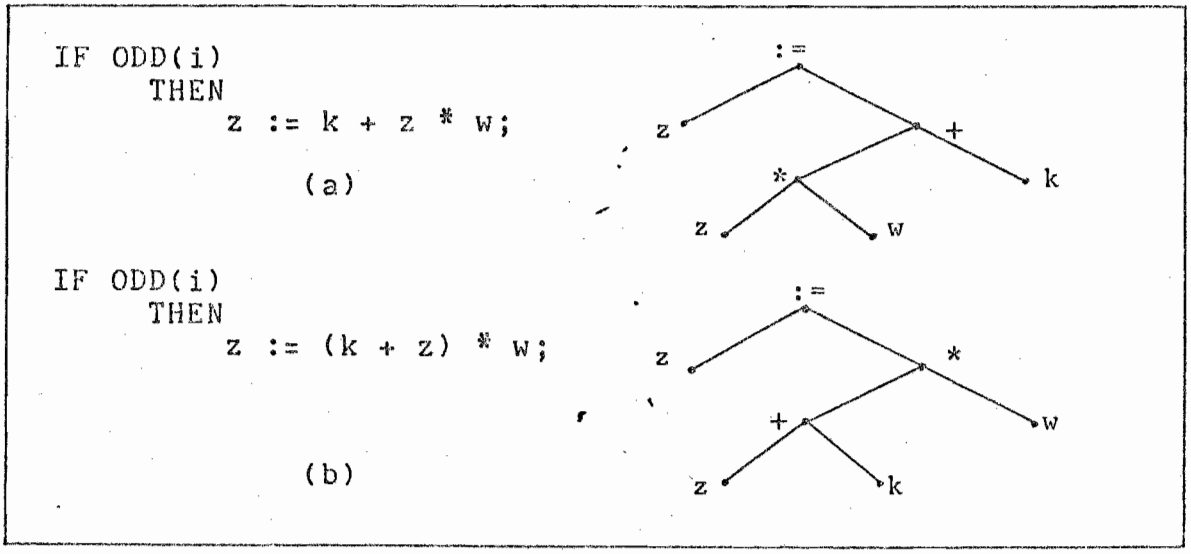


Figura 6-3: Cambio de la precedencia de un operador .

Este tipo de operaciones generalmente son mas fáciles

de realizar con un editor de texto, sin embargo, para el caso de transformar un operador a otro operador, dependiendo de la profundidad del árbol sintáctico, el realizar la modificación con un editor de texto puede ocasionar que el usuario cometa algun error sintáctico. En el caso del cambio de precedencia, en un editor de texto es necesario pensar en términos de inserción de paréntesis y no en términos de precedencia. Como corolario podemos concluir que lo que le hace falta a esta versión del EDOLEM es riqueza de operaciones, no de conceptos.

6.3 Un EDOLEM Versus un Editor de Texto.

No se puede realizar una comparación directa entre el EDOLEM y un editor de texto, dado que el objetivo de cada uno de ellos es diferente, es decir; en un editor de texto el contenido o significado de lo que se está editando no tiene ninguna importancia, este tipo de editores solo entiende y manipula caracteres y en algunos casos, comandos de formateo de texto. Por otro lado, el enfoque fundamental del EDOLEM es la manipulación de estructuras sintácticas, sin importar la forma de despliegue de dicha estructura. La comparación por lo tanto se tiene que realizar entre el EDOLEM y un medio ambiente de edición de texto para la producción de programas.

En el caso que nos ocupa, la comparación es en términos del tiempo necesario para producir un programa con cada uno de los sistemas mencionados, del tamaño del archivo generado y del ciclo de compilación. Por lo que respecta al tiempo de producción de un programa, éste está íntimamente

relacionado con el ciclo de compilación; es decir, en un medio ambiente de edición de texto gran parte del tiempo para producir un programa transcurre entre la edición de texto para modificar los errores sintácticos y la compilación, mientras que en el EDOLEM, dado que no es posible cometer errores sintácticos, el ciclo de edición y compilación solo se lleva a cabo para la corrección de errores semánticos, siendo ademas posible desarrollar chequeos semánticos para el EDOLEM. Finalmente, en cuanto al tamaño del archivo generado, dado que el EDOLEM manipula un arbol sintáctico y que en la definición de la gramática se incluye los esquemas de despliegue, generalmente el EDOLEM ocupará menos espacio.

6.4 Características del Equipo.

En esta sección analizaremos las características del equipo de cómputo utilizado en la implantación del EDOLEM. La microcomputadora utilizada es una CROMENCO SYSTEM ZERO; la cual está constituída por un microprocesador Z-80(8 bits), un espacio de memoria principal de 64 KBytes, dos unidades de disco flexible de 386 KBytes cada una y una terminal con pantalla de 24 renglones por 80 columnas.

El sistema operativo de la microcomputadora utilizada es CDOS, el cual es un sistema operativo tipo CP/M que se encuentra residiendo permanentemente en memoria; este sistema ocupa un tamaño de memoria de 15 KBytes, dejando disponibles solamente 49 KBytes. El manejo de la trasferencia de información entre la memoria principal y la memoria secundaria

se lleva a cabo a través de un área de memoria de 128 Bytes, correspondiendo este tamaño a un sector de disco. Por último, CDOS es un sistema operativo monousuario que no contempla el manejo de memoria virtual. Bajo estas consideraciones se llevo a cabo un análisis para incluir en la implantación del EDOLEM un esquema de manejo de memoria, cuyo análisis se presenta en el capítulo 4 y cuyos resultados se evalúan en la sección 6.6.

La manipulación de la pantalla se puede realizar a través de comandos de control que permiten: posicionar el cursor en cualquier parte de la pantalla (considerándola como un espacio bidimensional de 24 x 80 posiciones) y poner áreas de la pantalla en modo inverso.

Considerando que el EDOLEM se implantara en una microcomputadora diferente (por ejemplo una PC IBM compatible), con un espacio de memoria principal de 128 Kbytes (como mínimo), con el sistema operativo MS-DOS, los problemas antes mencionados no desaparecen pero se atenúan. Por ejemplo, considerando un espacio de memoria de 128 Kbytes y que el sistema operativo puede ocupar un espacio de 20 Kbytes, el espacio disponible para el EDOLEM sería de 108 Kbytes (120 % mayor que en una microcomputadora de 64 Kbytes). Dado que la generación de código no es uno a uno entre una microcomputadora de 8 bits y una de 16 bits, produciéndose en esta última mas código, el tamaño del EDOLEM[10] aumentaría, por

---

[10] nos referimos al programa ejecutable

lo cual posiblemente se tendría que manejar "overlays"; sin embargo, se tendría un área para el usuario mucho mayor, resultando en menores accesos a disco en el manejo de memoria virtual. Por otro lado, si además de las características antes mencionadas se cuenta con un disco duro, la velocidad de transferencia entre disco y memoria se incrementa en 51% (como se muestra en el capítulo 4).

6.5 Estrategias de Diseño y de Implantación.

Las estrategias de diseño e implantación del EDOLEM, se establecieron a partir de las características físicas del equipo utilizado así como de tres objetivos:

Transportabilidad

En este concepto se enmarca el hecho de hacer al EDOLEM lo menos dependiente posible de las características físicas de la máquina en que se realizó la implantación considerando la transportación posterior del sistema a otras microcomputadoras.

Optimización del espacio de memoria

Dado que el espacio de memoria con que se cuenta es pequeño, se realizó una evaluación de los requerimientos de memoria y de las diferentes alternativas para el manejo de `esta.

Interfaz con el usuario

Se implantaron un conjunto básico de comandos de edición que permiten una interacción cómoda y flexible entre el usuario y el sistema; sin embargo, no se implantaron algunos otros comandos de gran importancia (seccion 6.2.2).

6.6 Conclusiones.

Como corolario de este trabajo, expondremos los resultados obtenidos en este trabajo de tesis, los cuales están divididos en 4 aspectos: estado actual del sistema, manejo de memoria, interfaz con el usuario y, transportabilidad.

- Estado actual del sistema.- Actualmente se tiene la primera versión del EDOLEM disponible para su utilización, tanto en el aspecto de generación como en el aspecto de edición.

- Manejo de memoria.- Como se describe en el capítulo 4, el manejo de memoria consta de tres aspectos:

  * Manejo del área de usuario (memoria virtual).- En este aspecto, las pruebas realizadas demuestran una forma de operación adecuada.

  * Manejo de las tablas del lenguaje.- Como se explicó en la sección 4.3, se realizó un análisis de las características de las tablas del lenguaje así como del espacio de memoria disponible para su manejo; reservándose finalmente un espacio de 12.5 Kbytes para este efecto. En esta versión del EDOLEM se generó el sistema para el lenguaje de programación Pascal. La gramática de este lenguaje ocupa el 80 % del espacio reservado para el manejo de las tablas: Como conclusión, en esta primera versión del EDOLEM se puede generar un editor para un lenguaje cuya gramática sea a lo mas 20% mas grande que la gramática de Pascal.

  * Manejo de "overlays".- Las pruebas realizadas para probar la funcionalidad de este manejo desmostraron un funcionamiento adecuado. Sin embargo, cuando se aplica contantemente comandos de edición el retardo en la respuesta aumenta, por lo cual en versiones posteriores habría que considerar la posibilidad de optimizar este esquema o transportarlo a MS-DOS.

- Interfaz con el usuario.- El conjunto básico de comandos implementados en esta versión le ofrecen al usuario una forma cómoda de interacción con el

sistema, aunque este podría mejorarse aumentado estos comandos (como se menciona en la sección 6.2.2).

- Transportabilidad.- Para alcanzar este objetivo se establecieron módulos lógicos de operación, identificándose los aspectos dependientes de las características físicas del equipo utilizado: manejo de la pantalla (3 rutinas del módulo de entrada/salida), manejo de memoria virtual (manejo de direcciones de 16 bits, 2 rutinas del modulo de manejo de memoria), manejo de "overlays" (se utilizan direcciones físicas). Finalmente, la implantación del sistema se realizó en el lenguaje de programación C[Ritchie], por lo cual también habría que considerar las diferencias entre el compilador utilizado y el compilador de la máquina a que se quiera transportar el EDOLEM.

APENDICE A

Comandos de Edición

Como se mencionó en el capítulo 2, los comandos de edición son comunes a todos los EDOLEM, es decir, son independientes de la gramática. Estos se pueden aplicar por medio del nombre o por medio de su sinónimo; en caso de utilizar el nombre éste debe ir precedido de un punto "." , en el caso de utilizar el sinónimo éste se aplica directamente. Por ejemplo, si se quiere pedir ayuda se puede aplicar el comando ".HELP" o "^x"(donde ^x significa CONTROL-x). En esta versión se cuenta con cinco tipos de comandos de edición: comandos de movimiento del cursor, comandos de búsqueda, comandos de manejo de árboles, comandos de ayuda y otros.

A.1 Comandos de Movimiento del Cursor.

↓ (^N)

Mueve el cursor al primer descendiente legal del nodo actual, de acuerdo con el esquema de despliegue. Si se encuentra en un nodo terminal se mueve al siguiente descendiente legal del padre del nodo actual; si no existe otro descendiente, se mueve al hermano del padre del nodo actual en forma recursiva, finalmente, si este nodo es el último en el árbol(definido en pre-orden), el comando no tienen ningún efecto.

↑ (^O)

Mueve el cursor al padre del nodo actual.

→ (^X^N)

Mueve el cursor al siguiente descendiente del padre del nodo actual de acuerdo con el esquema de despliegue, si no existe otro descendiente se mueve al hermano del padre del nodo actual recursivamente. Si el nodo actual es el último en el árbol(definido en pre-orden), el comando no tiene ningún efecto.

← (^X^B)

Mueve el cursor al hermano anterior del nodo actual de acuerdo con el esquema de despliegue, si no existe, se mueve al hermano anterior del padre del nodo actual recursivamente. Si el nodo actual es el descendiente más a la izquierda del árbol(es decir) es el último nodo del árbol si se definiera éste en post-orden), el comando no tiene ningún efecto.

.HOME (^X^H)

Mueve el cursor a la raíz del árbol.

.BACK (^B)

Si el comando anterior fue un movimiento del cursor, regresa éste a la posición anterior, en caso contrario no tiene ningún efecto.

A.2 Comandos de Busqueda.

.FIND (^F)

Busca en el árbol una variable o una constante hacia adelante.

.RFIND (^XB)

Busca en el árbol una variable o una constante hacia atrás.

A.3 Comandos de Manejo de arboles.

.APPEND (^X^E)

Si el nodo actual es un elemento de una lista, inserta un meta-nodo al final de la lista, en caso contrario el comando no tiene ningún efecto.

.PREPEND (^X^A)

Si el nodo actual es un elemento de una lista, inserta un meta-nodo al inicio de la lista, en caso contrario el comando no tiene ningún efecto.

.EXTEND (^E)

Este comando inserta un meta-nodo en una lista. Si el nodo actual es un nodo lista, inserta el meta-nodo al principio de la lista; si el nodo actual es un miembro de una lista, inserta el meta-nodo inmediatamente después del nodo actual.

.BEXTEND (^XE)

Este comando inserta un meta-nodo en una lista. Si el nodo actual es un elemento de una lista, inserta el meta-nodo en la posición inmediatamente anterior del nodo actual, en caso contrario el comando no tiene ningún efecto.

.DELETE (^D)

Borra un subárbol. Si el padre del subárbol es un nodo de orden fijo, inserta un meta-nodo en el lugar del subárbol borrado. Si el padre del subárbol borrado es un nodo lista no se crea ningún meta-nodo.

A.4 Comandos de Ayuda.

.HELP (^X?)

Este es un comando de ayuda. Si el nodo actual es un meta-nodo despliega los comandos de construcción cuya aplicación es legal en dicho meta-nodo, en caso contrario despliega los comandos de edición.

.? (.?)

Este es otro comando de ayuda y despliega en forma incondicional los comandos de edición.

A.5 Otros Comandos.

.DISPLAY (^L)

Este comando redespliega la pantalla.

.MODE (^XM)

Este comando modifica el modo de operación del EDOLEM. En esta versión se cuenta con dos modos de operación: principiante y avanzado. Al aplicarse el comando, éste pide al usuario que le defina el modo, para lo cual tiene que teclear el caracter "p" para principiante y "a" para avanzado; en caso de ser otro el carácter tecleado el comando no tiene efecto.

.QUIT (^XS)

Salva el árbol editado en un archivo árbol y además crea y salva un archivo de texto con el mismo nombre que el archivo árbol, pero con diferente extensión.

.CANCEL (^Q)

Este comando le advierte al usuario que el árbol se ha modificado y le pregunta si en realida quiere cancelar la sesión, en caso afirmativo, cancela dicha sesión sin salvar el árbol.

APENDICE B

Comandos del Esquema

de Despliegue

Cada operador terminal o no-terminal debe tener
ociado un esquema de despliegue que proporciona la
scripción del texto a utilizar, los descendientes(si los
y) del operador y el formato de despliegue. El texto dado
el esquema de despliegue se despliega literalmente, solo
s secuencias que empiezan con "@" se tratan de manera
ferente. Los comandos de formateo para los esquemas de
spliegue utilizados en los ALOES [MEDINA-MORA 81b] y que se
ilizaran aquí son:

@n	Inserta una nueva línea en el despliegue.
@t o @>	Inserta un tabulador en el despliegue.
@<	Se regresa un tabulador (se para en el inicio de línea).
@+	Incrementa el nivel de indentación (tiene efecto en el siguiente "@n"). Todas las indentaciones son de cuatro espacios.
@-	Decrementa el nivel de indentación (tiene efecto en el siguiente cambio de línea -"@n").
@l	Se regresa al margen izquierdo de la línea actual.
@h	Se regresa un carácter.
@b	Se regresa a la línea anterior.
@p<n>	Realiza una entrada en el stack de marcas. Las marcas se utilizan para "recordar" las posiciones de columnas para el formateo. Esto es muy útil cuando el formateo deseado depende del tamaño del identificador.
@r<n>	Saca la marca del stack de marcas.
@g<n>	Trae la marca del stack de marcas, pero no la saca. Mueve el cursor de despliegue a la columna especificada por la marca.
@@	Despliega el carácter "@".

La forma en que se despliegan los operadores terminales y no-terminales es diferente, para los operadores terminales los comandos de despliegue son:

@c	Despliega el valor de una constante o un texto.
@s	Despliega el nombre de una variable, tomándolo de la tabla de símbolos.

Para los operadores no-terminales los comandos de despliegue son:

@<n>	Despliega el n-ésimo descendiente

recursivamente. Este comando es aplicable solo para operadores de orden fijo, donde los descendientes se numeran de uno en adelante.

<pr>@O<t>[@q<po>][@e<s>]

Recorre los operadores listo para su despliegue. La "@O" indica que cada elemento de la lista se recorre en orden, "<pr>" es la cadena de caracteres que se imprime antes de la lista, la cadena "<t>" se utiliza para separar la lista de elementos, ésta se termina al encontrar "@q" o el fin del esquema de despliegue. La cadena "<po>" se imprime después de recorrer la lista. La "@e" opcional, indica cómo se debe hacer el despliegue de la lista si ésta es una lista vacía. Por ejemplo, en la figura B-1.a se muestra el esquema de despliegue especificado en la gramática de Pascal (que se presenta en el apéndice D) para el operador COMPOUND. En la figura B-1.a se muestra el esquema de despliegue; donde "@+" incrementa el nivel de indentación, "@n" inserta un cambio de línea en el despliegue, "@O" indica que la lista se despliega en orden, ";" separa los elementos de la lista, "@-" decrementa el nivel de indentación y las palabras BEGIN y END se despliegan literalmente. En la figura B-1.b se muestra el despliegue cuando el operador tiene uno o más descendientes, mientras que en la figura B-1.c se muestra el despliegue cuando este operador no tiene descendientes.

```
"BEGIN@+@n@O;@n@q@-@nEND@eBEGIN@nEND"

        (a)


    BEGIN
        <statement>;
            :
            :
        <statement>;
    END

        (b)


    BEGIN
    END


        (c)
```

Figura B-1: Ejemplo de despliegue de un nodo no-terminal de orden variable.

APENDICE C

Como Construir y Utilizar un Editor

El proceso de construcción y utilización de un editor esta constituido por dos fases generación y edición (figura C-1). La fase de generación a su vez se divide en dos pasos: descripción de la gramática y procesamiento de la gramática.

C.1 Fase de Generación.

El primer paso para construir un editor para una estructura dada (ej. un lenguaje de programación), es definir la gramática de dicha estructura. Para esta definición se establece un formato estandar, como se mencionó en el capítulo

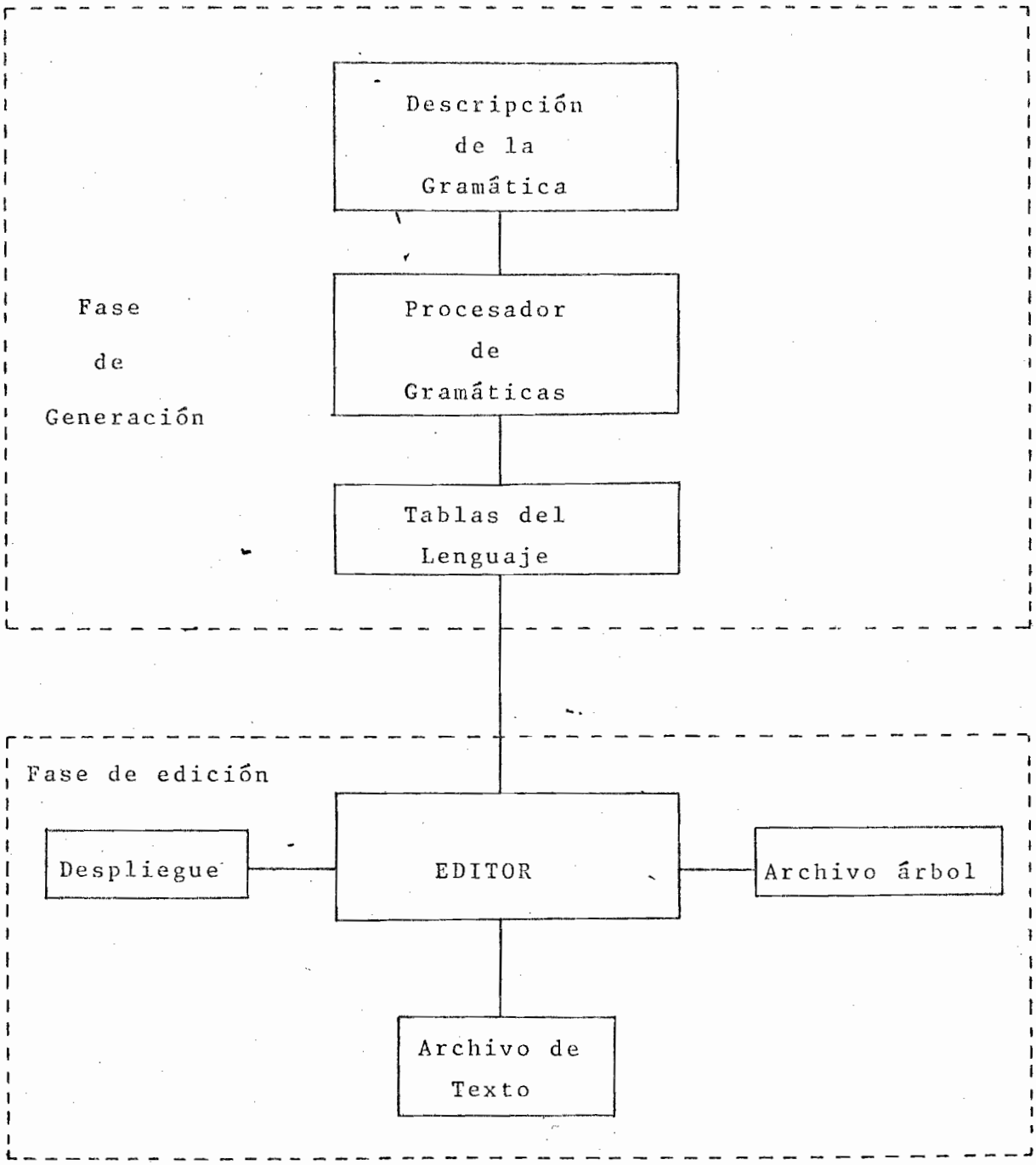


Figura C-1: Fases de un EDOLEM.

3. La descripción de la gramática está constituida por tres primitivas, estas son: operadores terminales, operadores no-terminales y operadores clase.

Como primera parte de la descripción de la gramática se debe proporcionar el nombre de ésta. Posteriormente se pasa a la descripción de los operadores terminales. La descripción de los operadores terminales se componen de 5 campos:

- Nombre.- Este es el identificador del operador y es el nombre del comando de construcción que el usuario utiliza.
- Tipo.- Este campo le indica al procesador de gramáticas el tipo de operador, el cual puede ser:
  * Variable.- Este tipo se utiliza para la manipulación de las variables del usuario.
  * Constante.- Este tipo se utiliza para el manejo de caracteres ASCII, tales como comentarios.
  * Estático.- Este tipo se utiliza para representar palabras reservadas del lenguaje, tales como la definición de tipos de variables.
- Rutinas de accion.- En esta versión no se tienen disponibles pero se debe incluir el campo en la descripción de la gramática.
- Sinónimo.- Este es un nombre alternativo a ser utilizado por el usuario y debe ser breve.
- Esquema de despliegue.- Este campo se utiliza para la interfaz entre el sistema y el usuario, esto es, nos indica cual es el texto a desplegar.

Por ejemplo, en la figura C-2 se muestra la descripción de los operadores terminales para una parte de la gramática de pascal; en donde se puede ver el operador IDENT,

```
/* Subconjunto de Pascal estandar          18.12.84 */

{subpas.v1}

/* Operadores terminales */

{
IDENT           = {v}
                  |
                  | "-"
                  | "@s";
IDENTFILE       = {v}
                  |
                  | "-"
                  | "@s";
INTEGER         = {s}
                  |
                  | ""
                  | "INTEGER";
REAL            = {s}
                  |
                  | ""
                  | "REAL";
BOOLEAN         = {s}
                  |
                  | ""
                  | "BOOLEAN";
CHAR            = {s}
                  |
                  | "",
                  | "CHAR";
TRUE            = {s}
                  |
                  | ""
                  | "TRUE";
FALSE           = {s}
                  |
                  | ""
                  | "FALSE";}
```

Figura C-2: Descripción de los operadores Terminales.

el cual es del tipo variable, no tiene rutina de acción asociada, no tiene sinónimo y su esquema de despliegue es ês (despliega el nombre del identificador) o el operador REAL, el cual es del tipo estatico, no tiene rutina de accion asociada, no tiene sinónimo y su esquema de despliegue es la palabra REAL.

El siguiente módulo de la descripción de la gramática es la descripción de los operadores no-terminales, la cual se componen de 6 campos:

- Nombre.- Este es el identificador del operador y es el nombre del comando de construcción que el usuario utiliza.
- Descendientes.- en este campo se tiene los nombres de las clases a las cuales debe pertenecer cada descendiente.
- Precedencia.- Se utiliza para proporcionar los paréntesis en forma automática. Esto es con fines del despliegue, para no inducir al usuario a confusión, ya que el árbol sintáctico interno nunca es ambiguo.
- Rutinas de acción.- En esta versión no se tienen disponibles pero se debe incluir el campo en la descripción de la gramática.
- Sinónimo.- Este es un nombre alternativo a ser utilizado por el usuario y debe ser breve.
- Esquema de despliegue.- Este campo se utiliza para la interfaz entre el sistema y el usuario, esto es, nos indica cual es el texto a desplegar.

Por ejemplo,en las figuras C-3 y C-4 se muestra la descripción de los operadores no-terminales para una parte de la gramática de Pascal; en donde se puede ver el operador ASIGNA, el cual tiene dos descendientes que deben pertenecer a

la clase "primario" y "expression" respectivamente, no tiene rutina de acción asociada, tiene la máxima precedencia (por lo cual no se especifica), su sinónimo es ":=" y su esquema de despliegue es "@1 := @2" (despliega el primer y segundo descendiente recursivamente); el operador SUMA, el cual tiene dos descendientes que deben pertenecer a la clase "expression", su precedencia es 11, no tiene rutina de acción asociada, su sinónimo es "+" y su esquema de despliegue es "@1 + @2" (despliega el primer y segundo descendiente recursivamente).

Por último tenemos la descripción de los operadores clase, los que estan conformados por dos campos:

- Nombre.- Este es el identificador de la clase y es el nombre que se despliega en un meta-nodo; para indicarle al usuario a que clase debe pertener el operador a aplicar.
- Descendientes.- En este campo se tiene los nombres de los operadores (terminales o no-terminales) cuya aplicación es legal en la clase correspondiente, es decir los operadores que pertenecen a esta clase.

En la figura C-5 se muestran algunos operadores clase de pascal, donde puede observarse por ejemplo el operador "expression", al cual pertenecen los operadores no-terminales EQUAL, NOTEQUAL, ADD, SUBTRACT, MULTIPLY, DIVIDE, AND, OR y los operadores terminales IDENT, TRUE y FALSE.

Finalmente, para completar el proceso de generacion, se llama al procesador de gramáticas, el cual recibe como entrada la descripción de la gramática y genera como salida un

```
/* Operadores no-terminales */
{
PROGRAM          = proghead decl compoundstate
                 |
                 |
                 | ""
                 | "@1@n@2:@n@3.";
PROGHEAD         = ident fileidents
                 |
                 |
                 | ""
                 | "PROGRAM @1@2:@n";
FILEIDENTS       = fileident
                 |
                 |
                 | ""
                 | "(@0, @q)";
DECLS            = decl
                 |
                 |
                 | ""
                 | "@0:@n@n@q@e";
CONSTDEFS        = constdef
                 |
                 |
                 | ""
                 | "CONST@+@n@0:@n@q@-";
CONSTDEF         = ident expression
                 |
                 |
                 | ""
                 | "@1 = @2";
VARDECLS         = vardecl
                 |
                 |
                 | ""
                 | "VAR@+@n@0:@n@q@-";
VARDECL          = idents type
                 |
                 |
                 | ""
                 | "@1 : @2";
IDENTS           = ident
                 |
                 |
                 | ""
                 | "@0, ";
ASSIGN           = leftside expression
                 |
                 |
                 | ":="
                 | "@1:= @2";
```

Figura C-3: Descripci´on de los operadores No-terminales.

```
ADD             = expression expression
                | (11)
                |
                | "+"
                | "@1 + @2";
SUBTRACT        = expression expression
                | (11)
                |
                | "-"
                | "@1 - @2;
MULTIPLY        = expression expression
                | (12)
                |
                | "*"
                | "@1 * @2";
DIVIDE          = expression expression
                | (12)
                |
                | "/"
                | "@1 / @2";
AND             = expression expression
                | (12)
                |
                | ""
                | "@1 AND @2";
OR              = expression expression
                | (11)
                |
                | ""
                | "@1 OR @2";
EQUAL           = expression expression
                | (10)
                |
                | "="
                | "@1 = @2";
COMPOUND        = statement
                |
                |
                | ""
                | "BEGIN@+@n@0:@n@q@-@nEND@eBEGIN@nEND";
IF              = expression statement
                |
                |
                | ""
                | "IF @1@+@nTHEN@+@n@2@-@-";
IFELSE          = expression statement statement
                |
                |
                | ""
                | "IF @1@+@nTHEN@+@n@2@-@nELSE@+@n@3@-@-";}
```

Figura C-4: Descripci´on de los operadores No-terminales.

```
/* operadores clase */

{

proghead          = PROGHEAD;
ident             = IDENT;
fileidents        = FILEIDENTS;
fileident         = IDENTFILE;
decls             = DECLS;
decl              = CONSTDEF TYPEDEF VARDECLS;
constdef          = CONSTDEF;
type              = INTEGER REAL BOOLEAN CHAR;
vardecl           = VARDECL;
idents            = IDENTS;
compound          = COMPOUND;
statement         = ASSIGN COMPOUND IF IFELSE;
leftside          = IDENT;
expression        = EQUAL ADD SUBTRACT MULTYPLY DIVIDE
                    AND OR IDENT TRUE FALSE;}
```

Figura C-5: Descripción de los operadores clase.

archivo con las tablas del lenguaje correspondientes, éste archivo tendrá el nombre indicado en la descripción de la gramática como nombre del lenguaje, que para nuestro ejemplo es SUBPAS.V1. En este paso el procesador de gramáticas verifica que la descripción de la gramatica sea sintácticamente correcta, si detecta algun error, lo marca y no genera las tablas correspondientes.

C.2 Fase de Edición.

Esta es la fase de utilización del EDOLEM para crear y modificar programas. La creación y modificación de programas o estructuras editadas con el EDOLEM para una gramática determinada se realiza a traves de invocar al EDOLEM, especificandole explícitamente para que gramática se quiere editar. Por ejemplo, utilizando la gramática que se muestra en el apéndice D, la invocación para crear un programa se realizaría como se muestra a continuación:

EDOLEM SUBPAS.V1 -l program<CR>

donde SUBPAS.V1 es el nombre del archivo donde estan las tablas del lenguaje obtenidas del proceso de generacion, "-l" le indica al EDOLEM que el nombre que sigue, es el nombre del operador raíz, que en este caso es "program". En la figura C-6 se muestra el despliegue que se obtiene después de esta invocación del EDOLEM. Después de aplicar una serie de comandos (que no se muestran aqui), nos encontramos en la construcción del programa mostrado en la figura C-7, donde aplicamos el operador IF-ELSE, para obtener la construcción mostrada en la figura C-8.

Para la modificación de un programa, la invocación es:

EDOLEM SUBPAS.V1 -l program prueba.tr<CR>

donde prueba.tr es el archivo árbol que debe cargar

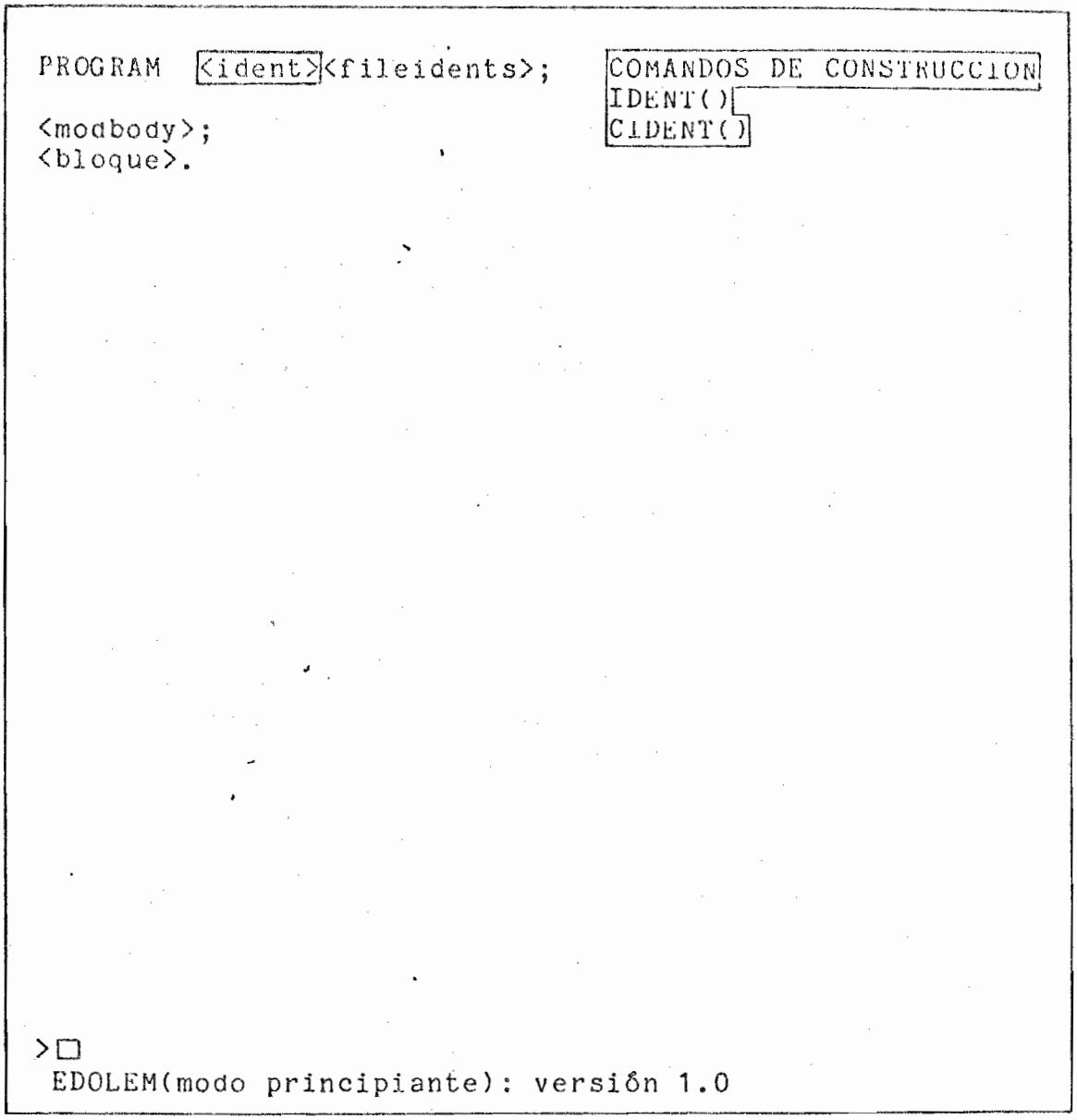


Figura C-6 : Ejemplo de una sesión con el EDOLEM.

```
PROGRAM  texto(input,output);

EXPORTS
    VAR
        gut : BOOLEAN;
PRIVATE
    VAR
        i,j,k : INTEGER;
BEGIN

    <estado>
END.
```

```
COMANDOS DE CONSTRUCCION
ASIGNA(:=)
PROCEDURE()
GOTO()
EMPTY()
BLOQUE()
IF()
IF-ELSE()
CASE()
WHILE()
REPEAT()
FOR()
LABEL()
EXIT()
```

```
>□
 EDOLEM(modo principiante): versión 1.0
```

Figura C-7 : Ejemplo de una sesión con el EDOLEM.

el EDOLEM. Considerando que el programa que queremos modificar fuera el mostrado en la figura C-8 , despues de la invocación del EDOLEM es despliegue seria el mismo que se muestra en la figura C-8 .

Apéndice D

Descripción de la

Gramática de Pascal

En este apéndice se muestra la descripción de la gramática de Pascal utilizada para probar el EDOLEM.

```
/* gramática de Pascal                    18.12.83  */
{pascal.v1}
/* operadores terminales */
{
LONGCOMMENT     = {t}
                  |
                  | "{"
                  | "(* @c *)";
IDENT           = {v}
                  |
                  | ""
                  | "@s";
```

```
IDENTFILE       = {v}
                  |
                  | ""
                  | "@s";
IDENTC          = {v}
                  |
                  | ""
                  | "@s";
IDENTT          = {v}
                  |
                  | ""
                  | "@s";
IDENTFIELD      = {v}
                  |
                  | ""
                  | "@s";
IDENTV          = {v}
                  |
                  | ""
                  | "@s";
IDENTFUNCT      = {v}
                  |
                  | ""
                  | "@s";
IDENTP          = {v}
                  |
                  | ""
                  | "@s";
IDENTF          = {v}
                  |
                  | ""
                  | "@s";
IDENTMOD        = {v}
                  |
                  | ""
                  | "@s";
IDENTPROCFUNCT  = {v}
                  |
                  | ""
                  | "@s";
INTCONST        = {c}
                  |
                  | "0"
                  | "@c";
REALCONST       = {c}
                  |
                  | ""
                  | "@c";
STRINGCONST     = {a}
                  |
                  | "\``"
                  | "`@c`";
```

```
CHARCONST       = {c}
                  |
                  | "``"
                  | "`@c`";
NIL             = {s}
                  |
                  | ""
                  | "NIL";
EMPTY           = {s}
                  |
                  | ""
                  | "";
PACKED          = {s}
                  |
                  | ""
                  | "PACKED ";
INTEGER         = {s}
                  |
                  | ""
                  | "INTEGER";
REAL            = {s}
                  |
                  | ""
                  | "REAL";
BOOLEAN         = {s}
                  |
                  | ""
                  | "BOOLEAN";
CHAR            = {s}
                  |
                  | ""
                  | "CHAR";
OTHERWISE       = {s}
                  |
                  | ""
                  | "OTHERWISE";
COMMENT         = {a}
                  |
                  | "{"
                  | "(* @c *)";
TRUE            = {s}
                  |
                  | ""
                  | "TRUE";
FALSE           = {s}
                  |
                  | ""
                  | "FALSE";}
```

```
/* operadores no-terminales */
{
COMPUNIT        = compunit
                |
                |
                | ""
                | "@1";
PROGRAM         = proghead modbody compound
                |
                |
                | ""
                | "@1@n@2:@n@3.";
PROGHEAD        = ident fileidents
                |
                |
                | ""
                | "PROGRAM @1@2:@n";
FILEIDENTS      = fileident
                |
                |
                | ""
                | "(@0, @q)";
MODULE          = modhead modbody
                |
                |
                | ""
                | "@1@n@2.";
MODHEAD         = ident
                |
                |
                | ""
                | "MODULE @1:@n";
MODBODY         = exports decls
                |
                |
                | ""
                | "@1PRIVATE@n@2";
EXPORTS         = expodecl
                |
                |
                | ""
                | "EXPORTS@+@n@0:@n@n@q:@-@n@e";
DECLS           = decl
                |
                |
                | ""
                | "@0:@n@n@q@e";
IMPORTS         = idetmod identf
                |
                |
                | ""
                | "IMPORTS @1 FROM @2 ";
```

```
LABELDECLS      = label
                |
                |
                | ""
                | "LABEL @0, ";
CONSTDEFS       = constdef
                |
                |
                | ""
                | "CONST@+@n@0:@n@q@-";
CONSTDEF        = ident expression
                |
                | ""
                | "@1 = @2";
TYPEDEFS        = typedef
                |
                |
                | ""
                | "TYPE@+@n@0:@n@q@-";
TYPEDEF         = ident type
                |
                |
                | ""
                | "@1 = @2";
SCALARTYPE      = ident
                |
                |
                | "("
                | "(@0, @q)";
SUBRANGETYPE    = constexp constexp
                |
                |
                | ".."
                | "@1 .. @2";
ARRAYTYPE       = packing indextypes type
                |
                | "["
                | "@1ARRAY[@2] OF @3";
INDEXTYPES      = simpletype
                |
                |
                | ""
                | "@0, ";
STRING          = maxlength
                |
                |
                | ""
                | "STRING@1";
```

```
MAXLENGTH       = intconst
                |
                |
                | ""
                | "[@1]";
RECORDTYPE      = packing fields
                |
                |
                | ""
                | "@+@n@1RECORD@+@n@2@-@nEND@-";
VARRECORDTYPE   = packing fixedpart variantpart
                |
                |
                | ""
                | "@+@n@1RECORD@+@n@2@3@-@nEND@-";
FIELDS          = field
                |
                |
                | ""
                | "@0:@n";
FIXEDPART       = field
                |
                |
                | ""
                | "@0:@n@q:@n";
FIELD           = fieldidents type
                |
                | ""
                | "@1 : @2";
FIELDIDENTS     = fieldident
                |
                |
                | ""
                | "@0, ";
VARIANTPART     = tagfield tident variants
                |
                |
                | ""
                | "CASE @1 @2 OF@+@n@3@-";
VARIANTS        = variant
                |
                |
                | ""
                | "@0:@n@q";
TAGFIELD        = fieldident
                |
                |
                | ""
                | "@1 :";
```

```
VARIANT         = caselabels fixedpart variantpart
                |
                |
                | ""
                | "@1 :@+@n(@2@3)@-";
CASELABELS      = constexp
                |
                |
                | ""
                | "@0,,";
SETTYPE         = packing simpletype
                |
                |
                | ""
                | "@1SET OF @2";
FILETYPE        = packing type
                |
                |
                | ""
                | "@1FILE OF @2";
POINTERTYPE     = tident
                |
                |
                | "^"
                | "^@1";
VARDECLS        = vardecl
                |
                |
                | ""
                | "VAR@+@n@0:@n@q@-";
VARDECL         = idents type
                |
                |
                | ""
                | "@1 : @2";
IDENTS          = ident
                |
                |
                | ""
                | "@0, ";
CIDENT          = ident comment
                |
                |
                | "{"
                | "@1@t@2";
FWDPROCDECL     = prochead
                |
                |
                | ""
                | "@1: FORWARD";
```

```
PROCDECL        = prochead block
                |
                |
                | ""
                | "@1:@+@n@2@-";
PROCHEAD        = ident formalparams
                |
                |
                | ""
                | "PROCEDURE @1@2";
FORMALPARAMS    = formalparam
                |
                |
                | ""
                | "@+@n(@0:@n @q)@-@e";
PARAMGROUP      = idents tident
                |
                |
                | ""
                | "@1 :@2";
VARPARAMGROUP   = idents tident
                |
                |
                | ""
                | "VAR @1 :@2";
FWDFUNCTDECL    = functhead
                |
                |
                | ""
                | "@1; FORWARD";
FUNCTDECL       = functhead block
                |
                |
                | ""
                | "@1:@+@n@2@-";
FUNCTHEAD       = ident formalparams restype
                |
                |
                | ""
                | "FUNCTION @1@2@3";
BLOCK           = decls compound
                |
                |
                | ""
                | "@1@n@2";
LABEL           = label statement
                |
                |
                | ""
                | "@<@1 : @2";
```

```
ASSIGN          = leftside expression
                  |
                  |
                  | ":="
                  | "@1:= @2";
INDEXEDVAR      = variable expressions
                  |
                  |
                  | "["
                  | "@1[@2]";
RECFIELD        = variable fieldident
                  |
                  |
                  | "."
                  | "@1.@2";
REFERENCEDVAR   = variable
                  |
                  |
                  | "^"
                  | "@1^";
FILEBUFFER      = variable
                  |
                  |
                  | ""
                  | "@1^";
EXPRESSIONS     = expression
                  |
                  |
                  | ""
                  | "@0, ";
EXPRESSION      = expression
                  |
                  |
                  | ""
                  | "@1";
PARENEXPR       = expression
                  | (13)
                  |
                  | ""
                  | "(@1)";
NOT             = expression
                  | (13)
                  |
                  | ""
                  | "NOT @1";
FUNCTDESIG      = functident actualparams
                  | (13)
                  |
                  | ""
                  | "@1@2";
```

```
ACTUALPARAMS    = expression
                |
                |
                | ""
                | "(@0, @q)@e";
ADD             = expression expression
                | (11)
                |
                | "+"
                | "@1 + @2";
SUBTRACT        = expression expression
                | (11)
                |
                | "-"
                | "@1 - @2";
OR              = expression expression
                | (11)
                |
                | ""
                | "@1 OR @2";
MULTIPLY        = expression expression
                | (12)
                |
                | "*"
                | "@1 * @2";
DIVIDE          = expression expression
                | (12)
                |
                | "/"
                | "@1 / @2";
INTDIVIDE       = expression expression
                | (12)
                |
                | ""
                | "@1 DIV @2";
MOD             = expression expression
                | (12)
                |
                | ""
                | "@1 MOD @2";
AND             = expression expression
                | (12)
                |
                | ""
                | "@1 AND @2";
EQUAL           = expression expression
                | (10)
                |
                | "="
                | "@1 = @2";
```

```
NOTEQUAL        = expression expression
                | (10)
                |
                | "<>"
                | "@1 <> @2";
LESSTHAN        = expression expression
                | (10)
                |
                | "<"
                | "@1 < @2";
LTEQL           = expression expression
                | (10)
                |
                | "<="
                | "@1 <= @2";
GREATERTHAN     = expression expression
                | (10)
                |
                | ">"
                | "@1 > @2";
GTEQL           = expression expression
                | (10)
                |
                | ">="
                | "@1 >= @2";
IN              = expression expression
                | (10)
                |
                | ""
                | "@1 IN @2";
SET             = element
                | (13)
                |
                | ""
                | "[@0, @q]";
EXPRPAIR        = expression expression
                |
                |
                | ".."
                | "@1..@2";
PROC            = pident actualparams
                |
                |
                | "("
                | "@1@2";
GOTO            = label
                |
                |
                | ""
                | "GOTO @1";
```

```
COMPOUND        = statement
                  |
                  |
                  | ""
                  | "BEGIN@+@n@0:@n@q@-@nEND@eBEGIN@nEND";
IF              = expression statement
                  |
                  |
                  | ""
                  | "IF @1@+@nTHEN@+@n@2@-@-";
IFC             = ident statement
                  |
                  |
                  | ""
                  | "{$IFC @1 THEN}@+@n@2@-@n{$ENDC}";
IFELSE          = expression statement statement
                  |
                  |
                  | ""
                  | "IF @1@+@nTHEN@+@n@2@-@nELSE@+@n@3@-@-";
CASE            = expression cases
                  |
                  |
                  | ""
                  | "CASE @1 OF@+@n@2@-@nEND";
CASES           = caseelement
                  |
                  |
                  | ""
                  | "@0:@n";
CASEELEMENT     = casestatlabs statement
                  |
                  |
                  | ""
                  | "@1 : @2";
CASESTATLABS    = casestatlab
                  |
                  |
                  | ""
                  | "@0, ";
SUBRANGE        = constexp constexp
                  |
                  |
                  | ".."
                  | "@1..@2";
WHILE           = expression statement
                  |
                  |
                  | ""
                  | "WHILE @1 DO @+@n@2@-";
```

```
REPEAT          = statements expression
                |
                |
                | ""
                | "REPEAT@+@n@1@-@nUNTIL @2";
STATEMENTS      = statement
                |
                |
                | ""
                | "@0:@n";
CSTATE          = comment statement
                |
                |
                | "{"
                | "@1@n@2";
FOR             = ident forlist statement
                |
                |
                | ""
                | "FOR @1:= @2 DO @+@n@3@-";
TO              = expression expression
                |
                |
                | ""
                | "@1 TO @2";
DOWNTO          = expression expression
                |
                |
                | ""
                | "@1 DOWNTO @2";
WITH            = recordvar statement
                |
                |
                | ""
                | "WITH @1 DO@+@n@2@-";
EXIT            = identprocfunct
                |
                |
                | ""
                | "EXIT(@1)";
REDORDVAR       = variable
                |
                |
                | ""
                | "@0, ";
CFIELD          = fieldidents type comment
                |
                |
                | "{"
                | "@1 : @2@t@3";
```

```
RESTYPE         = tident
                |
                |
                | ""
                | " : @1";
CTYPEDEF        = ident type comment
                |
                |
                | "("
                | "@1 = @2@t@3";
FIXED           = caselabels fields
                |
                |
                | ""
                | "@1 :@+@n(@2)@-";}

/* classes */
{
compunit        = PROGRAM MODULE;
modbody         = MODBODY;
exports         = EXPORTS;
proghead        = PROGHEAD;
modhead         = MODHEAD;
ident           = IDENT CIDENT;
fileidents      = FILEIDENTS;
fileident       = IDENTFILE;
identf          = IDENTF;
identmod        = IDENTMOD;
comment         = COMMENT;
decls           = DECLS;
decl            = IMPORTS LABELDECLS CONSTDEF TYPEDEF VARDECLS
                  FWDPROCDECL PROCDECL FWDFUNCTDECL FUNCTDECL
                  LONGCOMMENT;
expodecl        = IMPORTS LABELDECLS CONSTDEFS TYPEDEFS VARDECLS
                  PROCHEAD FUNCTHEAD LONGCOMMENT;
label           = INTCONST;
constdef        = CONSTDEF;
typedef         = TYPEDEF CTYPEDEF;
type            = SCALARTYPE SUBRANGETYPE IDENTT INTEGER BOOLEAN
                  REAL CHAR POINTERTYPE STRING ARRAYTYPE
                  RECORDTYPE VARRECORDTYPE SETTYPE FILETYPE;
packing         = EMPTY PACKED;
indextypes      = INDEXTYPES;
maxlegth        = MAXLENGTH EMPTY;
intconst        = INTCONST;
simpletype      = SCALARTYPE SUBRANGETYPE IDENTT INTEGER BOOLEAN
                  REAL CHAR;
fixedpart       = FIXEDPART EMPTY;
fields          = FIELDS;
variantpart     = VARIANTPART;
field           = FIELD CFIELD;
caselabels      = CASELABELS OTHERWISE;
fieldidents     = FIELDIDENTS;
```

```
fieldident     = FIELDIDENT;
tagfield       = TAGFIELD EMPTY;
variants       = VARIANTS;
variant        = VARIANT;
vardecl        = VARDECL;
idents         = IDENTS;
prochead       = PROCHEAD;
pident         = IDENTP;
block          = BLOCK;
formalparam    = PARAMGROUP VARPARAMGROUP;
tident         = IDENTT INTEGER REAL BOOLEAN CHAR STRING;
statements     = STATEMENTS;
compound       = COMPOUND;
statement      = ASSIGN PROC GOTO EMPTY COMPOUND IF IFELSE
                 CASES WHILE REPEAT FOR WITH LABEL EXIT
                 CSTATE IFC;
leftside       = IDENTV INDEXEDVAR RECFIELD FILEBUFFER
                 REFERENCEDVAR IDENTFUNCT;
variable       = IDENTV INDEXEDVAR RECFIELD FILEBUFFER
                 REFERENCEDVAR;
expressions    = EXPRESSIONS;
expression     = EQUAL NOTEQUAL LESSTHAN LTEQL GREATERTHAN
                 GTEQL IN ADD SUBTRACT OR MULTIPLY DIVIDE
                 INTDIVIDE MOD AND PARENEXPR FUNCTDESIG SET NOT
                 IDENTV INDEXEDVAR RECFIELD FILEBUFFER
                 REFERENCEDVAR INTCONST REALCONST STRINGCONST
                 IDENTC NIL TRUE FALSE;
constexp       = EQUAL NOTEQUAL LESSTHAN LTEQL GREATERTHAN
                 GTEQL IN ADD SUBTRACT OR MULTIPLY DIVIDE
                 INTDIVIDE MOD AND PARENEXPR FUNCTDESIG SET NOT
                 IDENTV INDEXEDVAR RECFIELD FILEBUFFER
                 REFERENCEDVAR INTCONST REALCONST STRINGCONST
                 IDENTC NIL TRUE FALSE;
functident     = IDENTFUNCT;
identprocfunc  = IDENTPROCFUNCT;
element        = EXPRESSION EXPRPAIR;
actualparams   = ACTUALPARAMS;
cases          = CASES;
caseelement    = CASEELEMENT;
casestatlabs   = CASESTATLABS;
casestatlab    = INTCONST REALCONST IDENTC CHARCONST OTHERWISE
                 SUBRANGE;
forlist        = TO DOWNTO;
recordvar      = RECORDVAR;}
```

REFERENCIAS

[BACKUS 59] Backus J. W.
The Syntax and Semantics of the Proposed International Algebraic Languaje of the Zurich ACM-GAMM Conference.
Technical Report, Proceeding of the international conference of information processing, 1959.

[CHOMSKY 59] Chomsky N.
On Certain Formal Properties of Grammars.
Information and control, 1959.

[COLIN 71] Colin A.J.T.
Introduction to Operating Systems.
MacDonald-Elsevier, 1971.

[DENNING 70] Denning J. Peter.
Virtual Memory.
Computing Surveys 2(3), septiembre, 1970.

[DONOVAN 81] Donovan J. John.
Systems Programming.
McGraw-Hill, 1981.

[HABERMANN 76] Habermann A. N.
Introduction to Operating System Design.
SRA, 1976.

[HANSEN 71] Hansen Wilfred J.
Creation of Hierarchic Text with a Computer Display..
PhD thesis, Stanford University, 1971.

[HORGAN 84] Horgan J.R. y Moore D.J.
Techniques for Improving Languaje-Based Editors.
ACM SIGPLAN 19(3), MAYO, 1984.

[JENSEN 75] Jensen K. y Wirth N.
PASCAL: User Manual and Report
1975.

[KURZBAN 75] Kurzban A. S. et. al.
Operating Systems Priciples.
Petrocelli/charter, 1975.

[LISTER 79] Lister A.M.
Fundamentals Of Operating Systems.
MacMillan Press LTD, 1979.

[MEDINA-MORA 81a]
Medina-Mora I. Raul.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, 1981.

[MEDINA-MORA 81b]
Medina-Mora Raul & Feiler Peter.
ALOE users and Implementors Guide.
Technical Report CMU-CS-81-145, Carnegie-Mellon University, noviembre, 1981.

[MEDINA-MORA 81c]
Medina-Mora Raul & Notkin David S.
An Incremental Programming Environment.
IEEE transaction on Software Engineering , noviembre, 1981.

[NICHOLLS 75] Nicholls J. E.
The Structure and Design of Programming Languages.
Addion-Wesley, 1975.

[RANDELL 68] Randell B. & Kuehner C.J.
Dynamic Storage Allocation Systems.
Communications of ACM 11(5), mayo, 1968.

[RITCHIE 78] Kernighan Brian & Ritchie Dennis.
The C Programming Languaje.
Prentice-Hall, 1978.

[TEITELBAUM 81]
Teitelbaum T. y Reps T. .
The Cornell Program SYNTHESIZER a Syntax-Directed Programing Environment.
Comunication of the ACM 24(9), septiembre, 1981.

[WATSON 70] Watson W. Richard.
Timesharing System Design Concepts.
Mc. Graw-Hill, 1970.

[ZELKOWITZ 84] Zelkowitz V. Marvin.
A Small Contribution to Editing with a Syntax Directed Editor.
ACM SIGPLAN 19(3), MAYO, 1984.