HERRAMIENTAS DE SOFTWARE PARA COMPILADORES

CIRO ARAUJO RAMIREZ

NO. CTA. 8457135-1

TESIS DE GRADO PARA OBTENER EL TITULO DE
LIC. EN MATEMATICAS APLICADAS Y COMPUTACION

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES ACATLAN

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

MEXICO, D.F.

1990

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE.

INTRODUCCION.

En algunas situaciones, dentro del área de desarrollo de sistemas de información, a varios de nosotros se nos ha presentado el problema de escribir manualmente un programa que se comporte como un compilador para incluirlo en algún otro programa de aplicación, como por ejemplo uno para analizar sintácticamente y resolver una ecuación. Para poder escribir un programa con estas características, se requiere contar con técnicas y herramientas de software que simplifiquen dicha tarea y que puedan usarse en diferentes aplicaciones. Esta es la razón por la cual, en esta tesis se describen y se ponen en práctica algunas de las principales técnicas que nos ayudan a construir un compilador.

En el primer Capítulo se discuten las fases de un compilador, la interacción de éstas con la tabla de símbolos y el manejo de errores. Concluye con un programa en Turbo Pascal para un parser/traductor.

En el segundo Capítulo se describe la función y la programación de un analizador léxico para el lenguaje Mini-Pascal, un subconjunto del Pascal.

El tercer Capítulo está dedicado al análisis sintáctico. Primeramente, se discuten dos clases de análisis sintáctico: ascendente y descendente, de las cuales se hace énfasis en el análisis sintáctico descendente. A continuación se define la sintaxis del lenguaje Mini-Pascal y se desarrolla un analizador sintáctico para este lenguaje.

El cuarto Capítulo trata el análisis de alcance, el cual es una extensión del analizador sintáctico. Para realizar el análisis de alcance, primero se definen las reglas de alcance y los objetos -constantes, tipos de datos, registros, campos, variables y procedimientos- de un programa en Mini-Pascal y después se describen los algoritmos que se usarán para realizar este análisis.

El quinto Capítulo describe la forma en la que el compilador usa las definiciones de objetos para realizar el análisis de tipos, el cual también es una extensión del analizador sintáctico.

La generación de código se discute en el sexto Capítulo. En primer lugar, se describe el conjunto de instrucciones para una computadora hipotética y después se explica cómo se genera el código para esta computadora.

En el apéndice A se muestran los listados de los programas que se describen en los capítulos que integran esta tesis.

CAPITULO 1. INTRODUCCION A LOS COMPILADORES.

INTRODUCCION.

Los principios y técnicas para escribir compiladores son indispensables para un Profesionista en Ciencias de la Computación o en Informática, debido a que existen varias áreas muy importantes que usan estos principios y técnicas para su desarrollo. Algunas de estas áreas son: preprocesadores, sistemas operativos, lenguaje natural, rutinas de análisis de comandos, procesamiento de textos y conversores fuente-fuente.

En este Capítulo se da una idea general del proceso de compilación mediante la descripción de cada uno de los componentes de un compilador y el medio ambiente en el que trabajan los compiladores. Se concluye con la construcción de un pequeño parser/traductor para analizar y traducir a lenguaje ensamblador del microprocesador 8088 un mini lenguaje para expresiones aritméticas. El objetivo de este parser/traductor es ilustrar en forma sencilla algunas de las técnicas de análisis y traducción. Este parser/traductor considera sólo una línea de entrada y tokens de un carácter. Como salida, se despliega en la pantalla el código equivalente en lenguaje ensamblador. La extensión al mundo real es directa. Agregar nombres multicarácter y varias líneas, por ejemplo, no cambia la estructura del parser. Si podemos emitir código en pantalla, también podemos emitir código en un archivo.

1.1 COMPILADORES.

1.1.1 El modelo de compilación análisis-síntesis.

En la compilación de un programa fuente podemos distinguir dos partes: el análisis y la síntesis. La parte de análisis se encarga de separar el programa fuente en varios componentes y crea una representación intermedia del programa fuente. La parte de síntesis construye el programa objeto deseado a partir de la representación intermedia. De éstas dos partes, la de síntesis requiere técnicas más especializadas.

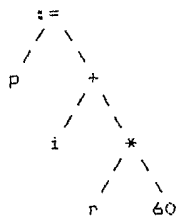


Fig. 1.1 Arbol sintáctico para p := i + r * 60.

Durante el análisis, se determinan las operaciones contenidas en el programa fuente y se registran en una estructura jerárquica llamada árbol. Por lo regular, se usa un tipo especial de árbol llamado árbol sintáctico, en el cual cada nodo representa una operación y los hijos de los nodos representan los argumentos de la operación. Por ejemplo, un árbol sintáctico para la proposición de asignación p := i + r * 60 se muestra en la figura 1.1.

1.1.2 El contexto de un compilador.

Además del compilador, se requieren otros programas para crear un programa objeto ejecutable. Un programa fuente puede ser dividido en varios módulos almacenados en archivos separados. La tarea de reunir el programa fuente es algunas veces encomendada a un programa distinto, llamado preprocesador. A su vez, el preprocesador también puede ser expandido en otras partes llamadas macros.

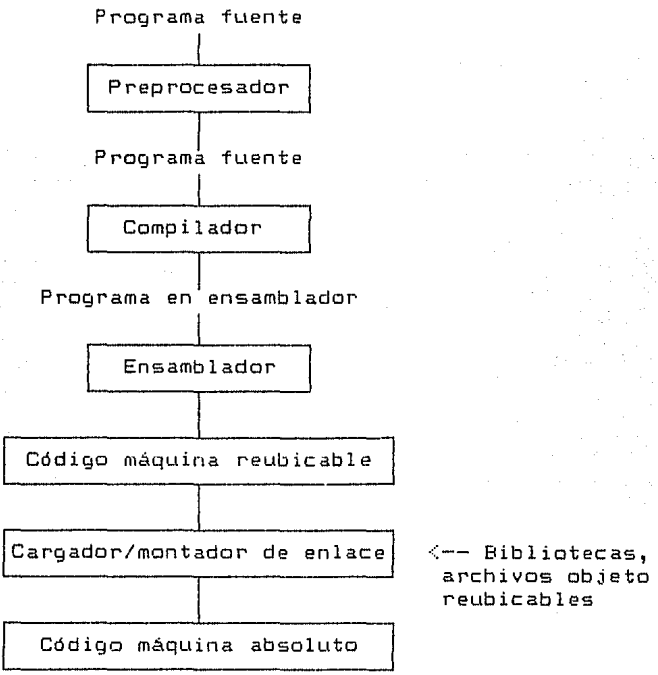


Fig. 1.2 Esquema típico de la compilación de un programa.

La figura 1.2 muestra un esquema de una compilación típica. El programa objeto creado por el compilador puede requerir un procesamiento adicional antes que pueda ejecutarse. El compilador de la figura 1.2 crea el código en lenguaje ensamblador, el cual es traducido posteriormente, mediante un ensamblador, a código máquina para que de esta forma pueda ser enlazado con algunas bibliotecas de código y realmente, se ejecute en la computadora.

1.2 ANALISIS DEL PROGRAMA FUENTE.

En la compilación, el análisis consiste de tres fases:

i) Análisis lineal, en el cual el conjunto de caracteres del programa fuente se lee de izquierda a derecha y se agrupa en tokens - secuencias de caracteres que tienen un significado colectivo.

ii) Análisis jerárquico, en el cual los caracteres o tokens son agrupados jerárquicamente dentro de colecciones anidadas de significado colectivo.

iii) Análisis semántico, en éste se realizan ciertos chequeos para asegurarse que los componentes de un programa sean completamente significativos.

1.3 LAS FASES DE UN COMPILADOR.

Conceptualmente, un compilador opera en fases, cada una de las cuales transforma el programa fuente de una representación a otra. Una descomposición típica de un compilador se muestra en la figura 1.3. En la práctica, algunas de las fases pueden ser agrupadas en una sola, y su representación intermedia de éstas no necesita ser construida explícitamente.

Las tres primeras fases forman la parte de análisis de un compilador. En la figura 1.3 también se muestran otras dos actividades adicionales, la tabla de símbolos y el manejo de errores, interactuando con las seis fases: análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, optimización de código y generación de código.

1.3.1 Análisis léxico.

En un compilador, el análisis lineal se denomina análisis léxico o scanning. Por ejemplo, en el análisis léxico los caracteres de la proposición de asignación:

$$p := i + r * 60 \qquad (1.1)$$

pueden ser agrupados dentro de los siguientes tokens:

- El identificador p.
- El símbolo de asignación.
- El identificador i.
- El signo +.
- El identificador r.
- El signo de multiplicación, *.
- El número 60.

Los espacios que separan los caracteres de estos tokens pueden ser eliminados durante el análisis léxico.

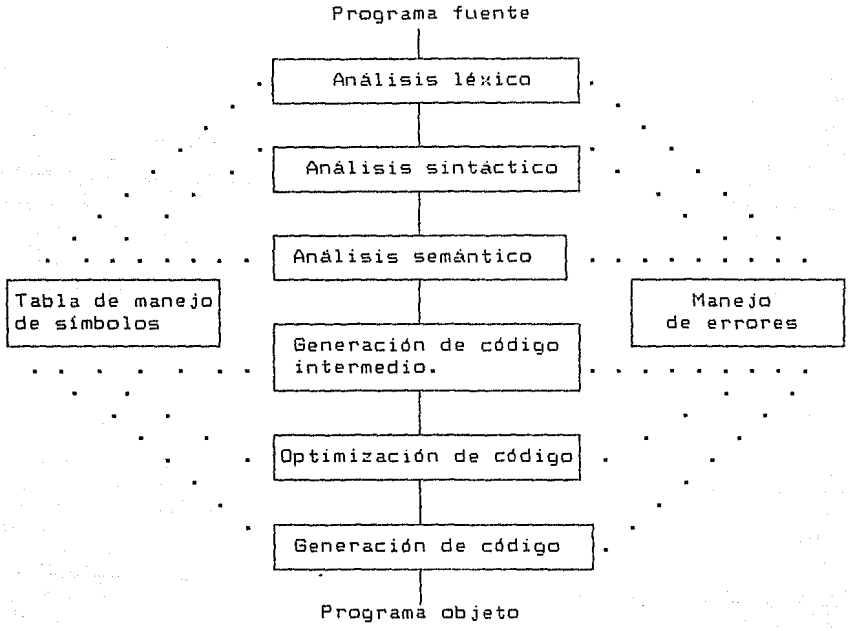


Fig. 1.3 Fases de un compilador.

1.3.2 Análisis sintáctico.

El análisis jerárquico es llamado análisis sintáctico o parsing. Consiste en agrupar los tokens del programa fuente en frases gramaticales que son utilizadas por el compilador para sintetizar la salida. Usualmente, las frases gramaticales del programa fuente están representadas por un árbol de reconocimiento sintáctico como el que se muestra en la figura 1.4.

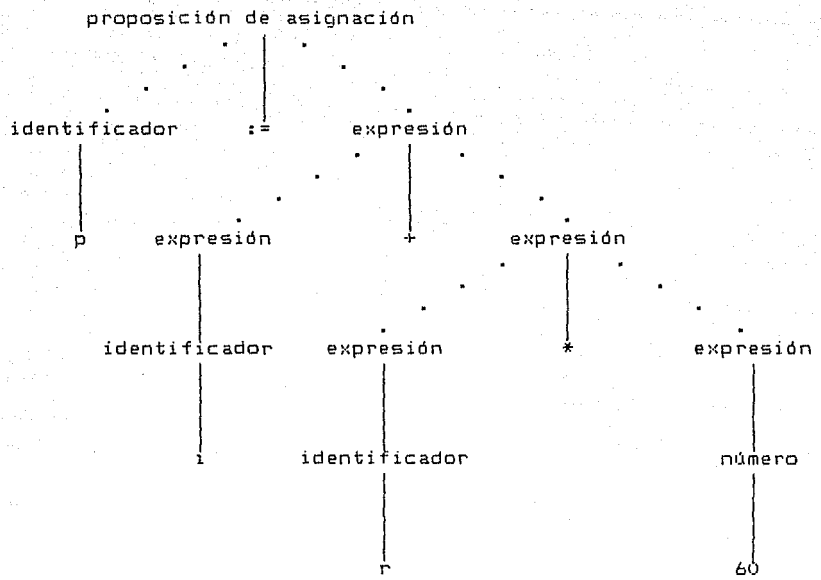


Fig. 1.4 Arbol de reconocimiento sintáctico para p := i+r*60.

En la expresión i + r * 60, la frase r * 60 es una unidad lógica debido a las convenciones usuales de la aritmética, que nos dicen que la multiplicación se realiza antes que la suma. Debido a que la expresión i + r está seguida por un *, ésta no se agrupa dentro de una sola frase por si misma en la figura 1.4.

La estructura jerárquica de un programa es usualmente expresada mediante reglas recursivas. Por ejemplo, podemos tener las siguientes reglas como parte de la definición de expresiones:

1. Cualquier identificador es una expresión.
2. Cualquier número es una expresión.
3. Si $expresión_1$ y $expresión_2$ son expresiones, entonces también lo son:

$expresión_1$ + $expresión_2$
$expresión_1$ * $expresión_2$
($expresión_1$)

Las reglas (1) y (2) son reglas básicas (no recursivas), mientras que (3) define expresiones en términos de operadores aplicados a otras expresiones. Por lo tanto, por la regla (1), i y r son expresiones. Por la regla (2), 60 es una expresión, mientras

que por la regla (3), podemos deducir primero que r * 60 es una expresión y finalmente, que i + r * 60 es una expresión.

Similarmente, muchos lenguajes definen proposiciones recursivamente mediante reglas como:

i) Si identificador$_1$ es un identificador, y expresión$_2$ es una expresión, entonces

identificador$_1$:= expresión$_2$

es una proposición.

ii) Si expresión$_1$ es una expresión y proposición$_2$ es una proposición, entonces

while (expresión$_1$) **do** proposición$_2$
if (expresión$_1$) **then** proposición$_2$

son proposiciones.

La división entre análisis léxico y análisis sintáctico es un poco arbitraria. Usualmente, escogemos una división que simplifique la tarea completa de análisis. Un factor en determinar la división es verificar si la construcción de un lenguaje fuente es inherentemente recursivo o no lo es. Las construcciones léxicas no requieren recursión, mientras que las construcciones semánticas frecuentemente lo requieren. Las gramáticas de contexto libre son una formalización de reglas recursivas que pueden ser usadas para guiar el análisis sintáctico.

Por ejemplo, no se requiere recursión para reconocer identificadores, los cuales típicamente son cadenas de letras y dígitos. Normalmente, reconoceremos identificadores mediante una exploración simple de la secuencia de entrada, esperando hasta que se encuentre un carácter que no sea una letra o un digito, y después agrupando todas las letras y dígitos encontrados hasta el momento en un token de identificador. Los caracteres asi agrupados se registran en una tabla, llamada tabla de símbolos, y eliminados de la entrada de tal forma que pueda empezar el procesamiento del siguiente token.

Por otra parte, este tipo de exploración lineal no es muy eficiente para analizar expresiones o proposiciones. Por ejemplo, no podemos agrupar paréntesis adecuadamente en expresiones, o el begin y el end en proposiciones sin colocar algún tipo de jerarquía o estructura anidada en la entrada.

El árbol de reconocimiento sintáctico de la figura 1.4 describe la estructura sintáctica de la entrada. Una representación interna más común de esta estructura sintáctica

está dada por el árbol sintáctico de la figura 1.5(a). Un árbol sintáctico es una representación compacta del árbol de reconocimiento sintáctico en el cual los operadores aparecen como los nodos internos, y los operandos de un operador son los hijos del nodo de ese operador.

1.3.3 Análisis semántico.

La fase del análisis semántico examina el programa fuente para detectar errores de semántica y reune información de tipos para la fase de generación de código. Usa la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

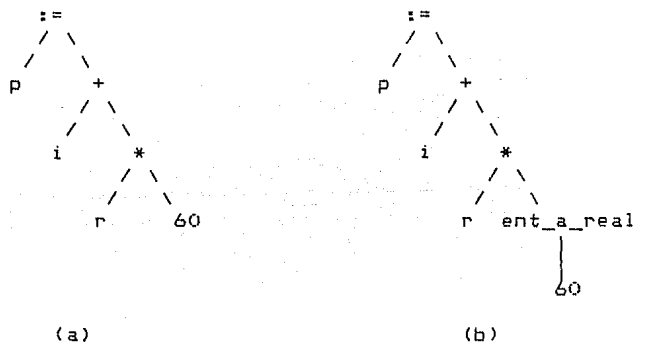


Fig. 1.5 Análisis semántico para insertar la conversión de entero a real.

Un componente importante del análisis semántico es la verificación de tipos. Aqui el compilador verifica que cada operador tenga los operandos que estén permitidos por la especificación del lenguaje fuente. Por ejemplo, varias definiciones de lenguajes de programación requieren un compilador para que reporte un error cada vez que un número real se use como un índice en un arreglo. Sin embargo, la especificación del lenguaje puede permitir algunas coacciones de operandos, por ejemplo, cuando un operador aritmético binario es aplicado a un entero y a un real. En este caso, el compilador puede necesitar convertir el entero a real.

Ejemplo 1.1. Dentro de una computadora, el patrón de bits para representar un entero por lo general difiere del patrón de bits para un real, aún si el número entero y el real tienen el mismo valor. Supongamos que todos los identificadores en la figura 1.5 han sido declarados como reales y el 60 por si mismo va a asumir un valor entero. La verificación de tipos de la figura

1.5(a) revela que el operador * es aplicado a un real, r, y a un entero, 60. El enfoque general es convertir el entero a real. Esto se muestra en la figura 1.5(b) con la creación de un nodo extra para el operador ent_a_real que explícitamente convierte un entero a un real. Alternativamente, puesto que el operando de ent_a_real es una constante, el compilador puede reemplazar la constante entera por una constante real equivalente.

1.3.4 Tabla de manejo de símbolos.

La función esencial de un compilador es registrar los identificadores utilizados en el programa fuente y reunir información acerca de los distintos atributos de cada identificador. Estos atributos pueden proporcionar información referente a la asignación de memoria para un identificador, su tipo, su alcance (en que partes del programa es válido), y, en el caso de procedimientos, algunas cuestiones como el número y tipo de sus argumentos, el método para pasar cada argumento (por ejemplo, por referencia), y el tipo regresado, si es que lo hay.

Una tabla de símbolos es una estructura de datos que contiene un registro para cada identificador, con campos para los atributos del identificador. La estructura de datos nos permite encontrar el registro para cada identificador y almacenar o recuperar rápidamente datos de ese registro.

Cuando se detecta un identificador en el programa fuente mediante el análisis léxico, el identificador es colocado dentro de la tabla de símbolos. Sin embargo, los atributos de un identificador no se pueden determinar normalmente durante el análisis léxico. Por ejemplo, en una declaración del lenguaje Pascal como:

```
Var p, i, r : real;
```

el tipo real no es conocido cuando p, i, y r son examinados por el analizador léxico.

Las fases restantes introducen información referente a los identificadores dentro de la tabla de símbolos y después usan esta información de varias formas. Por ejemplo, cuando se realiza el análisis semántico y la generación de código intermedio, necesitamos saber que tipos de identificadores son, para que de esta forma se pueda verificar que el programa fuente los use en las formas válidas, y así podamos generar las operaciones apropiadas con ellos. El generador de código típicamente introduce y usa información detallada acerca del almacenamiento asignado a los identificadores.

1.3.5 Detección y reporte de errores.

En cada fase se pueden encontrar errores. Por lo tanto, una fase debe encargarse de esos errores, de tal forma que la compilación pueda proceder, permitiendo que los errores subsiguientes del programa puedan ser detectados.

Las fases del análisis sintáctico y semántico usualmente manejan gran parte de los errores detectados por un compilador. La fase lexicográfica puede detectar errores donde los caracteres que quedan en la entrada no forman un token del lenguaje. Los errores donde el token viola las reglas de estructura (sintaxis) del lenguaje están determinadas por la fase del análisis sintáctico.

Durante el análisis semántico el compilador trata de detectar construcciones que tienen la estructura sintáctica correcta, pero no tienen significado en la operación involucrada, por ejemplo, si tratamos de sumar dos identificadores, uno de los cuales es el nombre de un arreglo, y el otro el nombre de un procedimiento.

1.3.6 Generación de código intermedio.

Después del análisis sintáctico y del semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Podemos pensar esta representación intermedia como un programa para una máquina abstracta. Esta representación intermedia debería tener dos propiedades importantes; debe ser fácil de producir, y fácil de traducir a código objeto.

La representación intermedia puede tener una variedad de formas. Una de estas formas se conoce como "código de tres direcciones", en la cual cada localidad de memoria puede actuar como un registro. El código de tres direcciones consiste de una serie de instrucciones, cada una de las cuales tiene a lo más tres operandos. El programa fuente (1.1) puede ser representado en código de tres direcciones como:

```
temp1 := ent_a_real(60)
temp2 := id3 * temp1
temp3 := id2 + temp2                    (1.2)
id1   := temp3
```

Esta forma intermedia tiene varias propiedades. Primera, cada instrucción en código de tres direcciones tiene a lo más un operador, además del de asignación. Por lo tanto, cuando generamos estas instrucciones, el compilador tiene que decidir, el orden en el cual se efectuarán las operaciones; la multiplicación precede a la adición en el programa fuente (1.1). Segunda, el compilador debe generar un nombre temporal para almacenar el valor calculado por cada instrucción. Tercera, algunas instrucciones del código de

tres direcciones tienen menos de tres operandos, por ejemplo, la primera y última instrucción en (1.2).

1.3.7 Optimización de código.

La fase de optimización de código intenta mejorar el código intermedio, de tal forma que la máquina pueda ejecutarlo más rápido. Algunas optimizaciones son triviales. Por ejemplo, un algoritmo natural genera el código intermedio mostrado en (1.3), usando una instrucción para cada operador en el árbol representado en la figura (1.6) después del análisis semántico, aunque existe una mejor forma para realizar el mismo cálculo, usando dos instrucciones:

```
temp1 := id3 * 60.0
id1   := id2 + temp1                    (1.3)
```

No hay nada incorrecto con este simple algoritmo, puesto que el problema puede ser resuelto durante la fase de optimización de código. Esto es, el compilador puede deducir que la conversión del 60 de la representación entera a real puede hacerse desde la fase de compilación, por lo tanto, la operación de conversión de entero a real puede ser eliminada. Además, temp3 se utiliza solamente una vez, para transmitir su valor a id1. En este caso es recomendable sustituir id1 por temp3, con lo cual la última proposición de (1.2) no es necesaria y el resultado es el código de (1.3).

Hay una gran variación de la cantidad de optimización de código diferente que realizan los compiladores. La mayoria de éstos, llamados "compiladores optimizadores", ocupan una fracción de tiempo significativo en esta fase. Sin embargo, hay optimizaciones simples que significativamente mejoran el tiempo de ejecución del programa objeto sin hacer muy lenta la compilación.

1.3.8 Generación de código.

La fase final del compilador es la generación de código objeto, integrada normalmente, de código máquina reubicable o código ensamblador. Las localidades de memoria son seleccionadas para cada una de las variables usadas por el programa. Posteriormente, cada instrucción intermedia se traduce a una secuencia de instrucciones en lenguaje máquina que realizan la misma tarea. Un aspecto crucial es la asignación de registros a las variables.

Por ejemplo, usando los registros 1 y 2, la traducción del código de (1.3) es como se muestra en (1.4).

El primero y segundo operandos de cada instrucción especifican el destino y el origen, respectivamente. La letra F en cada instrucción indica que esas instrucciones manejan números de

punto flotante. Este código mueve el contenido de la dirección id3 al registro 2, después multiplica éste por la constante real 60.0. El # significa que el 60.0 va a ser tratado como una constante. La tercer instrucción transfiere el valor de id2 al registro 1 y lo suma al valor previamente calculado en el registro 2. Finalmente, el valor del registro 1 se transfiere a la dirección id1, implementando así el código de la proposición de asignación de la figura 1.6.

```
MOV  R2,id3
MULF R2,#60.0
MOVF R1,id2                    (1.4)
ADDF R1,R2
MOVF id1,R1
```

1.3.9 Las fases de análisis.

Conforme la compilación avanza, la representación interna que hace el compilador del programa fuente cambia. Ilustraremos estas representaciones considerando la compilación de la proposición:

p := i + r * 60

La figura 1.6 muestra la representación de esta proposición después de cada fase. La fase de análisis léxico lee los caracteres del programa fuente y los agrupa en una serie de cadenas de tokens en los cuales cada token representa una secuencia de caracteres coherente, tal como un identificador, una palabra clave (if, while, etc.), un carácter de puntuación, o un operador multicarácter como :=. La secuencia de caracteres que forman un token se llama lexema del token.

Ciertos tokens serán aumentados con un "valor léxico". Por ejemplo, cuando se encuentra un identificador, como por ejemplo r, el analizador léxico no sólo generará un token, escrito id, sino también introducirá el lexema r dentro de la tabla de símbolos, si es que aún no está allí. "El valor léxico" asociado con ésta ocurrencia de id apunta al elemento de la tabla de símbolos para el identificador r.

En este caso, utilizaremos id_1, id_2, e id_3 para p, i, y r respectivamente, para enfatizar que la representación interna de un identificador es diferente de la secuencia de caracteres que forman el identificador. La representación de p := i + r * 60 después del análisis léxico es como se ilustra en (1.5).

$$id_1 := id_2 + id_3 * 60 \qquad (1.5)$$

También debemos crear tokens para el operador multicarácter := y el número 60 para reflejar su representación interna, pero lo pospondremos para tratarlo más adelante.

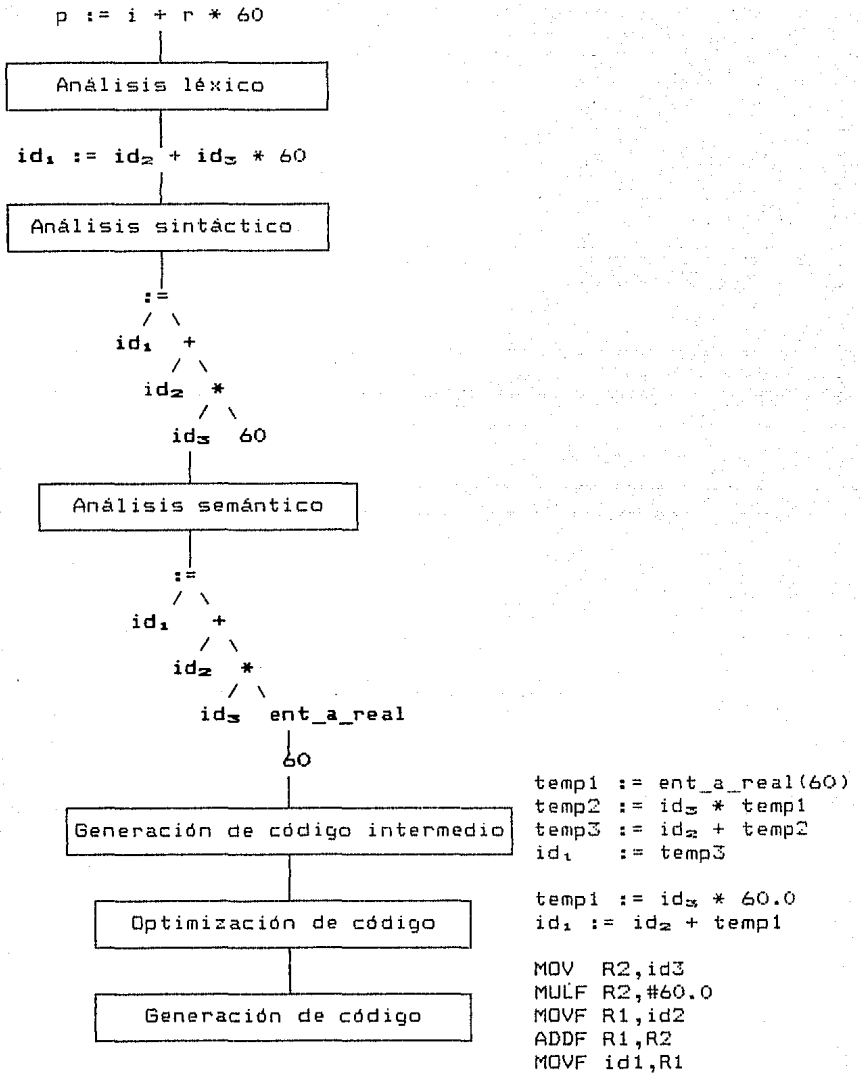


Fig. 1.6 Compilación de una proposición.

1.4 AGRUPAMIENTO DE LAS FASES.

La discusión de las fases en la sección 1.3 trató con la organización lógica de un compilador. En una implementación, las actividades de más de una fase se agrupan en dos partes conocidas como parte traductora y parte generadora.

1.4.1 Parte traductora y parte generadora.

La parte traductora consiste de las fases, o partes de las fases, que dependen principalmente del lenguaje fuente y son independientes del código máquina. Normalmente, incluye el análisis léxico y el análisis sintáctico, la creación de la tabla de símbolos, el análisis semántico, y la generación de código intermedio. La parte traductora también comprende el manejo de errores de cada una de estas fases.

La parte generadora incluye las porciones del compilador que dependen del código máquina y generalmente, éstas no dependen del lenguaje fuente, sino del código intermedio. En la parte generadora encontramos aspectos de la fase de optimización de código, y la generación de código, junto con las operaciones necesarias para el manejo de errores y de la tabla de símbolos. La figura 1.7 muestra el esquema de la parte traductora y la parte generadora de un compilador desde el punto de vista de un lenguaje intermedio LI.

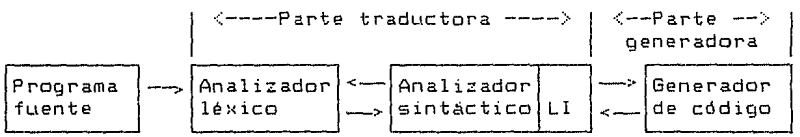


Fig. 1.7 Parte traductora y parte generadora de un compilador desde el punto de vista de un lenguaje intermedio LI.

1.4.2 Pasos.

Es común que varias fases de compilación se agrupen en pasos que consisten en leer un archivo de entrada y escribir un archivo de salida. Por ejemplo, el análisis léxico, análisis sintáctico, análisis semántico y la generación de código intermedio podrían ser agrupadas en un paso. De esta forma, el flujo de tokens resultante del análisis léxico podría ser traducido directamente a código intermedio. El analizador sintáctico intenta descubrir la estructura gramatical de los tokens que analiza; obtiene los tokens conforme los necesite, mediante una llamada al analizador léxico. Conforme va descubriendo la estructura gramatical, el

parser llama al generador de código intermedio para realizar el análisis semántico y generar una porción de código.

1.4.3 Reducción del número de pasos.

Es deseable tener relativamente menos pasos, dado que lleva tiempo leer y escribir archivos intermedios. Por otro lado, si agrupamos varias fases en un paso, podríamos estar forzados a mantener el programa completo en memoria, dado que una fase podría necesitar información en orden diferente al que generó alguna fase anterior.

Para algunas fases, la agrupación en un paso presenta pocos problemas. Por ejemplo, la interface entre el analizador léxico y el analizador sintáctico puede limitarse a un token. Por otro lado, con frecuencia es muy difícil realizar la generación de código hasta que se haya completado la generación de código intermedio. No podemos generar el código objeto para una construcción si no sabemos los tipos de variables involucrados en esta construcción. No podemos determinar la dirección de un salto hacia delante hasta que no se haya visto el código fuente que interviene y se haya generado el código resultante para éste.

En algunos casos, es posible dejar un espacio en blanco para colocar la información que falta, y llenar este espacio cuando la información esté disponible. En particular, la fase de generación de código intermedio y la generación de código objeto pueden ser agrupadas en un paso usando una técnica llamada backpatching.

1.5 CONSTRUCCION DE UN PARSER/TRADUCTOR.

En esta sección ilustraremos un método que hace que el análisis sea un proceso muy sencillo, construiremos un pequeño parser/traductor que usaremos en el contexto de análisis y traducción de expresiones aritméticas, pero las técnicas también se aplican a otros elementos de traducción.

1.5.1 Definición de la gramática.

Para definir los elementos del lenguaje, usaremos la forma de Backus-Naur (BNF), la forma convencional para representar la definición de un lenguaje.

```
<Asignación>   ::= <Variable> = <Expresión>
<Expresión>    ::= <Término> [<Op. sumador> <Término>]*
<Término>      ::= <Factor> [<Op. mult.> <Factor>]*
<Factor>       ::= <Número> | <variable> | (<expresión>)
<Variable>     ::= 'A'..'Z'                                    (1.6)
<Número>       ::= '0'..'9'
<Op. sumador>  ::= '+' | '-'
<Op. mult.>    ::= '*' | '/'
```

El mini lenguaje está definido por la gramática (1.6), en la cual el asterisco es el símbolo BNF para repetición. Esto significa que el término que está entre paréntesis cuadrados puede repetirse cero o más veces.

Algunas de las instrucciones válidas para esta gramática son:

```
a = b * (c - d)
x = i + p * 6
y = (5+6) + (5*2)/(4-2)
```

1.5.2 Características del parser/traductor.

El parser para esta gramática tiene las características siguientes:

i) E/S a través del monitor.

ii) Se considera sólo una línea de entrada.

iii) Los tokens -nombres de variables, constantes y operadores- están formados de un carácter para simplificar el análisis léxico.

iv) Usa análisis sintáctico de descenso recursivo, ya que es la única técnica que tiene sentido al construir un compilador manualmente.

v) Al detectar un error se desplegará un mensaje apropiado y se suspenderá el proceso.

vi) La salida es código fuente en ensamblador en línea para el µp 8088 conforme se realiza el análisis sintáctico, de esta forma se dejan al ensamblador los detalles de la tabla de símbolos y el manejo del puntero de instrucción.

vii) El resultado de una expresión se coloca en el registro AX. Para las variables intermedias se usa la pila, de tal forma que las operaciones aritméticas involucran el registro AX y la variable que está en el tope de la pila.

1.5.3 Programación del parser/traductor.

El lenguaje que usaremos para la programación del parser/traductor será Turbo Pascal, debido a que es el lenguaje con el cual estamos más familiarizados. Sin embargo, es muy fácil pasar el programa a otro lenguaje, por ejemplo C. Las funciones tales como Es_alfa, Es_digito y Obt_car, simulan las funciones de C conocidas como IsAlpha(), IsDigit() y GetChar(), respectivamente.

El procedimiento Equipara espera un token especifico. Si lo encuentra, obtiene el siguiente token. Los procedimientos Obt_ident y Obt_num son reconocedores: aceptan un token del tipo correspondiente y saltan al siguiente token antes de regresar.

Para generar el código en ensamblador, se utilizan los procedimientos que empiezan en Carga_cte y terminan en el procedimiento almacena.

Existen tres rutinas para el manejo de errores: Obt_ident, Obt_num y Equipara.

El listado del programa se muestra en el apéndice A.1.

1.5.4 Prueba.

Al ejecutar el programa del parser/traductor obtenemos:

Entrada: y = (5+6)+(5*2)/(4-2)

Salida:

```
MOV AX,5
PUSH AX     ; Coloca el reg. AX en la pila
MOV AX,6
MOV BX,AX   ; Segundo sumando en BX
POP AX      ; Primer sumando en AX
ADD AX,BX   ; AX <--- AX + BX
PUSH AX     ; Coloca el reg. AX en la pila
MOV AX,5
PUSH AX     ; Coloca el reg. AX en la pila
MOV AX,2
MOV CX,AX   ; Primer factor en CX, parte baja
POP AX      ; Segundo factor en AX, parte baja
MUL CL      ; AX <--- AL * CL
PUSH AX     ; Coloca el reg. AX en la pila
MOV AX,4
PUSH AX     ; Coloca el reg. AX en la pila
MOV AX,2
MOV BX,AX   ; Sustraendo en BX
POP AX      ; Minuendo en AX
SUB AX,BX   ; AX <--- AX-BX
MOV CX,AX   ; Divisor en la parte baja de CX
POP AX      ; Dividendo en la parte baja de AX
DIV CL      ; Divide AL entre CL
              AH = cociente, AL = residuo
MOV BX,AX   ; Segundo sumando en BX
POP AX      ; Primer sumando en AX
ADD AX,BX   ; AX <--- AX + BX
MOV Y,AX    ; Se almacena el resultado en la variable Y
```

Mediante una segunda ejecución obtenemos:

Entrada: x = i + p * 6

Salida:

```
MOV AX,I
PUSH AX     ; Coloca el reg. AX en la pila
MOV AX,P
PUSH AX     ; Coloca el reg. AX en la pila
MOV AX,6
MOV CX,AX   ; Primer factor en CX, parte baja
POP AX      ; Segundo factor en AX, parte baja
MUL CL      ; AX <--- AL * CL
MOV BX,AX   ; Segundo sumando en BX
POP AX      ; Primer sumando en AX
ADD AX,BX   ; AX <--- AX + BX
MOV X,AX    ; Se almacena el resultado en la variable X
```

CAPITULO 2. ANALISIS LEXICO.

INTRODUCCION.

La función del analizador léxico es leer un programa fuente, un carácter a la vez, y traducirlo a una secuencia de unidades primitivas denominadas tokens. Las palabras clave, los identificadores, las constantes y los operadores son ejemplos de tokens.

2.1 EL ROL DEL ANALIZADOR LEXICO.

El análisis léxico es la primer fase de un compilador. Su tarea principal es leer una cadena de caracteres y producir como salida una secuencia de tokens que utiliza el parser para el análisis sintáctico. Esta interacción, resumida en forma esquemática se muestra en la figura 2.1, comúnmente se implementa haciendo que el analizador léxico sea una subrutina del parser. Después de recibir un comando "obtener el siguiente token", enviado por el parser, el analizador léxico lee caracteres de entrada hasta que pueda identificar el siguiente token.

Puesto que el analizador léxico es la parte del compilador que lee el programa fuente, éste también puede realizar ciertas tareas. Una de las tareas es saltar los comentarios y los espacios en blanco en la forma de carácter tabulador, blanco y nuevalínea. Otra es la correlación de mensajes de error del compilador con el programa fuente. Por ejemplo, el analizador léxico puede llevar el control del número de caracteres nuevalínea pulsados, de tal forma que el número de línea pueda ser asociado con un mensaje de error. En algunos compiladores, el analizador léxico se encarga de hacer una copia del programa fuente con los mensajes de los errores detectados. Si el lenguaje fuente soporta algunas funciones de macro preprocesador, entonces éstas pueden ser también implementadas como analizadores lexicográficos.

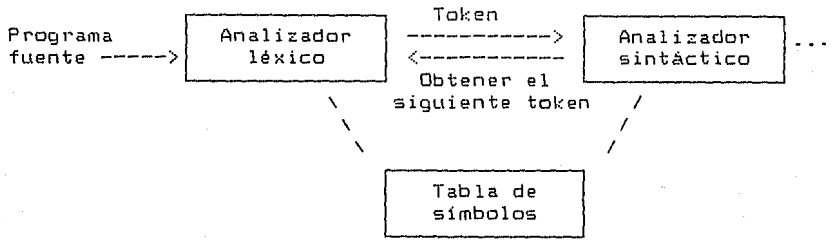


Fig. 2.1 Interacción del analizador léxico con el parser.

2.1.1 Razones de la división del análisis en análisis léxico y análisis sintáctico.

Existen varias razones para separar la compilación en las fases de análisis léxico y sintáctico.

1. La simplificación del diseño es quizás la consideración más importante. La separación del análisis léxico del análisis sintáctico, con frecuencia permite simplificar una o las otras tareas de estas fases.

2. Se mejora la eficiencia del compilador. El analizador léxico puede usar técnicas especializadas de almacenamiento intermedio para leer caracteres de entrada y aumentar significativamente la velocidad del procesamiento de tokens.

3. Se incrementa la portabilidad del compilador. Las particularidades del alfabeto de entrada y otras anomalias de dispositivos específicos pueden ser restringidas al analizador léxico. La representación de símbolos especiales o los símbolos no estándar, tales como ^ en Pascal, pueden ser aislados durante el análisis léxico.

2.1.2 Tokens, patrones y lexemas.

Los términos "token", "patrón" y "lexema" tienen significados específicos. La figura 2.2 muestra ejemplos de su uso. Un token es el símbolo resultante del análisis léxico que posteriormente será utilizado por el analizador sintáctico. En general, hay un conjunto de cadenas en el programa fuente para las que se produce el mismo token como salida. Este conjunto de cadenas se describe mediante una regla llamada patrón asociado al token. Se dice que el patrón hace un equiparamiento con cada cadena del conjunto. Un lexema es una secuencia de caracteres que comprenden un token del programa fuente. Esta secuencia de caracteres es equiparada con el patrón asociado a ese token. Por ejemplo, en la proposición en Pascal:

Const pi = 3.1416;

la subcadena pi es un lexema para el token "identificador".

Token	Ejemplos de lexemas	Descripción informal del patrón
const	const	const
if	if	if
relación	<, <=, =, <>, >, >=	< ó <= ó = ó <> ó >= ó >
id	pi, cont, valor	Letra seguida por letras o díg.
num		Cualquier constante numérica.

Fig. 2.2 Ejemplo de tokens.

En muchos lenguajes de programación, las siguientes construcciones son tratadas como tokens: palabras clave, operadores, identificadores, constantes, cadenas de literales, y símbolos de puntuación tales como paréntesis, punto y coma, coma, dos puntos y el punto. En el ejemplo anterior, cuando la secuencia de caracteres pi aparece en el programa fuente, un token que representa a un identificador es devuelto al parser. La devolución de un token es a menudo implementada pasando el entero correspondiente del token. Este es el entero al cual se hace referencia en la figura 2.2 como **id** en negritas.

Un patrón es una regla que describe el conjunto de lexemas que pueden representar un token particular en el programa fuente. El patrón para el token **const** en la figura 2.2 es la cadena const que denota la palabra clave. El patrón para el token **relación** es el conjunto de los seis operadores de relación del lenguaje Pascal.

2.2 DEFINICION DEL LENGUAJE MINI-PASCAL.

En esta sección damos una definición del modelo de lenguaje de programación para el cual desarrollaremos el analizador léxico, analizador sintáctico y el generador de código. Esta definición es el componente central para la escritura del compilador. Además, determina el conjunto de programas para los cuales el compilador va a generar código objeto equivalente. La definición toma la forma de una descripción sintáctica formal, usa la notación BNF ampliada. A la descripción sintáctica la acompaña una breve descripción semántica en lenguaje natural. El lenguaje definido, el cual llamaremos Mini-Pascal, es un subconjunto del lenguaje Pascal que se usará en esta tesis como un lenguaje apropiado para ilustrar las técnicas y problemas que se presentan en la construcción de compiladores.

2.2.1 Resumen del Lenguaje.

Un programa de computadora consiste de dos partes esenciales, una descripción de las acciones que van a efectuarse, y una descripción de los datos que serán manipulados por éstas. Las acciones son descritas por proposiciones y los datos por declaraciones.

Los datos están representados por los valores de las variables. Cada variable que ocurre en una proposición debe introducirse mediante una declaración de variable, la cual asocia un identificador y un tipo de dato a esa variable. Los tipos de datos esencialmente, definen el conjunto de valores que pueden ser asumidos por esa variable.

Los tipos de datos son los tipos estándar: Boolean e integer. También se permiten los tipos de datos arreglo y registro.

La proposición más fundamental es la proposición de asignación. Esta especifica que un nuevo valor calculado va ser asignado a la variable (o componente de la variable). El valor se obtiene mediante la evaluación de una expresión. Las expresiones consisten de variables, constantes y operadores. El conjunto de operadores de Mini-Pascal está subdividido de la siguiente forma:

1. Operadores aritméticos de adición, resta, inverso de un número, multiplicación y división.

2. Operadores Booleanos para negación, unión y conjunción.

3. Operadores relacionales de igualdad y desigualdad.

La transferencia de información de un dispositivo de entrada o salida se lleva a cabo mediante una proposición **read** o **write** dentro de un programa.

A una proposición se le puede dar un nombre (identificador), y ser referenciada mediante ese identificador. A esta proposición se le llama procedimiento, y su declaración es una declaración de procedimiento. Tal declaración puede contener adicionalmente, un conjunto de declaraciones de variables, o declaraciones de procedimientos, para formar un bloque. Puesto que los procedimientos pueden ser declarados dentro de los bloques que definen otros procedimientos, los bloques pueden anidarse. Esta estructura de bloques anidados determina el alcance de los identificadores que denotan variables, procedimientos, tipos y valores de constantes, y también determinan la vida de las variables.

2.2.2 Notación, terminología y vocabulario.

De acuerdo a la tradicional forma de Backus-Naur, las construcciones sintácticas se denotan por palabras encerradas entre pico paréntesis <>. Estas palabras también describen la naturaleza del significado de la construcción, y son usadas acompañando la descripción de la semántica. En nuestra BNF ampliada, la posible repetición de una construcción de cero o más veces se indica encerrando la construcción con llaves {}. El símbolo <cadena vacia> denota una secuencia nula de símbolos.

El vocabulario básico de Mini-Pascal consiste de símbolos básicos clasificados en letras, dígitos y símbolos especiales.

<Letra> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<Dígito> ::= 0|1|2|3|4|5|6|7|8|9

```
<Simbolos especiales> ::= + | - | * | = | <> | < | > | <= | >= |
                          ( | ) | [ | ] | := | . | , | ; | : | ..

                        | AND | ARRAY | BEGIN | CONST | DIV | DO
                        | ELSE | END | IF | MOD | NOT | OF | OR
                        | PROCEDURE | PROGRAM | RECORD | THEN
                        | TO | TYPE | VAR | WHILE

                        | INTEGER | BOOLEAN | FALSE | TRUE | READ
                        | WRITE
```

2.2.3 Identificadores.

Los identificadores denotan constantes, tipos, variables y procedimientos. Su asociación debe ser única dentro del bloque en el cual se declaran.

```
<Identificador>  ::= <Letra> {<Letra ó digito>}
<Letra ó digito> ::= <Letra> | <digito>
```

2.2.4 Constantes.

Las constantes son los valores particulares que las variables o los tipos básicos, Boolean e integer, pueden tomar.

```
<Constante entera> ::= <Digito> {<Digito>}
```

2.2.5 Tipos de datos.

Un tipo de dato determina el conjunto de valores que las variables de ese tipo pueden asumir.

Los tipos de datos entero y Boolean son los tipos estándar de Mini-Pascal.

```
<Tipo arreglo> ::= ARRAY[<Rango>] OF <Id. de tipo>
<Rango>        ::= <Constante>..<Constante>
```

2.2.6 Declaración y denotación de variables.

La declaración de variables consiste de una lista de variables seguidas por su tipo.

```
<Declaración de variable> ::= <id> {,<id>} : <Tipo>
```

Ejemplos:

```
X,Y,Z : INTEGER;
A     : ARRAY[1..10] OF INTEGER;
P,Q   : BOOLEAN;
```

Las denotaciones de variables designan una variable entera o una variable indexada.

<Variable> ::= <Variable ordinaria> | <Variable indexada>

Variables ordinarias.

Una variable entera se denota por su identificador.

<Variable ordinaria> ::= <Identificador de variable>

<Identificador de variable> ::= <Identificador>

Variables indexadas.

Un componente de un arreglo se denota por una variable seguida por una expresión de índice.

<Variable indexada> ::= <Variable arreglo> [<Expresión>]
<Variable arreglo> ::= <Variable ordinaria>

El tipo de la expresión de índice debe ser entero y su valor debe estar en el rango definido por el tipo arreglo.

Ejemplos: a[12], a[i+j]

2.2.7 Expresiones.

Las expresiones son construcciones que denotan reglas para cálculos a fin de obtener valores de variables y generar nuevos valores mediante la aplicación de operadores. Las expresiones consisten de operandos (esto es, variables y constantes) y operadores.

Las reglas de precedencia se reflejan en la siguiente sintaxis:

<Factor> ::= <Constante> | <Variable> | (<Expresión>) | **NOT** <Factor>

<Término> ::= <Factor> {<Op. multiplicador> <Factor>}
<Exp. simple> ::= <Signo> <Término> {<Op. sumador><Término>}
<Expresión> ::= <Exp. simple> [<Op. relacional> <Exp. simple>]

<Op. multiplicador> ::= * | DIV | MOD | AND

<Signo> ::= + | - | <cadena vacía>
<Op. sumador> ::= + | - | **OR**
<Operador relacional> ::= < | = | > | <= | >= | <>

Ejemplos:

Factores:

```
x
15
(x+y+z)
NOT p
```

Términos:

```
x*y
(x <= y) AND (y < z)
```

Expresiones simples:

```
-x
i*j+k
```

Expresiones:

```
x = 1
p <= q
(i<j) = (j<k)
```

2.2.8 Proposiciones.

Las proposiciones especifican acciones que la computadora va a ejecutar.

```
<Prop.> ::= <Prop. de asignación> | <Prop. procedure>
            <Prop. if> | <Prop. while> |
            <Prop. compuesta> | <cadena vacía>
```

2.3 PROGRAMACION DEL ANALIZADOR LEXICO.

La función del analizador léxico, implementada por el procedimiento Explora_simbolo, está definida por las reglas sintácticas dadas en la definición del lenguaje para símbolos especiales, identificadores y constantes.

Estas reglas sintácticas las podemos escribir de la siguiente forma:

```
<símbolo especial> ::= <Identificador o palabra reservada>
                     | <Constante entera> |
                     | <>  | <=  | <
                     | :   | ...
```

en la cual podemos ver que las alternativas de cada línea se distinguen completamente de las otras líneas por el primer carácter involucrado, y que la ocurrencia de cualquier otro carácter diferente a los que se muestran, representa un símbolo ilegal. Esto nos conduce inmediatamente a una estructura de código para el proceso de exploración de símbolos especiales de la siguiente forma:

```
CASE car OF
  'A'..'Z' : Explora identificador o palabra reservada;
  '0'..'9' : Explora constante entera;
  '<'      : Explora símbolo <>, <=, ó <;
  '>'      : Explora símbolo >= ó >;
  ':'      : Explora símbolo := ó :;
  '+'      : Explora símbolo +;
   ...     ...
otros caracteres : Explora símbolo ilegal
```

2.3.1 Interface entre el analizador léxico y el parser.

Una interface adecuada entre el analizador léxico y el analizador sintáctico es que el analizador léxico ponga a disposición del analizador sintáctico el símbolo "actual" en la cadena de símbolos de entrada, unida a la habilidad de reemplazar este símbolo por el siguiente símbolo cuando el analizador sintáctico lo requiera.

Un símbolo puede ser una palabra reservada, un identificador, una constante, o uno de los símbolos especiales que se mencionan en la definición del lenguaje. Por lo tanto, podemos definir el rango de "valores" que un símbolo puede tomar como un tipo enumerado:

```
t_simb = (AND1, ARRAY1, BEGIN1, CONST1, DIV1, DO1, ELSE1,
          END1, IF1, MOD1, NOT1, OF1, OR1, PROCEDURE1,
          PROGRAM1, RECORD1, THEN1, TO1, TYPE1, VAR1,
          WHILE1,
                        { Identificadores estándar }

           INTEGER1,BOOLEAN1, FALSE1,TRUE1,READ1,WRITE1,

                        { Operadores }

          ASIGNACION, MAS, MENOS, POR, DIVISION, IGUAL,
          DIFERENTE, MENOR_QUE, MENOR_O_IGUAL, MAYOR,
          MAYOR_O_IGUAL,

                        { Otros }

          NUEVALINEA, APOSTROFO, IDENT, CTE_ENT,
          CTE_TIPO_CHAR, CORCHETE_IZQ, PARENT_IZQ,
          PARENT_DER,CORCHETE_DER, COMENT, COMA,PUNTO,
          PUNTOYCOMA, DOSPUNTOS, PUNTOPUNTO, OTRO);
```

La interface sugerida por el analizador léxico será establecida por una variable que represente el símbolo actual:

```
simbolo : t_simb;
```

y un procedimiento que reemplace el símbolo actual por el símbolo siguiente:

```
Procedure Explora_simbolo;
```

El analizador sintáctico, el cual recibe estos símbolos, puede posteriormente reportar los errores de sintaxis detectados en el programa fuente.

2.3.2 Manejo de errores.

Algunas de las fases requerirán que el compilador reporte algunos errores. Para este propósito introducimos el siguiente tipo de datos:

```
tipo_error = (dup, coment_inc, tipo_id_inc, cte_inc,
              rango_inc, error_sintáctico, tipo_inc,
              id_indefinido, id_inc, desconocido)
```

Cuando una de las fases descubre un error, ésta llama al procedimiento llamado error.

```
PROCEDURE Error(tipo_error : t_error);
VAR
  texto : STRING[35];
BEGIN
  IF NOT ha_habido_errores THEN
  BEGIN
    CLOSE(temp1);
    todo_bien := FALSE;
    ha_habido_errores := TRUE
  END;

  CASE tipo_error OF
     coment_inc : texto := 'Comentario inválido ';
     cte_inc    : texto := 'Constante entera fuera de rango';
     rango_inc  : texto := 'Indice de rango inválido';
     id_inc : texto := 'Nombre de identificador muy grande';
  END;
  IF lin_correcta THEN
  BEGIN
    WRITELN(notas);
    WRITELN(notas,'* Error en linea ',num_lin,':  ',texto);
    WRITELN(notas);
    num_errs := num_errs + 1;
    lin_correcta := FALSE
  END
END;
```

Algoritmo 2.1 Manejo de mensajes de error.

El programa del analizador generará mensajes de error legibles en un archivo llamado TEMP1.LST y desplegará el mensaje:

n errores encontrados durante el análisis léxico

al final del proceso del análisis léxico.

Al iniciar la compilación, aún no hay errores y el compilador está listo para emitir código. En el momento que el compilador encuentra el primer error en el programa, éste cierra el archivo de código intermedio, TEMP1.COD, y genera los mensajes de error en el archivo llamado TEMP1.LST.

Un sólo error algunas veces causa que el compilador produzca varios errores redundantes. Los mensajes innecesarios son suprimidos por un método efectivo. Al principio de cada línea del programa fuente, las fases de análisis llaman a un procedimiento para registrar un número de línea e indicar que aún no se han encontrado errores en la nueva línea:

```
VAR
  num_lin       : integer;
  lin_correcta : Boolean;

PROCEDURE Nueva_lin(num:INTEGER);
BEGIN
  num_lin := num;
  lin_correcta := TRUE
END;
```

Cuando un error es reportado, se genera un mensaje y la línea es marcada como incorrecta para suprimir mensajes adicionales relacionados a esa línea (algoritmo 2.1).

2.3.3 Código intermedio.

Cuando el analizador léxico reconozca una palabra reservada, tal como begin, éste emitirá el valor correspondiente al simbolo BEGIN1:

Emite(BEGIN1)

Un identificador es generado como el símbolo IDENT seguido por un entero. El entero es el índice del identificador. Los identificadores estándar tienen índices fijos. A los otros identificadores se les asignan índices consecutivos en el orden que aparecen en el programa fuente.

Al principio de cada línea, el analizador léxico genera el símbolo NUEVALINEA seguido por un número de línea.

Cuando el analizador léxico detecta una secuencia de caracteres desconocida, éste genera el símbolo OTRO. El analizador sintáctico reportará un error de sintaxis al momento de recibir este símbolo.

En el instante que el analizador léxico alcance el fin del programa fuente, generará el símbolo PUNTO y el proceso terminará.

2.3.4 Búsqueda.

EL analizador léxico construye una tabla de todas las palabras usadas en un programa fuente. Cada entrada de la tabla de símbolos describe una palabra con los siguientes atributos:

(1) La secuencia de caracteres de la palabra.
(2) Un valor Boolean que indica si ésta es un identificador o palabra reservada.
(3) Un índice de identificador o el valor ordinal de una palabra reservada.

Inicialmente, el analizador léxico coloca todas las palabras reservadas y los identificadores estándar en la tabla de símbolos. Cuando se ha introducido una palabra, el analizador léxico primero trata de encontrarla en la tabla. Si no se encuentra, ésta es insertada y descrita como un identificador con un nuevo índice de identificador.

2.3.5 Hashing.

La técnica de transformación de claves conocida como hashing, nos permite organizar una tabla de símbolos de tal forma que con un sólo acceso se pueda obtener el identificador o palabra reservada almacenada.

Para usar hashing debemos decidir cuantas listas de palabras necesitamos. Sea este número N. Después debemos definir una función H que transforme cualquier palabra en un número de 1 a N. El número H(w), la clave hash, se usa como índice para seleccionar la lista a la que pertenece la palabra w.

El objetivo es encontrar una función hash que distribuya las palabras entre las listas. Si fuera posible colocar cada palabra en una lista diferente, el número de comparaciones sería una por palabra. Esto obviamente requiere que el número de palabras M no exceda al número de listas N.

La función hash debe elegirse con cuidado para reducir las colisiones en la tabla de símbolos. El algoritmo 2.2 trabaja bien. Suma los caracteres individuales de una palabra, divide la suma

entre la longitud de la palabra y usa el residuo como clave hash. Es conveniente que la longitud de la tabla sea un número primo.

```
FUNCTION cve_hash(texto:cad; long:INTEGER) : INTEGER;
CONST
  w = 32641;  { 32768 - 127 }
  n = num_cves;
VAR
  sum, i : INTEGER;

BEGIN
  sum := 0;
  i:= 1;
  WHILE i <= long DO
  BEGIN
     sum := (sum + ORD(texto[i])) MOD w;
     i := i + 1
  END;
  cve_hash := sum mod n+1;
END;
```

Algoritmo 2.2. Cálculo de la clave hash.

2.3.6 Tabla de símbolos.

Antes de implementar la tabla de simbolos, tenemos que tomar una decisión. ¿Cómo almacenamos los caracteres de cada palabra en la tabla de símbolos?.

En los casos que los identificadores pueden tener una gran longitud, como por ejemplo en Pascal, conviene tener los nombres de los identificadores en una tabla de lexemas.

La figura 2.3 muestra una forma eficiente para almacenar los caracteres de todas las palabras en una tabla separada, conocida como tabla de lexemas.

La tabla hash es un arreglo de punteros:

```
CONST
  num_cves = 631;
TYPE
  Ptro_a_palabra  = ^Reg_palabra;
  TablaHash = ARRAY[1..num_cves] OF Ptro_a_palabra;

VAR
  Hash : TablaHash;
```

Si una lista de palabras está vacia, el puntero correspondiente de la tabla hash tiene el valor nil; en caso contrario, éste apunta a la primer palabra registrada en la lista.

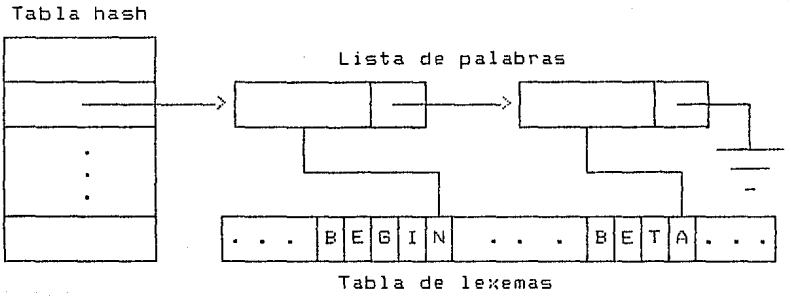


Fig. 2.3 Tabla de símbolos.

Cada palabra en una lista está descrita por un registro que define si la palabra es un identificador o una palabra reservada.

```
TYPE
  Reg_palabra = RECORD
                  palabra_sig : Ptro_a_palabra;
                  es_id       : BOOLEAN;
                  indice,long,
                  ult_car     : INTEGER;
                END;
```

El índice de una palabra es el valor ordinal de una palabra reservada o el número correspondiente a un identificador.

La tabla de lexemas es un arreglo de caracteres:

```
CONST
  num_car  = 5000;
TYPE
  arr_lexemas = ARRAY[1..num_car] OF CHAR;
VAR
  lexemas : arr_lexemas;
```

El algoritmo 2.3 define la inserción de una nueva palabra en la tabla de símbolos. El procedimiento verifica_lim checa que haya espacio para una nueva palabra en la tabla de lexemas.

```
PROCEDURE inserta(es_id : BOOLEAN; texto : cad;
                  long, indice, num_cve : INTEGER);
VAR
  Ptro : Ptro_a_palabra;
  m,n  : INTEGER;
```

```
BEGIN      { Inserta la palabra en la tabla de lexemas }
  caracteres := caracteres + long;
  verifica_lim(caracteres,num_car);
  m := long;
  n := caracteres - m;
  WHILE m > 0 DO
  BEGIN
     lexemas[m+n] := texto[m];
     m := m - 1
  END;
     { Inserta la palabra en la lista de palabras }
  NEW(Ptro);
  Ptro^.palabra_sig := hash[num_cve];
  Ptro^.es_id       := es_id;
  Ptro^.indice      := indice;
  Ptro^.long        := long;
  Ptro^.ult_car     := caracteres;
  hash[num_cve]     := Ptro
END;
```

Algoritmo 2.3. Inserción de una nueva palabra en la tabla de símbolos.

Las palabras reservadas y los identificadores estándar son insertados por medio del algoritmo 2.4.

```
PROCEDURE Instala(es_id:BOOLEAN; texto:cad; long,indice :INTEGER);
BEGIN
  inserta(es_id, texto, long, indice, cve_hash(texto,long))
END;
```

Algoritmo 2.4.

Este procedimiento es llamado como sigue:

```
     { Inserta la palabras reservadas }

  Instala(FALSE, 'AND',       3,ORD(AND1));
  Instala(FALSE, 'ARRAY',     5,ORD(ARRAY1));
                  ...
  Instala(FALSE, 'WHILE',     5,ORD(WHILE1));

  { Inserta los identificadores estándar }

  Instala(TRUE, 'INTEGER',  7,ORD(INTEGER1));
  Instala(TRUE, 'BOOLEAN',  7,ORD(BOOLEAN1));
                  ...
  Instala(TRUE, 'WRITE',    5,ORD(WRITE1));
```

El analizador léxico usa el algoritmo 2.5 para buscar una palabra particular en la tabla de símbolos. Se usa la clave hash de la palabra para seleccionar una lista en la cual se buscará

ésta. Si se encuentra, se recuperará de la tabla el valor del símbolo; en caso contrario, la palabra se instalará en la lista y se describirá como un identificador.

```
PROCEDURE busca(texto:cad; long: INTEGER; VAR es_id : BOOLEAN;
                VAR indice : INTEGER);
VAR
  num_cve    : INTEGER;
  Ptro       : Ptro_a_palabra;
  realizado : BOOLEAN;

BEGIN
  num_cve    := cve_hash(texto,long);
  Ptro       := hash[num_cve];
  realizado := FALSE;
  WHILE NOT realizado DO
  IF Ptro = nil THEN
  BEGIN
     { La lista no contiene la palabra }
     es_id  := TRUE;
     num_ids   := num_ids + 1;
     indice    := num_ids;
     inserta(TRUE, texto, long, indice, num_cve);
     realizado := TRUE
  END
  ELSE
    IF encont(texto,long,Ptro) THEN
    BEGIN
       { La lista contiene la palabra }
       es_id := Ptro^.es_id;
       indice    := Ptro^.indice;
       realizado := TRUE
    END
    ELSE
       { La lista ya contiene otra palabra }
       Ptro := Ptro^.palabra_sig
END;
```

Algoritmo 2.5. Búsqueda de una palabra en la tabla de símbolos.

El algoritmo 2.6 determina si una palabra dada es o no la misma que una palabra de la tabla de símbolos. Para acelerar el algoritmo, las dos palabras son comparadas sólo si tienen la misma longitud.

```
FUNCTION encont(texto : cad; long : INTEGER;
                Ptro : Ptro_a_palabra):BOOLEAN;
VAR
   misma : BOOLEAN; m,n : INTEGER;
```

```
BEGIN
   IF Ptro^.long <> long THEN misma := FALSE
   ELSE
     BEGIN
       misma := TRUE; m := long; n := ptro^.ult_car - m;
       WHILE (misma = TRUE) AND ( m > 0) DO
       BEGIN
         misma := texto[m] = lexemas[m+n];
         m := m - 1
       END;
     END;
   encont := misma
END;
```

Algoritmo 2.6. Verifica si una palabra dada ya existe en la tabla de simbolos.

2.3.7 Prueba.

Cada fase del compilador se prueba permitiéndole compilar pequeños programas construidos específicamente para prueba. La salida del programa de prueba consiste de código intermedio impreso como una secuencia de enteros.

En la práctica, es fácil encontrar un conjunto sistemático de casos de prueba que obliguen al compilador a ejecutar cada proposición al menos una vez.

Cuando observemos en detalle el procedimiento busca, debemos construir casos de prueba para ejecutar todas las proposiciones de éste. El cuerpo del ciclo busca tiene forma:

```
if ptro = nil then

   { 1: La lista no contiene la palabra dada. }

else

   if encont(...) then

      { 2: La lista contiene la palabra. }

   else

      { 3: La lista ya contiene otra palabra. }
      ptro := ptro^.palabra_sig;
```

La siguiente secuencia de palabras cubre los tres casos de prueba:

and dna dna

Debido a que la suma de caracteres módulo W es una operación conmutativa, las palabras anteriores tienen la misma clave hash, puesto que éstas son permutaciones de las mismas letras. Después que el analizador léxico inicializa la tabla de símbolos, éste inserta la palabra and. Cuando encuentra la misma palabra en el programa de prueba, la palabra ya está en la tabla (caso 2). En el momento que el identificador dna es introducido por primera vez, la lista ya contiene otra palabra (la palabra and - caso 3), pero aún no contiene el identificador dna (caso 1).

El apéndice A.2 muestra el listado del analizador léxico para programas escritos en Mini-Pascal.

El programa Prueba1 (ver apéndice A.4) se construyó especialmente para usarlo como entrada para el analizador léxico. La salida proporcionada por el analizador léxico para este programa es la siguiente:

```
1    { Mini-Pascal. Prueba1: Símbolos correctos }
2    Program prueba1;
3
4     and array begin const div do else
5     end if mod not of or procedure
6     program record then type var while
7
8     { Identificadores estándar }
9
10     integer Boolean false true read write
11
12     dna dna
13
14     { { Comentario } }
15
16     alfa1 x1 x2
17
18     0 32767
19
20     + - * / .
21
22     = < <= > >= < > :=
23     () [] , : ; ..
24     ' ( )
25     .
       Análisis lexico terminado sin errores

       Número de identificadores usados: 5
```

CAPITULO 3. ANALISIS SINTACTICO.

INTRODUCCION.

La segunda fase en el proceso de compilación de un programa fuente es el análisis sintáctico al que, en terminología anglosajona, suele denominarse parser. Analizar sintácticamente una cadena de tokens es encontrar para ella un árbol sintáctico o de derivación que tenga como raíz el axioma de la gramática de contexto libre mediante la aplicación sucesiva de sus reglas de derivación. En caso de éxito se dice que la cadena pertenece al lenguaje generado por la gramática y puede proseguirse con el proceso de compilación. En caso contrario, se dice que la cadena a analizar no pertenece al lenguaje.

De la forma de construir dicho árbol se desprenden dos clases de análisis sintáctico: ascendente y descendente.

El análisis sintáctico ascendente intenta construir un árbol de reconocimiento sintáctico para una cadena de entrada, empezando por los terminales y finalizando en la raíz.

Existen varios métodos de reconocimiento ascendente: reducción-desplazamiento, precedencia simple, precedencia de operador, y lenguaje de producciones.

Los analizadores sintácticos descendentes van construyendo el árbol.sintáctico de la proposición a reconocer de una forma descendente: inician por la raíz y llegan a los terminales de la proposición en cuestión.

Las técnicas más importantes de análisis descendente son las siguientes:

- El descenso recursivo.
- Parsing LL(1) con tabla.

Dado el caracter eminentemente práctico del descenso recursivo, en este Capítulo desarrollaremos un parser basado en esta técnica para analizar programas escritos en Mini-Pascal. También se discutirá el problema de la recuperación de errores de sintaxis.

3.1 DESCENSO RECURSIVO.

Una de las formas de reconocimiento sintáctico más usadas hoy en dia es la de descenso recursivo. Con este nombre se quiere indicar que se realiza una construcción descendente del árbol sintáctico como se indicó anteriormente. En cuanto al calificativo de recursivo, será muy fácil de aprovechar por el diseñador cuando el parser se programe con un lenguaje de programación de alto

nivel que permita la programación de procedimientos recursivos (tal como Pascal, C, PL/1, ALGOL), lo cual es la tendencia actual en la escritura de compiladores.

Comenzaremos definiendo la sintaxis del lenguaje Mini-Pascal, clasificada en orden descendente recursivo. Los siguientes símbolos constituyen metasímbolos que pertenecen al formalismo BNF y no al lenguaje Mini-Pascal.

::= Significa "se define como".
{} significa que el contenido se puede repetir cero o más veces.
<> Significa que el contenido es una construcción sintáctica BNF.
| Significa "o".

El resto de los símbolos forma parte del lenguaje. Cada construcción sintáctica aparece entre picoparéntesis, por ejemplo: <Bloque> y <Prop. compuesta>. Las palabras reservadas de Mini-Pascal aparecen en negritas.

```
<PROGRAMA> ::= PROGRAM <Identificador>; <Bloque>.
<Bloque>   ::= [<Parte de def. de constantes>]
               [<Parte de def. de tipos>]
               [<Parte de def. de variables>]
               {<Def. de procedimiento>}
               <Prop. compuesta>

<Parte de def. de constantes> ::= CONST
                                    <Def. de constante>
                                    {<Def. de constante>}

<Def. de constante> ::= <Id. de constante> = <Constante>;

<Parte de def. de tipos> ::= TYPE <Def. de tipo>
                                  {<Def. de tipo>}

<Def. de tipo>     ::= <Id. de tipo> = <Tipo>;
<Tipo>             ::=  <Tipo arreglo> | <Tipo registro>
<Tipo arreglo>     ::= ARRAY[<Rango>] OF <Id. de tipo>
<Rango>            ::= <Constante>..<Constante>
<Tipo registro>    ::= RECORD <Lista de campos> END;
<Lista de campos>  ::= <Sección de regs>
                         {;<Sección de regs>}

<Sección de regs>  ::= <Id. de campo>
                       {,<Id. de campo>} : <Id. de tipo>

<Parte de def. de variables> ::= VAR <Def. de variable>
                                     {<Def. de variable>;}
                                     | <Cadena vacía>
```

```
<Def. de variable> ::= <Lista de variables>;
<Lista de variables> ::= {,<Id. de variable>} : <Id. de tipo>
<Def. de procedimiento> ::= PROCEDURE <Id. de proc.>
                                    <Bloque del proc.>

<Bloque del proc.> ::= [(<Lista de parámetros formales>)];
                       <Bloque>

<Lista de parámetros formales> ::= <Def. de parámetro>
                                   {;<Def. de parámetro>}

<Def. de parámetro> ::= [VAR] <Lista de variables>

<Prop.> ::= <Prop. de asignación> | <Prop. procedure> |
            <Prop. if> | <Prop. while> | <Prop. compuesta> |
            <cadena vacía>

<Prop. de asignación> ::= <variable> := <Expresión>
<Prop. procedure>     ::= <Identificador de procedimiento>
                          [(<Lista de parámetros actuales>)]

<Lista de parámetros actuales> ::= <Parámetro actual>
                                   {,<Parámetro actual>}

<Parámetro actual> ::= <Expresión> | <Variable>
<Prop. if>         ::= IF <Expresión> THEN <Prop.>
                       [ELSE <Prop.>]

<Prop. while>      ::= WHILE <Expresión> DO <Prop.>
<Prop. compuesta>  ::= BEGIN
                         <Prop.> {;<Prop.>}
                       END

<Expresión> ::= <Exp. simple> {<Op. relacional> <Exp. simple>}
<Exp. simple> ::= <Signo> <Término> {<Op. sumador><Término>}
<Op. relacional> ::= < | = | > | <= | >= | <>
<Signo>       ::= + | - | <cadena vacía>
<Op. sumador> ::= + | - | OR
<Término>     ::= <Factor> {<Op. multiplicador> <Factor>}
<Op. multiplicador> ::= * | DIV | MOD | AND
<Factor>      ::=  <Constante> | <Variable> | (<Expresión>) |
                   NOT <Factor>

<Variable>    ::= <Id. de variable> | {<Selector>}
<Selector>    ::= <Selector de indice> | <Selector de campo>
<Selector de indice>::= [<Expresión>]
<Selector de campo> ::= . <Id. de campo>
<Constante>         ::= <Constante entera> | <Id. de constante>
<Constante entera>  ::= <Dígito> {Dígito}
<Identificador>     ::= Letra {Letra | Dígito}
```

3.2 CONSTRUCCION DEL ANALIZADOR SINTACTICO PARA MINI-PASCAL.

En la programación del analizador sintáctico usaremos el método de un sólo símbolo delantero. Este método se implementa de la siguiente forma:

1) Se le manda al analizador sintáctico el primer símbolo explorado en el programa fuente.

2) Cuando el analizador sintáctico ha reconocido el símbolo como parte de una construcción sintáctica particular, inmediatamente acepta el siguiente símbolo explorado.

3.2.1 Algoritmos.

En el Capítulo anterior intuitivamente, traducimos las reglas sintácticas para símbolos en fragmentos equivalentes de código. Podemos usar una técnica similar para construir un analizador sintáctico a partir de las reglas sintácticas del lenguaje. De esta forma, de la regla sintáctica para <PROGRAMA>:

<PROGRAMA> ::= **PROGRAM** <Identificador>; <Bloque>.

podemos formular un procedimiento para el análisis de un programa:

```
PROCEDURE Programa;
BEGIN
  Acepta(PROGRAM1);
  Acepta(IDENT);
  Acepta(PUNTOYCOMA);
  Bloque;
  Acepta(PUNTO);
END;
```

donde acepta es un procedimiento que verifica que el símbolo actual sea el especificado y explora el siguiente, en cualquier otro caso reporta un error sintáctico:

```
PROCEDURE Acepta(simbolo_esperado: t_simb; Stop: simbolos);
BEGIN
  IF simbolo = simbolo_esperado
  THEN
    Explora_simbolo
  ELSE
    Error_de_sintaxis(Stop);
END;
```

Algoritmo 3.2.1 Verifica el símbolo actual.

El procedimiento para el análisis de un bloque se deriva en forma similar de la regla sintáctica correspondiente:

```
<Bloque> ::= [<Parte de def. de constantes>]
             [<Parte de def. de tipos>]
             [<Parte de def. de variables>]
             {<Def. de procedimiento>}
             <Prop. compuesta>
```

la cual se puede expresar como:

```
PROCEDURE Bloque;
BEGIN
  IF simbolo = CONST1 then Parte_de_def_ctes;
  IF simbolo = TYPE1 then Parte_def_tipos;
  IF simbolo = VAR1 then Parte_def_var;
  WHILE simbolo = PROCEDURE1 DO
     Def_de_procedimiento;
  Prop_compuesta;
END;
```

Algoritmo 3.2.2 Análisis de un bloque.

De esta forma podemos continuar desarrollando procedimientos de análisis sintáctico, uno para cada regla sintáctica de la definición del lenguaje. Sin embargo, es claro que cada procedimiento desarrollado es una traducción directa de la regla sintáctica correspondiente. Por lo tanto, antes de continuar, formularemos un conjunto de reglas para el proceso de traducción. Cada regla sintáctica tiene la forma:

<Construcción sintáctica> ::= forma permitida

donde la forma permitida está expresada en términos de:

a) Los símbolos básicos del lenguaje, los cuales denotaremos por las letras minúsculas a,b,...,z;

b) Otras construcciones sintácticas <A>, <B>,...,<Z>;

c) Los metasímbolos | y {} denotando selección y posible repetición.

Nuestro objetivo es traducir la regla sintáctica para cada construcción sintáctica en un procedimiento, cuya acción sea analizar la secuencia de símbolos resultantes, y verificar que sean la forma permitida.

Podemos ilustrar nuestro proceso de traducción como la conversión de una regla sintáctica:

<S> ::= α

a un procedimiento equivalente:

```
PROCEDURE S;
BEGIN
  T(α)
END;
```

La transformación T está definida por las reglas siguientes:

1. Si la forma α es un símbolo único en el lenguaje, la acción requerida es inspeccionar el símbolo actual de entrada y si es el símbolo permitido, entonces explorar el siguiente, en caso contrario reportar un error. Asumiendo que el procedimiento acepta es el que definimos anteriormente, nuestra primer regla de transformación es:

Regla 1: T(α) ----> acepta(α)

2. Si la forma permitida es una sola construcción sintáctica, por ejemplo <A>, la acción requerida es simplemente una llamada al procedimiento A correspondiente. Así, la regla 2 es:

Regla 2. T(<A>) ----> A

3. Si la forma permitida es una secuencia de símbolos y construcciones sintácticas, la acción requerida es la secuencia correspondiente de acciones apropiadas:

Regla 3. $T(\alpha_1\ \alpha_2 \ldots \alpha_n)$ ----> begin
$T(\alpha_1)$
$T(\alpha_2)$
...
$T(\alpha_n)$
end;

4. Si la forma permitida consiste de un número de alternativas $\alpha|\beta|\ldots|\xi$, la acción requerida es alguna selección entre las acciones apropiadas a cada alternativa:

```
case ? of
     ? : T(α)
     ? : T(β)
       ...
     ? : T(ξ)
```

¿En base a qué se hace la selección? En el analizador léxico la decisión correspondiente fue hecha sobre el valor del carácter del símbolo bajo exploración. En forma similar, aquí la elección debería hacerse tomando en consideración el símbolo actual de

entrada. Si definimos los símbolos que pueden iniciar una secuencia de símbolos de la forma α como iniciadores(α), la transformación necesaria sería:

```
Regla 4. T(α|β|...|Ω) ----> case simbolo of
                                iniciadores(α) : T(α)
                                iniciadores(β) : T(β)
                                    ...
                                iniciadores(Ω) : T(Ω)
                            end;
```

Esta regla está sujeta a las siguientes condiciones:

a) Ningún símbolo puede ser un iniciador de más de una de las alternativas de cada forma permitida.

La acción del analizador para una alternativa <cadena vacía> de una forma permitible es no aceptar símbolos. Esta acción debería considerarse cuando el siguiente símbolo sea un seguidor de la forma permitida, esto es:

```
T(α|β|...|Ω| <cadena vacía>) ----> case simbolo of
                                       iniciadores(α) : T(α)
                                       iniciadores(β) : T(β)
                                           ...
                                       seguidores(S);
                                   end;
```

De esta forma obtenemos la segunda condición:

b) Ningún símbolo puede ser un posible iniciador y un posible seguidor de una forma permitida que tenga una alternativa vacía.

La construcción case usada en la regla 4 expresa una elección entre cualquier número de alternativas. Si sólo existen dos, pueden ser expresadas como if...then...else, y si una de éstas es la cadena vacía, esto se reduce a if...then.

5. Si la forma permitida involucra una posible repetición, {}, la acción requerida es un ciclo. Al igual que en el analizador léxico, el criterio para la terminación del ciclo está basado en el símbolo actual, de esta forma podemos expresar la regla 5:

```
Regla 5. T({α}) ----> while simbolo in iniciadores(α) do
                          T(α)
```

Las reglas 1 a 5 permiten la traducción del conjunto de reglas sintácticas que definen un lenguaje en un conjunto equivalente de procedimientos sintácticos.

El analizador opera de una manera determinística, determina la trayectoria de análisis apropiada mediante una inspección del símbolo actual de entrada, siempre que las reglas sintácticas subyacentes cumplan las condiciones (a) y (b). Un conjunto de reglas sintácticas que reunen estas condiciones constituyen lo que se conoce como una gramática LL(1).

3.2.2 Recuperación de errores.

Las acciones realizadas por el compilador después de descubrir un error de sintaxis se conocen como recuperación de errores. El propósito de la recuperación de errores es permitir que el compilador pueda continuar con el análisis del programa para que encuentre tantos errores como sea posible y asegurar que cada error sea reportado sólo una vez.

Nuestro enfoque para la recuperación de errores será saltar cero o más símbolos hasta que el parser alcance un símbolo mayor que pueda reconocer. A estos símbolos les llamaremos símbolos de parada y entre ellos están los siguientes: CONST1,TYPE1,VAR1, PROCEDURE1, BEGIN1, IF1, WHILE1, PUNTO.

El parser usa el algoritmo 3.2.3 para realizar la recuperación de errores después de detectar un error de sintaxis.

```
PROCEDURE Error_de_sintaxis(stop : simbolos);
BEGIN
  Error(error_sintactico);
  WHILE NOT (simbolo in stop) DO
        Explora_simbolo
END;
```

Algoritmo 3.2.3 Recuperación de errores de sintaxis.

La idea de saltar símbolos hasta que el parser alcance un símbolo de parada es fina, pero si usamos un número pequeño de símbolos de parada, el compilador saltará demasiados símbolos después de un error de sintaxis. Como consecuencia, no serán analizadas varias sentencias. Por ejemplo, después de detectar el error de sintaxis en la línea cuatro del programa Prueba3 (ver apéndice A.4), el parser saltara todas las otras definiciones de constantes hasta que alcance la palabra type, la cual es un símbolo de parada:

```
3  const
4     a := 1;
5     b = 2;
6     c. = ;
7     d = 4;
8  type
      ...
```

Para que esto no suceda, podemos ampliar temporalmente el conjunto de símbolos de parada con el símbolo correspondiente al token punto y coma, en este caso, y con otros símbolos según las sentencias a analizar.

Una vez que tenemos la idea de usar conjuntos de símbolos de parada para analizar diferentes tipos de sentencias, el siguiente paso es incluir esta idea en la construcción del parser, para ello usaremos las siguientes reglas:

i) Para cada regla BNF:

N ::= E

el parser define un procedimiento del mismo nombre:

```
PROCEDURE N(stop : simbolos);
BEGIN
   Acepta(E,Stop)
END;
```

ii) Cuando el parser espera un símbolo único s seguido por un símbolo de parada, éste llama al procedimiento acepta.

```
PROCEDURE acepta(s : t_simb; stop : simbolos);
BEGIN
  IF simbolo = s THEN
     Explora_simbolo
  ELSE
     Error_de_sintaxis(stop);
  Verifica_sintaxis(stop);
END;
```

Después de examinar el símbolo actual, el parser siempre se asegura que el siguiente símbolo sea uno de los símbolos de parada esperados. Esta verificación se realiza mediante el algoritmo 3.2.4.

```
PROCEDURE Verifica_sintaxis(stop : simbolos);
BEGIN
  IF NOT (simbolo in stop) THEN
     error_de_sintaxis(stop)
END;
```

Algoritmo 3.2.4 Verifica sintaxis del símbolo siguiente.

Después de un error de sintaxis, el parser salta la entrada hasta que alcance uno de los símbolos de parada (algoritmo 3.2.5).

```
PROCEDURE Error_de_sintaxis(stop : simbolos);
BEGIN
  Error(error_sintactico);
  WHILE NOT(simbolo in stop) DO
        Explora_simbolo;
END;
```

Algoritmo 3.2.5 Salta símbolos hasta leer un símbolo de parada.

iii) Para reconocer una sentencia descrita por una regla BNF denominada N, el parser llama al procedimiento correspondiente N usando los símbolos de parada de la sentencia como un parámetro:

```
Acepta(N,stop);
```

El único símbolo que puede seguir después de un programa completo es el punto. De esta manera, inicialmente éste es el único símbolo de parada. Cuando el parser llama otros procedimientos, tales como acepta y bloque, éste agrega los símbolos adicionales a los símbolos de parada sin remover los anteriores.

3.2.3 Código intermedio.

El código intermedio consiste de símbolos representados por valores de enumeración de tipo t_simb. A algunos símbolos le sigue un argumento entero n:

ID n NUEVALINEA n CTE_ENT n

El código intermedio es, por lo tanto, una secuencia de valores de símbolos y enteros.

El objetivo de los números de línea es ayudar a localizar errores, no forman parte de las sentencias de Mini Pascal.

3.2.4 Prueba.

Es conveniente escribir dos programas de prueba diferentes que obliguen al analizador sintáctico a ejecutar todas las proposiciones al menos una vez: uno que pruebe el análisis de sentencias correctas y otro que pruebe la detección de errores de sintaxis (ver Prueba2 y Prueba3 respectivamente, en el apéndice A.4).

La salida para estos programas de prueba es la siguiente:

```
1   { Mini-Pascal. Prueba2: Análisis sintáctico }
2   Program prueba2;
3   const
4      a = 1;
5      b = a;
6   type
7      T = array [1..2] of integer;
8      U = record
9            f,g : integer;
10           h   : boolean
11         end;
12     V = record
13           f : integer
14         end;
15  var
16     x,y : T;
17     z : U;
18
19     procedure P(var x:integer; y : boolean);
20     const
21        a = 1;
22        procedure Q(x : integer);
23        type
24           T = array [1..2] of integer;
25        begin
26           x := -1;
27           x := x;
28           x := (2 - 1) * (2 + 1) div 2 mod 2;
29           if x < x then
30              while x = x do Q(x);
31           if x > x then
32              while x <= x do P(x,false)
33           else
34              if not (x <> x) then { vacio }
35        end;
36     begin
37        if x >= x then y := true
38     end;
39
40     procedure R;
41     var
42       x : T;
43     begin
44        x[1] := 5
45     end;
46  begin
47     z.f := 6
48  end.
```

```
Análisis sintáctico terminado sin errores
Número de identificadores usados: 15
```

```
1    { Mini-Pascal. Prueba3: Errores sintácticos }
2    program Test3;
3    const
4       a := 1;

* Error en linea 4:  Error de sintaxis

5       b = 2;
6       c = ;

* Error en linea 6:  Error de sintaxis

7       d = 4;
8    type
9       s = recrod

* Error en linea 9:  Error de sintaxis

10              f, g : integer
11           end;
12        T = array [1..2] of integer;
13   var
14        x : integer;
15   begin
16        if = 2 then

* Error en linea 16:  Error de sintaxis

17          x := 1
18   end.

     4 Error(es) encontrado(s)
     Número de identificadores usados: 11
```

CAPITULO 4. ANALISIS DE ALCANCE.

INTRODUCCION.

Este Capítulo define las reglas de alcance de Mini-Pascal y explica cómo le hace el compilador para que se cumplan. El análisis de alcance es una extensión del analizador sintáctico.

4.1 BLOQUES.

Un programa en Mini-Pascal usa identificadores para referirse a constantes, tipos de datos, campos de registros, variables y procedimientos. Estas entidades se llaman objetos del programa.

Los tipos entero y Boolean, las constantes false y true, y los procedimientos read y write son objetos predefinidos que pueden usarse en cualquier programa en Mini-Pascal.

Cualquier otro objeto que se use en un programa debe ser definido mediante una definición que introduce el identificador del objeto y describe algunas de sus propiedades.

Una definición de constante introduce una constante, por ejemplo:

```
Const
  c = 10;
```

Una definición de tipo introduce un tipo de datos, por ejemplo:

```
Type
  T = array[1..c] of integer;
```

Un tipo registro introduce uno o más campos:

```
Type
  U = record
        f : T;
        g : Boolean
      end;
```

Una definición variable introduce una o más variables:

```
Var
  x,y : T;
```

Una definición de procedimiento define un procedimiento:

```
Procedure P(x,y : T);
begin
  ...
end;
```

Un programa combina proposiciones y definiciones relacionadas en unidades sintácticas llamadas bloques. Hay tres clases de bloques.

i) El bloque estándar.
ii) Programa.
iii) Procedimientos.

En el bloque estándar se definen los objetos estándar. El programa mismo contiene procedimientos. Cada procedimiento puede contener otros procedimientos.

4.2 REGLAS DE ALCANCE.

Regla 1. Todas las constantes, tipos, variables y procedimientos definidos en el mismo bloque deben tener nombres diferentes.

Regla 2. Una constante, tipo o variable definida en un bloque es conocida desde el final de su definición hasta el final del bloque. Un procedimiento definido en un bloque B es normalmente conocido desde el principio del bloque al final del bloque B.

Regla 3. Consideremos un bloque Q que define un objeto x. Si Q contiene un bloque R que define otro objeto llamado x, el primer objeto es desconocido en el alcance del segundo objeto.

Cuando un identificador se usa en un bloque, primero se trata de encontrar una definición del identificador en el mismo bloque. Si no se encuentra, se busca en el bloque anterior, y asi sucesivamente, hasta que se encuentre su definición o se llegue al bloque estándar.

4.3 METODO DE COMPILACION.

Usaremos el siguiente programa de ejemplo para explicar cómo realiza el compilador el análisis de alcance.

```
0  Program P;
1  type
2    T = array[1..100] of integer;
3  var
4    x : T;
5    procedure Q(x : integer);
6    const  c = 13;
7    begin ... x ... end;
8
9    procedure R;
10   var b,c : boolean;
11   begin
```

```
12      ... x ...
13    end;
14 begin
15      ...
16 end.
```

Durante el análisis de alcance, el compilador mantiene una pila de definiciones. Cuando el compilador reconoce una definición de un objeto nuevo, coloca el identificador del objeto en la pila. Inicialmente, el compilador coloca todos los identificadores estándar en la pila. Al final de un bloque, el compilador borra de la pila todos los identificadores definidos en ese bloque.

La figura 4.1 muestra la pila de varias etapas durante la compilación del programa de ejemplo.

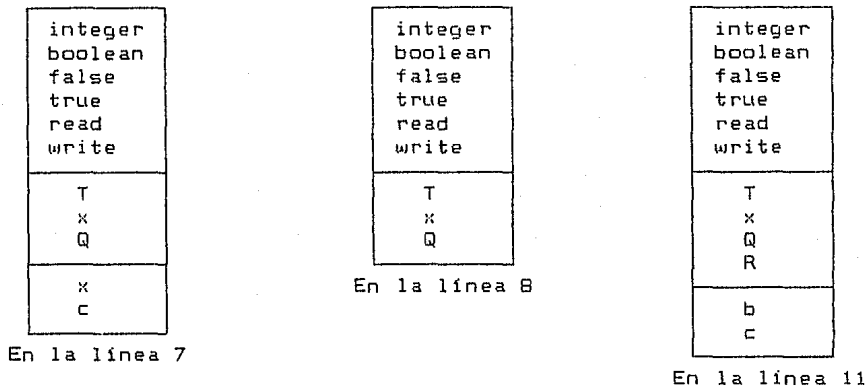


Fig. 4.1 La pila durante la compilación de un programa.

4.4 ESTRUCTURAS DE DATOS.

A cada bloque de un programa se le asigna un número de nivel. El bloque estándar está en el nivel 0. El número de nivel se incrementa en uno al principio de un nuevo bloque y se decrementa en uno al final del bloque. En el programa de ejemplo discutido antes, los bloques tienen los siguientes números de nivel:

Bloque	Número de nivel
Bloque estándar	0
Program P	1
Procedure Q	2
Procedure R	2

Todos los objetos que se definen en el mismo bloque estan representados mediante una lista ligada de registros. Cada registro describe un objeto mediante el índice del identificador y un puntero.

```
TYPE
  Ptro = ^Reg_obj;
  Reg_obj = RECORD
              id  : integer;
              ant : Ptro
            END;
```

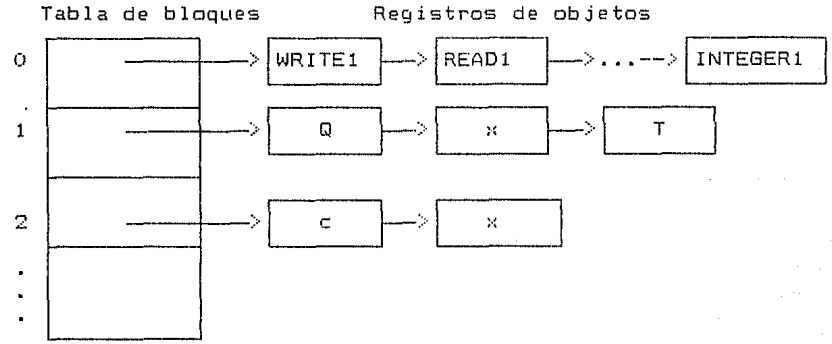


Fig. 4.2 Pila de identificadores representados por listas ligadas.

En el ejemplo anterior, el compilador está actualmente analizando un procedimiento Q en el nivel 2, el cual está contenido en el bloque del programa en el nivel 1 y en el bloque estándar en el nivel 0. Los objetos definidos en estos bloques anidados están descritos mediante tres listas ligadas (Fig. 4.2). El compilador usa una tabla de bloques para definir donde empiezan las listas:

```
CONST
  num_nivel = 10;
TYPE
  Reg_del_block = RECORD
                    ult_obj : Ptro
                  END;
  Tabla_de_blocks = ARRAY[0..num_nivel] OF Reg_del_block;

VAR
  Block : Tabla_de_blocks;
  Nivel_del_block : integer;
```

Cada entrada de la tabla de bloques contiene un puntero al último objeto definido en un bloque. Sin embargo, posteriormente agregaremos más atributos para realizar el análisis de tipos y generación de código.

4.5 ALGORITMOS.

Necesitamos un algoritmo para determinar si un identificador ya ha sido definido en un bloque dado. El bloque está identificado por su número de nivel.

```
PROCEDURE Busca_id(id, num_nivel : INTEGER;
                   VAR encont : boolean; VAR objeto : Ptro);
VAR
  mas : BOOLEAN;
BEGIN
  mas := TRUE;
  objeto := Block[num_nivel].ult_objeto;
  WHILE mas DO
  BEGIN
    IF objeto = NIL THEN
    BEGIN
      mas := FALSE;
      encont := FALSE;
    END
    ELSE
      IF objeto^.id = id THEN
      BEGIN
        mas := FALSE;
        encont := TRUE;
      END
      ELSE
        objeto := objeto^.previo
  END;
END;
```

Algoritmo 4.1 Determina si un identificador ya ha sido definido en un bloque determinado.

El analizador sintáctico usa el algoritmo 4.2 para definir un nuevo objeto. Si el bloque actual ya tiene definido otro objeto con el mismo nombre, el identificador es reportado como duplicado. En caso contrario el objeto es enlazado al otro objeto definido en el mismo bloque.

Cuando se usa un identificador para referirse a un objeto, el compilador primeramente trata de encontrar el objeto en el bloque actual. Si no lo encuentra, lo busca en bloque anterior, y así sucesivamente. Si el objeto no está descrito en la pila, el compilador lo reporta como indefinido y crea una definición del objeto en la pila.

```
PROCEDURE Define(id:INTEGER; clase:clas: VAR objeto : Ptro);
VAR
   encont : boolean; otro : Ptro;
BEGIN
   IF id <> no_hay_id THEN
   BEGIN
      Busca_id(id,nivel_block, encont,otro);
      IF encont THEN error(dup)
   END;
   NEW(objeto); objeto^.id := id;
   objeto^.previo := Block[nivel_block].ult_objeto;
   objeto^.clase := tipo;
   Block[nivel_block].ult_objeto := objeto;
END;
```

Algoritmo 4.2 Definición de un objeto nuevo en un bloque.

```
PROCEDURE Encuentra(id : INTEGER; VAR objeto:Ptro);
VAR
  mas, encont : BOOLEAN; num_nivel : INTEGER;
BEGIN
  mas := true; num_nivel := nivel_block;
  WHILE mas DO
  BEGIN
    busca_id(id, num_nivel, encont, objeto);
    IF encont or (num_nivel = 0) then mas := FALSE
    ELSE num_nivel := num_nivel - 1;
  END;
  IF NOT encont THEN
  BEGIN
    Error(id_indef);
    Define(id,Indefinido,objeto)
  END
END;
```

Algortimo 4.3 Verifica si existe un objeto en un bloque dado.

Al principio de cada bloque, el compilador se asegura que no se exceda al número máximo de niveles de bloques. Después incrementa el nivel del bloque en uno y crea una lista vacía para el nuevo bloque.

```
PROCEDURE BloqueNuevo;
BEGIN
  Verifica_limit(nivel_block,Num_niv);
  nivel_block := nivel_block + 1;
  Block[nivel_block].ult_objeto := nil
END;
```

Algortimo 4.4 Creación de una lista para un nuevo bloque.

Al final de un bloque, los objetos definidos en éste serán inaccesibles decrementando el nivel del bloque (algoritmo 4.5).

```
PROCEDURE Final_de_bloque;
BEGIN
  nivel_block := nivel_block - 1
END;
```

Algoritmo 4.5 Decrementa el nivel del bloque.

Esto es todo lo que necesitamos para agregar análisis de alcance al analizador sintáctico.

El análisis de un programa completo que ilustra el manejo de bloques se muestra en el algoritmo 4.6.

```
PROCEDURE Programa(Stop: simbolos);
BEGIN
   Acepta(PROGRAM1,[IDENT, PUNTOYCOMA, PUNTO] +
          simb_de_bloque + Stop);
   Acepta(IDENT,[PUNTOYCOMA,PUNTO]+simb_de_bloque +
          Stop);
   Acepta(PUNTOYCOMA, [PUNTO] + simb_de_bloque + Stop);
   BloqueNuevo;
   Bloque([PUNTO] + Stop);
   Final_de_bloque;
   Acepta(PUNTO,Stop);
END;
```

Algortimo 4.6 Análisis de un programa.

4.6 PRUEBA.

El análisis de alcance se prueba escribiendo un programa en Mini-Pascal que obligue ejecutar al analizador sintáctico cada proposición de los nuevos procedimientos. Un estudio de éstos muestra que la prueba del programa debe incluir proposiciones de las siguientes clases:

1) Una definición de constante.
2) Una definición de tipo.
3) Una definición de variable.
4) Una definición de un procedimiento recursivo.
5) Referencias a todos lo objetos estándar.
6) Referencias a todos los objetos definidos.

El programa Prueba4 (ver apéndice A.4), incluye todos éstos casos.

La salida del analizador sintáctico para este programa es la siguiente:

```
1   { Mini-Pascal. Prueba 4: Análisis de alcance }
2   program prueba4;
3   type
4      S = record
5             f, g : boolean
6          end;
7   var
8       v : S;
9
10       procedure P(x: integer);
11       const
12           n = 10;
13       type
14           T = array[1..n] of integer;
15       var
16           y, z : T;
17
18           procedure Q;
19           begin
20              read(x);
21              v.g := false
22           end;
23       begin
24         y := z;
25         Q;
26         P(5);
27         write(x)
28       end;
29   begin
30      v.f := true;
31      P(5)
32   end.

        Análisis sintáctico terminado sin errores
        Número de identificadores usados: 12
```

También debemos mostrar que el análisis de alcance puede reportar los siguientes errores:

1) Una definición de un identificador duplicado.
2) Una definición sin identificador.
3) Definiciones recursivas sin sentido.
4) Referencias a un identificador indefinido.

Estos casos de prueba son cubiertos por el programa Prueba5 (ver apéndice A.4). A continuación se muestra la salida proporcionada por el analizador sintáctico para este programa.

```
1    { Mini-Pascal. Prueba 5: Errores de alcance }
2    program prueba5;
3    const
4         {a} = 1;

* Error en linea 4:  Error de sintaxis

5         b   = b;

* Error en linea 5:  Identificador indefinido

6    type
7       T = array[1..10] of T;

* Error en linea 7:  Identificador indefinido

8       U = record
9              f, g : U

* Error en linea 9:  Identificador indefinido

10           end;
11   var
12      x, y, x : integer;

* Error en linea 12:  Identificador duplicado

13   begin
14      x := a;

* Error en linea 14:  Identificador indefinido

15      y := a
16   end.

     6 Error(es) encontrado(s)
     Número de identificadores usados: 9
```

CAPITULO 5. ANALISIS DE TIPOS.

INTRODUCCION.

Este Capítulo explica la forma en la que el compilador usa las definiciones de objetos para realizar el análisis de tipos. El análisis de tipos es una extensión del analizador sintáctico, el cual ya realiza análisis de alcance.

5.1 CLASES DE OBJETOS.

Durante el análisis de alcance, un objeto se describió sólo por su identificador y su enlace a los objetos definidos en el mismo bloque:

```
TYPE
  Ptro : ^Reg_obj;
  Reg_obj = RECORD
              id : INTEGER;
              ant: Ptro
            END;
```

Sin embargo, para efectuar el análisis de tipos, el compilador debe ser capaz de distinguir las diferentes clases de objetos: constantes, tipos, variables, procedimientos, etc. Usaremos el siguiente tipo de datos para clasificar objetos:

```
TYPE
  Clas = (Constantex, TipoEstandar, TipoArray, TipoRecord,
          Campo,Variable, Param_val,Param_var, Procedimiento,
          Proc_estandar, Indefinido);
```

El analizador sintáctico debe almacenar toda la información contenida en la definición de un objeto, incluyendo su clase. Por ejemplo, una constante está descrita por su tipo y valor. Por otra parte, un procedimiento está caracterizado por su lista de parámetros. Puesto que la información varía de una clase de objeto a otro, es más conveniente describir los objetos por registros variantes del siguiente tipo:

```
Reg_obj = RECORD
            id          : INTEGER;
            ant         : Ptro;
            CASE clase : clas OF
               constantex : (valor_de_la_cte : INTEGER;
                             tipo_de_la_cte : Ptro);
               TipoArray  : (Limite_inf, Limite_sup : INTEGER;
                             TipoIndice, TipoElemento : Ptro);
               TipoRecord : (ult_campo : Ptro);
               Campo      : (Tipo_campo : Ptro);
               Variable,Param_val,Param_var :(tipo_de_var:Ptro);
               Procedimiento : (ult_param : Ptro);
            END;
```

Después de extender los registros de los objetos con una parte variante, debemos modificar los procedimientos de análisis de alcance para habilitar al parser para clasificar objetos y accesar sus registros a través de punteros.

Cuando el parser reconoce la definición de un objeto, llama al procedimiento siguiente:

```
Procedure define(id:integer; clase:clas; var objeto:ptro);
```

Este procedimiento crea un registro con el identificador y clase de objeto y devuelve un puntero a este registro. El parser usa el puntero para llamar la parte variante del registro con información adicional referente al objeto.

En el momento de que el parser encuentre una referencia a un objeto, llama al siguiente procedimiento:

```
Procedure encuentra(id:integer; var objeto : ptro);
```

Este procedimiento trata de encontrar el registro de un objeto de un identificador dado. Si no existe, el procedimiento crea un registro de una clase ficticia, llamada indefinida. En ambos casos, el procedimiento devuelve un puntero al registro seleccionado, el cual usa el parser para accesar la información disponible referente al objeto.

A continuación se explica cómo se manejan las diferentes clases de objetos.

5.2 TIPOS ESTANDAR.

Cada tipo estándar -entero y booleano- está descrito por un registro de objeto. La parte fija del registro define el identificador del tipo y el enlace al objeto anterior definido en el bloque estándar. El enlace se usa sólo para el análisis de alcance y no juega ningún papel en el análisis de tipos. La parte variante consiste del campo clase con el valor TipoEstandar.

El parser define los tipos estándar de la siguiente forma:

```
Var
  TipoEntero, TipoBoolean : Ptro;
            ...
Begin
  Define(ord(INTEGER1), TipoEstandar, TipoEntero);
  Define(ord(BOOLEAN1), TipoEstandar, TipoBoolean);
            ...
```

Durante el análisis de tipos, cada tipo está representado por un puntero al registro correspondiente. Estos punteros están almacenados en dos variables denominadas TipoEntero y TipoBoolean.

La figura 5.1 muestra el registro que describe el tipo Boolean y la variable que apunta a éste. El enlace al objeto anterior no se utiliza en el análisis de tipos.

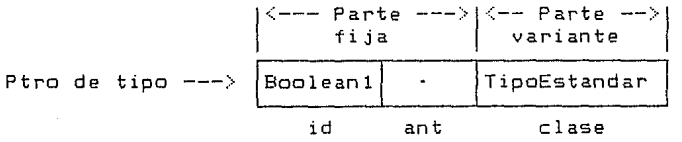


Fig. 5.1 Representación de datos del tipo Boolean.

5.3 CONSTANTES.

Una constante está descrita por un registro de objeto que incluye el identificador de la constante:

```
Reg_obj = RECORD
            id          : INTEGER;
            ant         : Ptro;
            CASE clase : clas OF
                 constantex : (valor_de_la_cte : INTEGER;
                 tipo_de_la_cte : Ptro);
                     ...
          END;
```

La parte variante del registro consiste de un campo de clase con el valor constantex y dos campos que definen el tipo y valor de la constante.

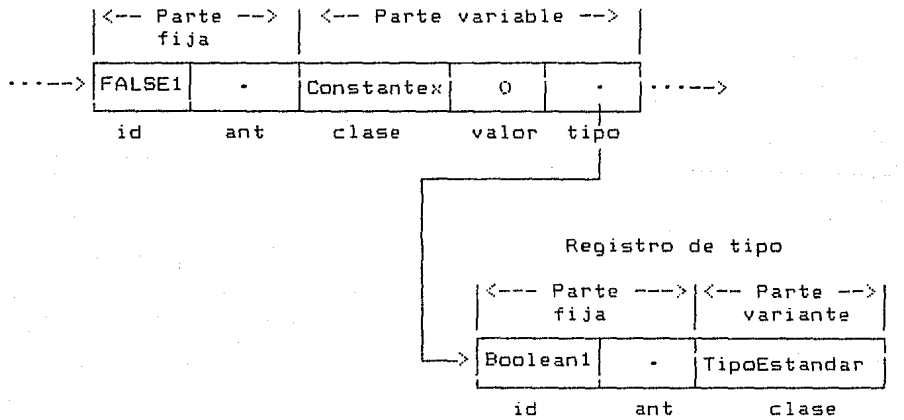


Fig. 5.2 Representación de datos de la constante false.

La fig. 5.2 muestra el registro que describe la constante estándar false. El campo de tipo apunta al registro que describe

el tipo Boolean. El parser crea esta representación de datos de la siguiente forma:

```
VAR
  Constx : Ptro;
  ...
  Define(ORD(FALSE1),constantex,Constx);
  Const^.valor_de_la_cte := ORD(FALSE);
  Const^.tipo_de_la_cte := TipoBoolean;
```

Al encontrar una constante, el parser debe determinar el tipo y valor de este objeto. Esta información es proporcionada por el algoritmo 5.1. Si el análisis sintáctico se basa sólo sobre sintaxis, el compilador no puede distinguir entre los identificadores de constantes y otras clases de objetos. Esta confusión se resuelve mediante el uso de clases de objetos.

```
PROCEDURE Constante(VAR valor: INTEGER; VAR Tipox: Ptro;
                    Stop: simbolos);
VAR
  Objeto: Ptro;
BEGIN
  IF simbolo = CTE_ENT THEN
  BEGIN
    valor := argumento;
    Tipox := TipoEntero;
    Acepta(CTE_ENT,Stop)
  END
  ELSE
    IF simbolo = IDENT THEN
    BEGIN
      Encuentra(argumento, Objeto);
      IF objeto^.clase = Constantex THEN
      BEGIN
        valor := Objeto^.valor_de_la_cte;
        Tipox := objeto^.tipo_de_la_cte
      END
      ELSE
        BEGIN
          Error_de_clase(Objeto);
          valor := 0;
          Tipox := TipoUniversal
        END;
        Acepta(IDENT, Stop)
    END
    ELSE
      BEGIN
        Error_de_sintaxis(Stop);
        valor := 0;
        Tipox := TipoUniversal
      END
END;
```

Algoritmo 5.1 Análisis de una constante.

El algoritmo 5.2 ilustra dos formas de recuperación de errores que se utilizan durante el análisis de tipos:

(1) Cuando el parser encuentra un identificador que no se refiere a una constante, éste lo reporta como un identificador de clase incorrecta mediante la ejecución del procedimiento 5.2. Si el identificador se refiere a un objeto indefinido, ya ha causado un mensaje de error durante el análisis de alcance. En este caso, el procedimiento no producirá ningún otro mensaje de error.

```
PROCEDURE Error_de_clase(objeto: Ptro);
BEGIN
  IF objeto^.tipo <> indefinido THEN Error(tipo_id_inc)
END;
```

Algoritmo 5.2 Recuperación de errores durante el análisis de tipos.

(2) Si el parser no reconoce una constante, devuelve el valor (cero) de un tipo ficticio conocido como tipo universal, con lo cual se asegura que un operando de este tipo nunca cause otro mensaje de error durante la verificación de tipos.

El tipo universal está definido como un tipo estándar con un valor de cero:

```
CONST
  No_hay_id = 0;
VAR
  TipoUniversal : ptro;
  ...
BEGIN
  ...
  Define(No_hay_id, TipoEstandar,TipoUniversal);
  ...
```

5.4 VARIABLES.

La siguiente definición de procedimiento ilustra las diferentes clases de variables que se usan en Mini-Pascal:

```
PROCEDURE P(u : INTEGER; VAR v:INTEGER);
VAR
  x,y : BOOLEAN;
BEGIN
  ...
END;
```

ésta define un parámetro por valor u, un parámetro variable v, y dos variables locales x, y.

Durante la compilación, cada variable es descrita por su identificador, clase y tipo:

```
Reg_obj = RECORD
            id          : INTEGER;
            ant         : Ptro;
            CASE clase : clas OF
                 ...
            Variable,Param_val,Param_var :(tipo_de_var:Ptro);
                 ...
          END;
```

El valor del campo clase puede ser variable, param_val o param_var. La figura 5.3 muestra dos registros que describen las variables locales x, y. Los campos de tipos de estos registros apuntan al registro que describe el tipo Boolean.

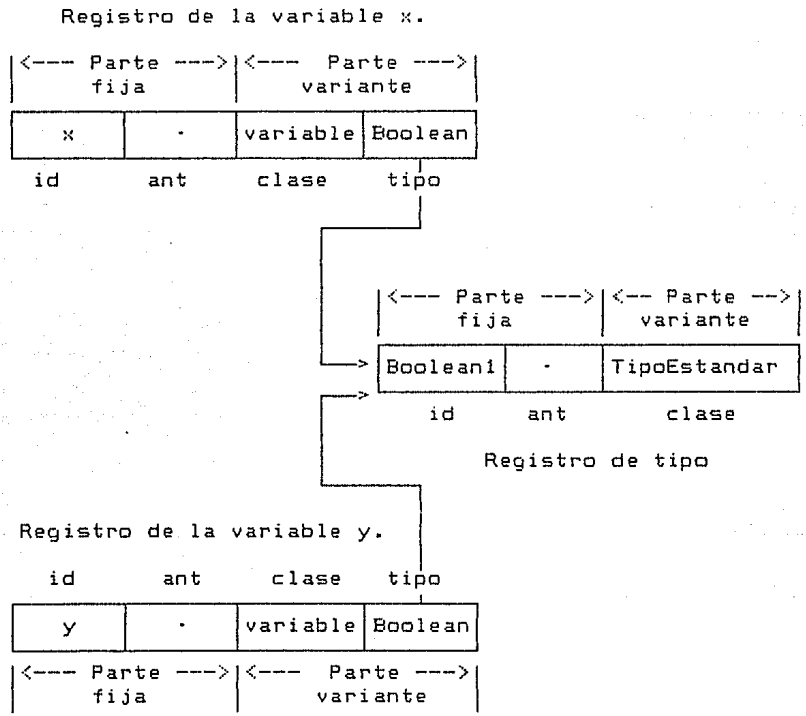


Fig. 5.3 Representación de datos de dos variables.

Los registros de objetos para parámetros locales y parámetros variables difieren solamente en el valor del campo clase. Por lo tanto, podemos usar un sólo procedimiento de análisis para construir cualquier clase de registro para una variable.

El algoritmo 5.3 define el procedimiento recursivo de análisis. Si encuentra un identificador de variable seguido por una coma, éste se llama a sí mismo. Por lo tanto, cada identificador de variable es manejado por una sola llamada separada al procedimiento. La última de éstas ocurre cuando el procedimiento alcanza el identificador de tipo y obtiene un puntero al registro del tipo correspondiente. Este puntero es asignado a un parámetro variable llamado tipox, el cual es compartido por todas las llamadas al procedimiento. Al final, cada llamada al procedimiento completa su propio registro de variable con una copia del puntero de tipo y termina.

```
PROCEDURE lista_variables(clase: clas; VAR ult_var,
                          tipox: Ptro; Stop: simbolos);
VAR
  id: INTEGER;
  Varx: Ptro;
BEGIN
  Acepta_id(id, [COMA, DOSPUNTOS] + Stop);
  Define(id,clase,Varx);
  IF simbolo = COMA THEN
  BEGIN
    Acepta(COMA, [IDENT] + Stop);
    lista_variables(clase, ult_var, Tipox, Stop)
  END
  ELSE
    BEGIN
      ult_var := Varx;
      IF simbolo = DOSPUNTOS THEN
      BEGIN
        Acepta(DOSPUNTOS, [IDENT] + Stop);
        Id_de_tipo(Tipox, Stop);
      END
      ELSE
        BEGIN
          Error_de_sintaxis(Stop);
          Tipox := TipoUniversal
        END
    END;
  Varx^.tipo_de_var := Tipox
END;
```

Algoritmo 5.3 Análisis de tipos para variables.

El procedimiento anterior también devuelve un puntero al último registro de una variable dentro de una lista de variables.

Cuando el parser espera encontrar un identificador de tipo en una sentencia, éste ejecuta el algoritmo 5.4 para obtener un puntero al registro del tipo correspondiente. Si no hay identificador o el identificador no se refiere a un tipo, este procedimiento devuelve un puntero al tipo universal, de tal forma que el resto del parser no se vea afectado por el error.

```
PROCEDURE Id_de_tipo(VAR Tipox: Ptro; Stop: simbolos);
VAR
  Objeto: Ptro;
BEGIN
  IF simbolo = IDENT THEN
  BEGIN
    Encuentra(argumento, Objeto);
    IF objeto^.clase IN tipos then
       Tipox := objeto
    ELSE
    BEGIN
      Error_de_clase(objeto);
      Tipox := TipoUniversal
    END
  END
  ELSE
    Tipox := TipoUniversal;
  Acepta(IDENT, Stop)
END;
```

Algoritmo 5.4 Análisis de un identificador de tipo.

El compilador usa un conjunto de valores para determinar si un identificador se refiere a un tipo:

```
TYPE
  clases = SET OF clas;
VAR
  tipos : clases;
          ...
BEGIN
          ...
  tipos := [TipoEstandar, TipoArray, TipoRegistro]
```

Ahora es fácil analizar definiciones de variables por medio del algoritmo 5.5.

```
ROCEDURE Def_de_variable(Stop:simbolos);
VAR
  ult_var, tipox : Ptro;
BEGIN
  lista_variables(Variable,ult_var,tipox,[PUNTOYCOMA]+Stop);
  Acepta(PUNTOYCOMA,Stop);
END;
```

Algoritmo 5.5 Análisis de tipos para la def. de variables.

5.5 ARREGLOS.

Un tipo arreglo está descrito por un registro de objeto. La parte variante consiste de un campo tipo con el valor TipoArray y cuatro campos más. Tres de estos campos definen el tipo de los elementos del arreglo. Este registro de objeto se muestra a continuación:

```
Reg_obj = RECORD
          id          : INTEGER;
          ant         : Ptro;
          CASE clase : clas OF
                ...
          TipoArray : (Limite_inf, Limite_sup : INTEGER;
                       TipoIndice, TipoElemento : Ptro);
                ...
        END;
```

La figura 5.4 muestra un registro que describe el tipo arreglo siguiente:

```
TYPE
   T = Array[1..10] of Boolean;
```

Los límites inferior y superior de los índices son 1 y 10, respectivamente. Los campos de tipo apuntan a registos que describen los tipos entero y Booleano.

Registros de tipos.

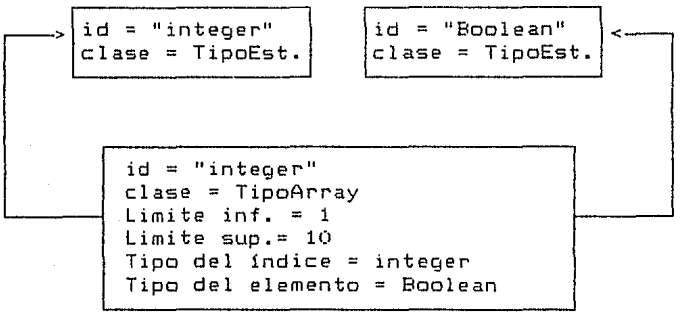


Figura 5.4 Representación de datos de un tipo arreglo.

Una definición completa de tipo tiene la siguiente sintaxis:

```
<Def. de tipo> ::= <Id. de tipo> = <Tipo>;
<Tipo> ::= <Tipo arreglo> | <Tipo registro>
```

El procedimiento que analiza una definición de tipo introduce el identificador de un nuevo tipo arreglo y lo pasa como parámetro al algoritmo 5.6. Este procedimiento analiza la definición de un tipo arreglo y construye el registro correspondiente. Para mayor claridad se han reemplazado los distintos símbolos de parada por puntos.

```
PROCEDURE TipoArreglo(id: INTEGER; Stop: simbolos);
VAR
  Tipo,tipo_del_lim_inf,tipo_del_lim_sup,TipoElemento : Ptro;
  Limite_sup, Limite_inf : INTEGER;
BEGIN
    Acepta(ARRAY1,...); Acepta(CORCHETE_IZQ,...);
    Constante(Limite_inf,tipo_del_lim_inf,...);
    Acepta(PUNTOPUNTO ,...);
    Constante(Limite_sup,tipo_del_lim_sup,...);
    Verifica_tipos(tipo_del_lim_inf,tipo_del_lim_sup);
    IF Limite_inf > Limite_sup THEN
    BEGIN
      Error(rango_inc);
      Limite_inf := Limite_sup
    END;
    Acepta(CORCHETE_DER,...);
    Acepta(OF1,...);
    Id_de_tipo(TipoElemento, Stop);
    Define(id ,TipoArray, Tipo);
    Tipo^.Limite_inf := Limite_inf;
    Tipo^.Limite_sup := Limite_sup;
    Tipo^.TipoIndice  := tipo_del_lim_inf;
    Tipo^.TipoElemento := TipoElemento
  END;
```

Algoritmo 5.6 Análisis de la definición de un tipo arreglo.

El parser usa el algortimo 5.7 para verificar que los índices del rango sean del mismo tipo. Si los tipos son diferentes, el parser reportará un error de tipo, a menos que uno de los tipos sea el tipo universal, el cual es compatible con cualquier otro tipo. Después de un error de tipo, el primer tipo es reemplazado por el tipo universal para suprimir mensajes de error adicionales.

```
PROCEDURE Verifica_tipos(VAR tipo1 : Ptro; tipo2: Ptro);
BEGIN
  IF tipo1 <> tipo2 THEN
  BEGIN
    IF (tipo1<>TipoUniversal) and (tipo2<>TipoUniversal) THEN
       Error(tipo_id_inc);
    tipo1 := TipoUniversal
   END
END;
```

Algortimo 5.7 Análisis de tipos de los índices de un rango.

Ahora consideraremos un acceso a una variable indexada x[i+1], definida de la siguiente forma:

```
TYPE
  T = ARRAY[1..10] OF BOOLEAN;
VAR
  x : T;
  i : INTEGER;
```

El procedimiento denominado accesa_var introduce el identificador x, y obtiene un puntero al registro que describe su tipo T. Este procedimiento después llama a otro procedimiento para analizar el selector de índice [x + 1].

```
PROCEDURE Accesa_var(VAR Tipox: Ptro; Stop: simbolos);
VAR
  Stop2 : simbolos;
  objeto: Ptro;
BEGIN
  IF simbolo = IDENT THEN
  BEGIN
    Stop2 := simb_selectores + simb_multiplicadores + Stop;
    Encuentra(argumento, objeto);
    Acepta(IDENT, Stop2);
    IF objeto^.clase IN Variables THEN
       Tipox := objeto^.tipo_de_var
    ELSE
       BEGIN
         Error_de_clase(objeto);
         Tipox := TipoUniversal
       END;
    WHILE simbolo in simb_selectores DO
      IF simbolo = CORCHETE_IZQ THEN
         Selector_de_indice(Tipox, Stop2)
      ELSE {simbolo = PUNTO}
         Selector_de_campo(Tipox, Stop2)
  END
  ELSE
    BEGIN
      Error_de_sintaxis(Stop);
      Tipox := TipoUniversal
    END
END;
```

Algoritmo 5.8 Accesa un identificador y su tipo.

5.6 REGISTROS.

Ahora discutiremos el análisis de alcance y de tipos del tipo de dato registro. El siguiente ejemplo:

```
TYPE
  R = RECORD
        f : BOOLEAN;
        g : T;
      END;
VAR
  x : R;
```

define una variable x del tipo registro R. La variable completa x consiste de dos campos variables:

x.f x.g

Las reglas de alcance de campos, como f y g, son diferentes de las reglas de alcance de otros objetos. Cuando un identificador x se usa varias veces en un bloque, éste se refiere usualmente al mismo objeto (en este caso una variable). Pero un identificador de campo f se puede referir a diferentes objetos en el mismo bloque, y ¡algunos de estos objetos pueden aún no ser campos!.

En un bloque con las definiciones:

```
TYPE
  R = RECORD
        f : BOOLEAN;
        g : T
      END;

  S = RECORD
        a, b : INTEGER;
        f : R;
      END;

VAR
  f : INTEGER;
  x : R;
  y : S;
```

el identificador f puede denotar cuatro objetos diferentes:

(1) La variable f de tipo entero.
(2) El campo variable x.f de tipo Boolean;
(3) El campo variable y.f de tipo R;
(4) El campo variable y.f.f de tipo Boolean

En el último caso, el primer identificador f selecciona un campo de tipo R dentro de la variable y. El segundo identificador f selecciona un subcampo de tipo Boolean dentro del campo anterior.

Para determinar si x.f se refiere a un campo variable, el parser debe observar la definición de x y verificar que ésta sea una variable de tipo registro que incluye un campo f. Puesto que esto requiere análisis de tipos, no puede ser realizado durante el análisis de alcance regular.

Después de esta introducción, definiremos las reglas de alcance para campos.

i) Todos los campos definidos en el mismo tipo de registro deben tener identificadores diferentes.

ii) Un campo f de una variable x es denotada como x.f y es conocido sólo en el alcance de x.

x.f x.g

Las reglas de alcance de campos, como f y g, son diferentes de las reglas de alcance de otros objetos. Cuando un identificador x se usa varias veces en un bloque, éste se refiere usualmente al mismo objeto (en este caso una variable). Pero un identificador de campo f se puede referir a diferentes objetos en el mismo bloque, y ¡algunos de estos objetos pueden aún no ser campos!.

En un bloque con las definiciones:

```
TYPE
  R = RECORD
        f : BOOLEAN;
        g : T
      END;

  S = RECORD
        a, b : INTEGER;
        f : R;
      END;

VAR
  f : INTEGER;
  x : R;
  y : S;
```

el identificador f puede denotar cuatro objetos diferentes:

(1) La variable f de tipo entero.
(2) El campo variable x.f de tipo Boolean;
(3) El campo variable y.f de tipo R;
(4) El campo variable y.f.f de tipo Boolean

En el último caso, el primer identificador f selecciona un campo de tipo R dentro de la variable y. El segundo identificador f selecciona un subcampo de tipo Boolean dentro del campo anterior.

Para determinar si x.f se refiere a un campo variable, el parser debe observar la definición de x y verificar que ésta sea una variable de tipo registro que incluye un campo f. Puesto que esto requiere análisis de tipos, no puede ser realizado durante el análisis de alcance regular.

Después de esta introducción, definiremos las reglas de alcance para campos.

i) Todos los campos definidos en el mismo tipo de registro deben tener identificadores diferentes.

ii) Un campo f de una variable x es denotada como x.f y es conocido sólo en el alcance de x.

```
PROCEDURE lista_de_campos(VAR ult_campo:Ptro;Stop:simbolos);
VAR
  Stop2: simbolos;
  Tipox: Ptro;
BEGIN
  Stop2 := [PUNTOYCOMA] + Stop;
  Seccion_de_regs(ult_campo, Tipox, Stop2);
  WHILE simbolo = PUNTOYCOMA DO
  BEGIN
    Acepta(PUNTOYCOMA, [IDENT] + Stop2);
    Seccion_de_regs(ult_campo,Tipox, Stop2);
  END;
END;
```

Algoritmo 5.10 Análisis de tipos para campos.

Ahora consideraremos el acceso a una variable x.f, donde x es una variable de tipo R:

```
TYPE
  R = RECORD
        f : Boolean;
        g : T
      END;
VAR
  x : R;
```

El parser llama al procedimiento accesa_var para introducir el identificador de variable x y obtener un puntero a su tipo R (algoritmo 5.8). Después, ejecuta el algoritmo 5.11 para verificar el selector de campo ".f". Al Principio de este procedimiento, el parámetro denominado tipox apunta al tipo de la variable x. Si éste es un registro de tipo, el procedimiento examina los campos buscando el identificador del campo f. Al final del procedimiento, tipox apunta al tipo del campo seleccionado – a menos que el parser encuentre un error, en cuyo caso el tipo de campo de colocado a "universal" para suprimir mensajes de errores adicionales.

```
PROCEDURE Selector_de_campo(VAR Tipox: Ptro; Stop: simbolos);
VAR
  encont: Boolean;
  Campox: Ptro;
BEGIN
  Acepta(PUNTO, [IDENT] + Stop);
  IF simbolo = IDENT THEN
  BEGIN
    IF Tipox^.clase = TipoRecord THEN
    BEGIN
      encont := false;
      campox := Tipox^.ult_campo;
      WHILE not encont and (campox <> nil) DO
        IF campox^.id <> argumento THEN campox := campox^.ant
        ELSE encont := true;
```

```
      IF encont THEN Tipox := campox^.Tipo_campo
      ELSE
      BEGIN
        Error(id_indef);
        Tipox := TipoUniversal
      END
    END
    ELSE
    BEGIN
      Error_de_clase(Tipox);
      Tipox := TipoUniversal
    END;
    Acepta(IDENT, Stop)
  END
  ELSE
  BEGIN
    Error_de_sintaxis(Stop);
    Tipox := TipoUniversal
  END
END;
```

Algoritmo 5.11 Análisis de tipos para un selector de campo.

5.7 EXPRESIONES.

El tipo de una expresión está determinado por cuatro procedimientos: Expresión, ExpresiónSimple, Término y Factor. El diagrama de dependencia de la figura 5.5 muestra el orden en el cual pueden llamarse estos procedimientos.

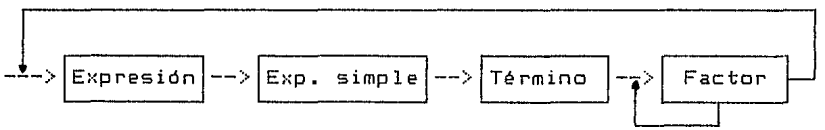


Fig. 5.5 Diagrama de dependencia para expresiones.

Observemos primero los factores (algoritmo 5.12). Este procedimiento ilustra la naturaleza recursiva de la compilación. Si encuentra una expresión encerrada entre paréntesis, llama al procedimiento expresión en forma recursiva. Si no encuentra un operador delante de un operando, se llama a sí mismo.

Un término es analizado por medio del algoritmo 5.13 usando el siguiente conjunto de símbolos:

```
simbolos_multiplicadores = [AND1, POR, DIV1, MOD1]
```

El procedimiento verifica que los operandos aritméticos sean aplicados sólo a operandos enteros y que el operador and tenga operandos Booleanos.

```
PROCEDURE Factor(VAR Tipox: Ptro; Stop: simbolos);
VAR
  Objeto: Ptro;
  valor: INTEGER;

BEGIN
  CASE simbolo OF

       CTE_ENT : Constante(valor,Tipox,...);

       IDENT   : BEGIN
                   Encuentra(Argumento,Objeto);
                   IF Objeto^.clase = Constantex
                   THEN
                      Constante(valor,Tipox, Stop)
                   ELSE
                      IF Objeto^.clase IN Variables
                      THEN
                         Accesa_var(Tipox,Stop)
                      ELSE
                      BEGIN
                        Error_de_clase(Objeto);
                        Tipox := TipoUniversal;
                        Acepta(IDENT,Stop)
                      END
                 END;

      PARENT_IZQ : BEGIN
                     Acepta(PARENT_IZQ,inic_expr + ...);
                     Expresion(Tipox, [PARENT_DER] + Stop);
                     Acepta(PARENT_DER,...);
                   END;


      NOT1 :       BEGIN
                     Acepta(NOT1, inic_de_factores + Stop);
                     Factor(Tipox,Stop);
                     Verifica_tipos(Tipox, TipoBoolean)
                   END;

      ELSE
                   BEGIN
                     Error_de_sintaxis(Stop);
                     Tipox:=TipoUniversal
                   END
    END  {CASE}

  END;   {Factor}
```

Algoritmo 5.12 Análisis de tipos para factores.

```
PROCEDURE Termino(VAR tipox : Ptro; Stop : simbolos);
VAR
  operador : t_simb;
  Tipo2 : Ptro;
BEGIN
  Factor(Tipox,Stop);
  WHILE simbolo IN simb_multiplicadores DO
  BEGIN
    operador := simbolo;
    Acepta(simbolo,inic_de_factores + Stop);
    Factor(Tipo2,Stop);
    IF Tipox = TipoEntero THEN
    BEGIN
      Verifica_tipos(Tipox,Tipo2);
      IF operador = AND1 THEN Error_de_tipo(Tipox)
    END
    ELSE
      IF Tipox = TipoBoolean THEN
      BEGIN
        Verifica_tipos(Tipox,Tipo2);
        IF operador <> AND1 THEN Error_de_tipo(Tipox)
      END
      ELSE
        Error_de_tipo(Tipox)
  END
END;
```

Algoritmo 5.13 Análisis de tipos para términos.

Los demás procedimientos para el análisis de una expresión son similares y se muestran en el listado del analizador sintáctico (ver apéndice A.3).

5.8 PROPOSICIONES.

El algoritmo 5.14 se usa para analizar proposiciones. Este procedimiento usa los registros de objetos para distinguir las diferentes clases de identificadores. Si una proposición empieza con un identificador de variable, ésta debe ser una proposición de asignación. Si empieza con un identificador de procedimiento, debe ser una proposición procedure.

Hay dos clases de procedimientos: procedimientos estándar, y los procedimiento que están definidos en el programa fuente.

Las proposiciones restantes son vacías o empiezan con una palabra reservada única.

En una proposición de asignación, la variable y la expresión deben ser del mismo tipo. Esta verificación de tipos se lleva a cabo mediante el algoritmo 5.15.

```
PROCEDURE Prop (Stop: simbolos);
VAR
  Objeto : Ptro;

BEGIN
  CASE simbolo OF
       IDENT : BEGIN
                 Encuentra(argumento,Objeto);
                 IF objeto^.clase in Variables THEN
                    Prop_asignacion(stop)
                 ELSE
                 IF objeto^.clase in procedimientos THEN
                    Prop_procedure(Stop)
                 ELSE
                 BEGIN
                   Error_de_clase(objeto);
                   Acepta(IDENT,Stop)
                 END
               END;
       IF1    : Prop_if(Stop);
       WHILE1 : Prop_WHILE(Stop);
       BEGIN1 : Prop_compuesta(Stop);
       ELSE   Verifica_sintaxis(Stop)
  END;
END;
```

Algoritmo 5.14 Análisis de tipos para una proposición.

```
PROCEDURE Prop_asignacion(Stop: simbolos);
VAR
  tipo_de_la_var, Tipo_de_la_expr : Ptro;
BEGIN
  Accesa_var(tipo_de_la_var,[ASIGNACION]+inic_expr+Stop);
  Acepta(ASIGNACION, inic_expr + Stop);
  Expresion(Tipo_de_la_expr,Stop);
  Verifica_tipos(tipo_de_la_var,Tipo_de_la_expr)
END;
```

Algoritmo 5.15 Análisis de tipos de una prop. de asignación.

El parser verifica que la expresión que está dentro de una proposición while sea de tipo Boolean.

```
PROCEDURE Prop_While(Stop: simbolos);
VAR
  Tipo_de_la_expr: Ptro;
BEGIN
  Acepta(WHILE1,inic_expr+[DO1]+inic_de_props+Stop);
  Expresion(Tipo_de_la_expr,[DO1]+inic_de_props+Stop);
  Verifica_tipos(Tipo_de_la_expr, TipoBoolean);
  Acepta(DO1, inic_de_props + Stop); Prop(Stop);
END;
```

Algoritmo 5.16 Análisis de tipos para la proposición while.

5.9 PROCEDIMIENTOS.

Una definición de procedimiento está descrita por un registro de objeto del siguiente tipo:

```
Reg_obj = RECORD
            id          : INTEGER;
            ant         : Ptro;
            CASE clase : clas OF
                 ...
                 Procedimiento : (ult_param : Ptro);
                 ...
          END;
```

La parte variante apunta al registro que describe el último parámetro. Los parámetros están descritos como variables.

Una definición de procedimiento tiene la siguiente sintaxis:

```
<Def. de procedimiento> ::= PROCEDURE <Id. de proc.>
                                <Bloque del proc.>;
<Bloque del proc.> ::= [<(Lista de parámetros formales)>]
                       <Bloque>
```

El algoritmo 5.17 muestra cómo el parser crea un registro de un procedimiento.

```
PROCEDURE Def_de_procedimiento(Stop: simbolos);
VAR
  id    : INTEGER; Proc : Ptro;
BEGIN
  Acepta(PROCEDURE1, [IDENT, PARENT_IZQ, PUNTOYCOMA] +
         inic_de_bloque + Stop);
  Acepta_id(id,[PARENT_IZQ,PUNTOYCOMA]+inic_de_bloque+ Stop);
  Define(id,procedimiento,Proc);
  BloqueNuevo;
  IF simbolo = PARENT_IZQ THEN
  BEGIN
    Acepta(PARENT_IZQ, inic_param + [PARENT_DER, PUNTOYCOMA]
           + inic_de_bloque+Stop);
    lista_de_param_formal(Proc^.ult_param, [PARENT_DER,
                          PUNTOYCOMA] + inic_de_bloque+Stop);
    Acepta(PARENT_DER, [PUNTOYCOMA] + inic_de_bloque + Stop)
  END
  ELSE
    Proc^.ult_param := NIL;
  Acepta(PUNTOYCOMA, [PUNTOYCOMA] + inic_de_bloque + Stop);
  Bloque([PUNTOYCOMA] + Stop); Acepta(PUNTOYCOMA, Stop);
  Final_de_bloque
END;
```

Algoritmo 5.17 Creación de un reg. para un procedimiento.

Del procedimiento anterior podemos observar lo siguiente:

i) El registro del procedimiento es creado al principio de la definición del procedimiento para permitir llamadas recursivas dentro del bloque de éste (regla de alcance 2).

ii) El bloque del procedimiento es tratado como un bloque separado. Los parámetros formales son locales al bloque del procedimiento, mientras que el procedimiento mismo es local al bloque circundante. Esto se debe a que el identificador de procedimiento se define antes que el nuevo bloque sea analizado.

iii) Si el procedimiento no tiene parámetros, el puntero a su último parámetro es puesto a nil. Si el procedimiento tiene una lista de parámetros, el primer parámetro es también el primer objeto definido en el bloque del procedimiento. Consecuentemente, el registro de este parámetro no apunta a ningún objeto previo, sino que tiene el valor asignado al campo denominado ant. El resultado es que los parámetros formales de un procedimiento siempre están enlazados por una cadena de punteros que empiezan en el registro del procedimiento y terminan con un puntero a nil. Esta cadena de punteros se utiliza para realizar el análisis de tipos de proposiciones procedure.

El procedimiento que analiza una lista de parámetros formales:

<Lista de parámetros formales > ::= <Def. de parámetro>
{<Def. de parámetro>}

PROCEDURE lista_param_formal(VAR ult_param:ptro; Stop:simbolos);

devuelve un puntero al último registro de parámetro. Los registros de parámetros son creados, uno a la vez, por el procedimiento denominado Def_de_param.

El último paso en la compilación de una definición de procedimiento es analizar un bloque, procedimiento descrito antes.

El parser usa la descripción de un procedimiento para analizar llamadas al procedimiento. El siguiente ejemplo muestra una llamada al procedimiento P:

```
VAR
  a : integer;
  b : Boolean;
BEGIN
    ...
    P(a + 1, b)
    ...
END.
```

Primeramente, el compilador encuentra el registro que describe el procedimiento. Después, sigue la cadena de registros de parámetros y verifica lo siguiente:

i) El número de parámetros actuales en la proposición procedure debe ser igual al número de parametros formales en la definición del procedimiento.

ii) El parámetro actual "a+1" debe ser una expresión del mismo tipo que el parámetro por valor x (tipo entero).

iii) El parámetro actual "b" debe ser una variable del mismo tipo que el parámetro y (tipo Boolean).

El análisis de una proposición procedure se muestra en el algoritmo 5.18. Cuando se llama a éste, se asume que el parser ya ha determinado de alguna forma que la proposición empieza con un identificador de procedimiento (algortimo 5.18).

```
PROCEDURE Prop_procedure(Stop: simbolos);
VAR
  stop2 : simbolos; Proc  : Ptro;
BEGIN
  Encuentra(argumento, Proc);
  IF Proc^.tipo = Proc_estandar THEN
     Prop_ES(Stop)
  ELSE
     IF Proc^.ult_param <> NIL THEN
     BEGIN
       stop2 := [PARENT_DER] + Stop;
       Acepta(IDENT, [PARENT_IZQ] + inic_expr + Stop2);
       Acepta(PARENT_IZQ,inic_expr + Stop2);
       Lista_de_param_act(Proc^.ult_param,Stop2);
       Acepta(PARENT_DER, Stop)
     END
     ELSE { No hay lista de parámetros }
       Acepta(IDENT,Stop)
END;
```

Algoritmo 5.18 Análisis de una proposición procedure.

La lista de parámetros actuales se analiza mediante el siguiente procedimiento recursivo (5.19). Este procedimiento sigue la cadena de registros de parámetros y se llama a sí mismo hasta que se ha alcanzado el primer registro del parámetro. Cada parámetro es manejado por una llamada separada al procedimiento. Al final, cada llamada al procedimiento introduce un parámetro actual y compara su tipo con el tipo del parámetro formal correspondiente.

Los procedimientos estándar están descritos por registros de objetos. La parte variante de estos registros consiste de un campo para la clase con el valor proc_estandar.

```
PROCEDURE Lista_de_param_act(ult_param:Ptro;Stop:simbolos);
VAR
  Tipox : Ptro;
BEGIN
  IF ult_param^.ant <> NIL THEN
  BEGIN
    Lista_de_param_act(ult_param^.ant,[COMA]+inic_expr+stop);
    Acepta(COMA,inic_expr + Stop );
  END;
  IF ult_param^.tipo = Param_val THEN
     Expresion(Tipox,Stop)
  ELSE {  ult_param^.tipo = param_var }
     Accesa_var(Tipox,Stop);
  Verifica_tipos(Tipox,ult_param^.tipo_de_var)
END;
```

Algoritmo 5.19 Análisis de la lista de parámetros actuales.

Los registros de los procedimientos estándar son creados antes que el programa fuente sea analizado.

```
VAR
  Proc : Ptro;
BEGIN
  ...
  Define(ord(read1),Proc_estandar,Proc);
  Define(ord(write1),Proc_estandar,Proc);
  ...
```

Los procedimientos estándar son invocados por proposiciones READ y WRITE, las cuales son analizadas por medio del algoritmo 5.20.

```
PROCEDURE Prop_ES(Stop : simbolos);
VAR
  id    : INTEGER;
  Tipox : Ptro;
  Stop2 : simbolos;
BEGIN
  Stop2 := [PARENT_DER] + Stop;
  id    := argumento;
  Acepta(IDENT, inic_expr + Stop2);
  Acepta(PARENT_IZQ, inic_expr + Stop2);
  IF id = ORD(READ1) THEN
     Accesa_var(Tipox,Stop2)
  ELSE  {id = ord(WRITE1) }
     Expresion(Tipox,Stop2);
  Verifica_tipos(Tipox,TipoEntero);
  Acepta(PARENT_DER,Stop2)
END;
```

Algoritmo 5.20 Análisis de tipos para los procedimientos estándar READ y WRITE.

5.10 PRUEBA.

Para probar el análisis de tipos se usaron tres programas: prueba6, prueba7 y prueba8 (ver apéndice A.4).

Prueba6 muestra que el parser puede realizar análisis de tipos de sentencias correctas. Para ilustrar cómo fué construido este programa, examinaremos el procedimiento que analiza un factor (algoritmo 5.12). Para probar sistemáticamente las proposiciones de este procedimiento, debemos usar factores de las siguientes clases:

i) Un número.
ii) Un identificador de constante.
iii) Una variable.
iv) Una expresión entre paréntesis.
v) La negación de una expresión Booleana.

El programa prueba6 incluye estos casos de prueba.

El programa prueba7 contiene errores de tipos. Para ilustrar esta prueba, observaremos nuevamente los factores. El procedimiento Factor reporta un error de tipo sólo cuando un operador not se aplique a un operando que no sea de tipo Boolean; por ejemplo:

```
VAR
  y : BOOLEAN;
        ...
  y := not 1 and 2 and 3;
        ...
```

Después de reportar un error de tipo, el parser asigna el tipo universal al factor negado, para evitar más errores en la misma expresión.

El programa Prueba8 comprende errores de clase. El procedimiento Factor reporta un error de clase si el identificador de un operando no se refiere a una constante o una variable; por ejemplo:

```
VAR
  x : INTEGER;

  PROCEDURE P(...);
  BEGIN  ... END;
BEGIN
  ... x := P; ...
END.
```

Estos tres programas son pruebas completas del análisis de tipos. La prueba de un compilador debe hacerse escribiendo programas que invoquen sistemáticamente todas las partes del compilador. Esto nos permite desarrollar gradualmente el

compilador en fases. En cada fase, podemos ver que los procedimientos desarrollados en etapas anteriores ya funcionan, de tal forma que podamos concentrarnos en probar la parte más reciente del compilador.

El programa del analizador sintáctico ampliado con análisis de alcance y análisis de tipos se muestra a en el apéndice A.3.

A continuación se muestran los resultados de estos tres programas de prueba. Primero se ilustra el listado del programa fuente y después el código intermedio generado.

```
1   { Mini-Pascal. Prueba 6: Análisis de tipos }
2   program prueba6;
3   const
4      a = 10;
5      b = false;
6   type
7      T1 = array[a..a] of integer;
8      T2 = record
9              f, g : integer;
10             h    : Boolean
11           end;
12  var
13     x, y : integer;
14     Z    : Boolean;
15
16     procedure Q(var x: T1; z: T2);
17     begin
18        x[10] := 1;
19        z.f := 1;
20        Q(x,z)
21     end;
22
23     procedure P;
24     begin
25        Read(x);
26        Write(x+1)
27     end;
28
29  begin
30     P;
31     x := 1;
32     x := a;
33     x := y;
34     x := - (x+1) * (y-1) div 9 mod 9;
35     z := not b;
36     z := z or z and z;
37     if x <> y then
38        while x < y do { vacio }
39  end.

    Análisis sintáctico terminado sin errores
    Número de identificadores usados: 13
```

```
42 1
42 2 15 44 1 54
42 3 3
42 4 44 2 36 45 10 54
42 5 44 3 36 44 27 54
42 6 20
42 7 44 4 36 1 47 44 2 56 44 2 50 12 44 24 54
42 8 44 5 36 16
42 9 44 6 52 44 7 55 44 24 54
42 10 44 8 55 44 25
42 11 7 54
42 12 22
42 13 44 9 52 44 10 55 44 24 54
42 14 44 11 55 44 25 54
42 15
42 16 14 44 12 48 22 44 9 55 44 4 54 44 11 55 44 5 49 54
42 17 2
42 18 44 9 47 45 10 50 31 45 1 54
42 19 44 11 44 6 31 45 1 54
42 20 44 12 48 44 9 52 44 11 49
42 21 7 54
42 22
42 23 14 44 13 54
42 24 2
42 25 44 29 48 44 9 49 54
42 26 44 30 48 44 9 32 45 1 49
42 27 7 54
42 28
42 29 2
42 30 44 13 54
42 31 44 9 31 45 1 54
42 32 44 9 31 44 2 54
42 33 44 9 31 44 10 54
42 34 44 9 31 33 48 44 9 32 45 1 49 34 48 44 10 33 45 1 49 4
45 9 10 45 9 54
42 35 44 11 31 11 44 3 54
42 36 44 11 31 44 11 13 44 11 0 44 11 54
42 37 9 44 9 37 44 10 18
42 38 23 44 9 38 44 10 5
42 39 7 53
```

```
1   { Mini-Pascal. Prueba 7: Errores de tipos }
2   program prueba7;
3   type
4      T = array[1..10] of integer;
5   var
6      x : integer;
7      y : Boolean;
8      z : T;
9
10      procedure P(x : integer);
11      begin
12      end;
13
14   begin
15      y := not 1 and 2 and 3;

* Error en linea 15:  Identificador de tipo inválido

16      y := false * true div false;

* Error en linea 16:  Error de sintaxis

17      z := z mod z;

* Error en linea 17:  Identificador de tipo inválido

18      x := 1 or 2 or 3;

* Error en linea 18:  Identificador de tipo inválido

19      y := false + true - true;

* Error en linea 19:  Identificador de tipo inválido

20      z := z - z;

* Error en linea 20:  Identificador de tipo inválido

21      if z <> z then

* Error en linea 21:  Identificador de tipo inválido

22         P(true)

* Error en linea 22:  Identificador de tipo inválido

23   end.

     8 Error(es) encontrado(s)
     Número de identificadores usados: 6

42 1
42 2 15 44 1 54
42 3 20
```

```
     42 4 44 2 36 1 47 45 1 56 45 10 50 12 44 24 54
     42 5 22
     42 6 44 3 55 44 24 54
     42 7 44 4 55 44 25 54
     42 8 44 5 55 44 2 54
     42 9
     42 10 14 44 6 48 44 3 55 44 24 49 54
     42 11 2
     42 12 7 54
     42 13
     42 14 2


1    { Mini-Pascal. Prueba 8: Errores de clase }
2    program prueba8;
3    const
4       a = integer;

* Error en linea 4:  Identificador de clase incorrecta

5    type
6       T = array[2..1] of integer;

* Error en linea 6:  Indice de rango inválido

7       U = record
8             f : integer
9           end;
10   var
11      x : integer;
12      y : U;
13      z : false;

* Error en linea 13:  Identificador de clase incorrecta

14
15      procedure P(var x : integer y : true);

* Error en linea 15:  Error de sintaxis

16      begin
17      end;
18
19   begin
20      x[1] := 1;

* Error en linea 20:  Identificador de clase incorrecta

21      x.f := 1;

* Error en linea 21:  Identificador de clase incorrecta

22      P(false, true);
```

```
* Error en linea 22:  Error de sintaxis
23      x := P;
* Error en linea 23:  Identificador de clase incorrecta
24      false := true;
* Error en linea 24:  Identificador de clase incorrecta
25      y.g := 1
* Error en linea 25:  Identificador indefinido
26   end.

     10 Error(es) encontrado(s)
     Número de identificadores usados: 10


     39 1
     39 2 14 41 1 51
     39 3 3
     39 4 41 2 33 41 21
```

CAPITULO 6. GENERACION DE CODIGO.

INTRODUCCION.

Este Capitulo describe el conjunto de instrucciones para una computadora hipotética que opera con una pila y explica la forma en que el compilador genera código para esta computadora. El generador de código está dividido en dos partes: (1) una extensión del parser, el cual genera código Mini-Pascal; y (2) un ensamblador, el cual define referencias hacia adelante y optimiza el código. Este es el método usado en el UCSD PASCAL que al estar escrito en código de la máquina ficticia, sólo se precisa tener un pequeño intérprete distinto para cada máquina objeto dada.

6.1 UNA COMPUTADORA IDEAL.

En vez de realizar un compilador para una computadora particular, inventaremos una computadora ideal para nuestro compilador. Esta nueva computadora se llamará computadora Mini-Pascal. El código generado para esta computadora se denominará código Mini-Pascal.

La computadora Mini-Pascal es ideal en el siguiente sentido:

(1) Las instrucciones Mini-Pascal corresponden directamente a los conceptos del lenguaje Mini-Pascal.

(2) El código Mini-Pascal de un programa tiene prácticamente la misma sintaxis que el programa mismo. Consecuentemente, el generador de código es una extensión trivial del parser.

(3) Una computadora Mini-Pascal implementada en hardware puede ejecutar programas en Mini-Pascal más rápido que la mayoría de las computadoras tradicionales.

6.2 LA PILA.

La memoria de la computadora Mini-Pascal es un arreglo de enteros. Los elementos de esta memoria y sus índices se conocen como palabras y direcciones.

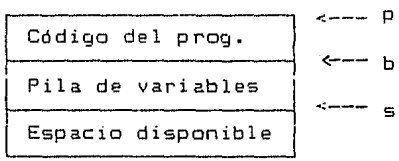


Fig. 6.1 La memoria y sus registros.

Esta memoria guarda el código y las variables de un programa en Mini-Pascal (figura 6.1). El código, el cual es de longitud fija, se coloca al principio de la memoria y el resto de ésta se

usa como una pila de variables. Durante la ejecución de proposiciones, la pila también guarda resultados temporales.

La computadora tiene tres registros índice, denominados p, b y s. El registro de programa p contiene la dirección de la instrucción actual. El registro base b se usa para accesar variables. El registro de pila s, guarda la dirección del tope de la pila.

En Mini-Pascal, la memoria y los registros índice están definidos de la siguiente forma:

```
CONST
  min = 0;
  max = 2000;
TYPE
  almacenamiento = ARRAY[min..max] OF INTEGER;
VAR
  st     : almacenamiento;
  p,b,s  : INTEGER;
```

Con estas definiciones, una localidad de memoria con la dirección x es denotada como st[x].

Las variables de un bloque se guardan en el segmento de pila conocido como registro de activacion (figura 6.2). Este registro consiste de cuatro partes: La parte de parámetros, la parte de contexto, la parte de variables y la parte temporal.

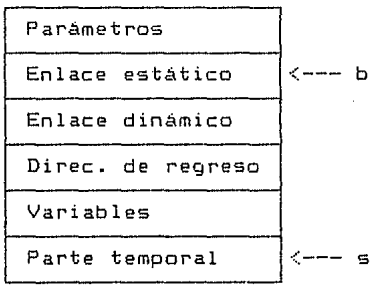


Fig. 6.2 Estructura de un registro de activación.

La figura 6.3 muestra el registro de activación del siguiente procedimiento:

```
PROCEDURE Quicksort(m,n : INTEGER);
VAR
  i,j : INTEGER;
BEGIN
  ...
END;
```

La parte de parámetros contiene localidades de memoria para los parámetros formales m, n.

La parte de contexto contiene tres direcciones, llamadas enlace estático, enlace dinámico, y la dirección de regreso. Estas direcciones definen el contexto en el cual se activó el procedimiento. El registro b contiene la dirección del enlace estático. Esta dirección se llama dirección base del registro de activación.

La parte de variables contiene las localidades para las variables locales i y j.

La parte temporal guarda los operandos y resultados durante la ejecución de las proposiciones.

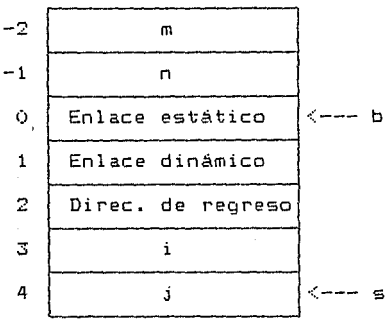


Fig. 6.3 Ejemplo de un registro de activación.

Si un procedimiento es activado recursivamente, cada activación crea otra instancia del registro de activación.

En Mini-Pascal, cada variable es de un tipo fijo. Este puede ser estándar, arreglo o registro. Una variable de tipo estándar ocupa una palabra. Dado que los arreglos y registros son combinaciones de un número fijo de elementos de los tipos estándar, entonces cada variable ocupa un número fijo de palabras.

Las definiciones de tipos permiten que el compilador pueda calcular la longitud de estas variables. Combinando esta información, el compilador puede calcular la dirección relativa de cada variable en un registro de activación. Las direcciones relativas son desplazamientos relativos a la dirección base del registro de activación (ver figura 6.3).

En el ejemplo anterior, las variables tienen el siguiente desplazamiento:

Variable	Desplazamiento
m	-2
n	-1
i	3
j	4

Al activar un procedimiento, la computadora crea un registro de activación y hace que el registro b apunte a la dirección base de éste. Cualquier variable dentro del registro puede ser accesada agregando este desplazamiento al valor del registro b:

Variable	Dirección
m	b-2
n	b-1
i	b+3
j	b+4

El siguiente fragmento de programa incluye el procedimiento Quicksort anterior. Este programa es un buen ejemplo del uso de bloques anidados y la recursión:

```
Program Prueba9;
Const
  max = 10;
Type
  T = array[1..max] of integer;
var
  A : T;
  k : integer;

  Procedure Quicksort(m,n : integer);
  var
    i,j : integer;

    Procedure Particion;
    var
      r,w : integer;
    begin ... end;
  Begin
    if m < n then
    begin
      Particion;
      Quicksort(m,j);
      Quicksort(i,n);
    end;
  End;
Begin
  ...
  Quicksort(1,max);
  ...
End.
```

Observemos la pila en el momento en el que se activan los siguientes bloques:

(1) El bloque del programa principal.
(2) El procedimiento Quicksort (primera activación).
(3) El procedimiento Quicksort (segunda activación).
(4) El procedimiento Quicksort (tercera activación).
(5) El procedimiento Partición.

La figura 6.4 muestra los registros de activación correspondientes. El bloque del programa principal se trata como un procedimiento sin parámetros. Cuando termina un procedimiento, la computadora debe eliminar de la pila el registro de activación correspondiente. Para lograr esto, cada registro de activación se enlaza al registro anterior. El enlace dinámico de un registro de activación contiene la dirección base del registro de activación anterior. Cuando un procedimiento termina, el enlace dinámico que está almacenado en el registro de activación actual se asigna al registro b. La cadena de enlaces dinámicos se conoce como cadena dinámica, debido a que define la secuencia dinámica en la cual se han activado los bloques.

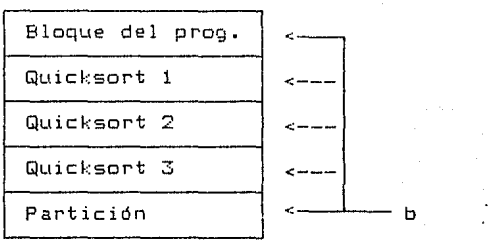


Fig. 6.4 La cadena dinámica.

Los enlaces estáticos definen el conjunto de variables que son accesibles en el bloque actual. Este conjunto de variables se llama contexto actual del programa. En la situación anterior, el contexto actual consiste de las variables creadas durante las siguientes activaciones:

(1) La activación más reciente del procedimiento Partición.
(2) La activación más reciente del procedimiento Quicksort.
(3) La activación más reciente del bloque del programa.

Los registros de activación que contienen estas variables son enlazados mediante enlaces estáticos (figura 6.5). El registro de activación de Partición incluye un enlace estático que apunta al tercer registro de activación de Quicksort. El enlace estático de este registro, a su vez apunta al registro de activación del bloque del programa. Esta cadena de enlaces se conoce como la cadena estática actual. Se llama "estática" debido a que representa la estructura estática del bloque del programa fuente.

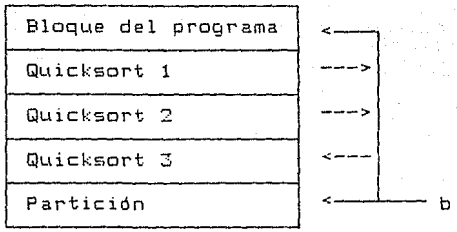


Fig. 6.5 La cadena estática.

En general, cada activación de un bloque puede tomar lugar en un contexto diferente. Consecuentemente, cada registro de activación es el inicio de una cadena estática separada. En la situación anterior, cada registro de activación de Quicksort apunta al registro de activación del bloque del programa.

Sin embargo, en algún momento dado el contexto actual está definido por una sola cadena estática que inicia con el registro b. La figura 6.6 muestra los registros de activación que son accesibles en el contexto actual.

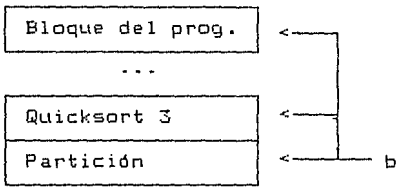


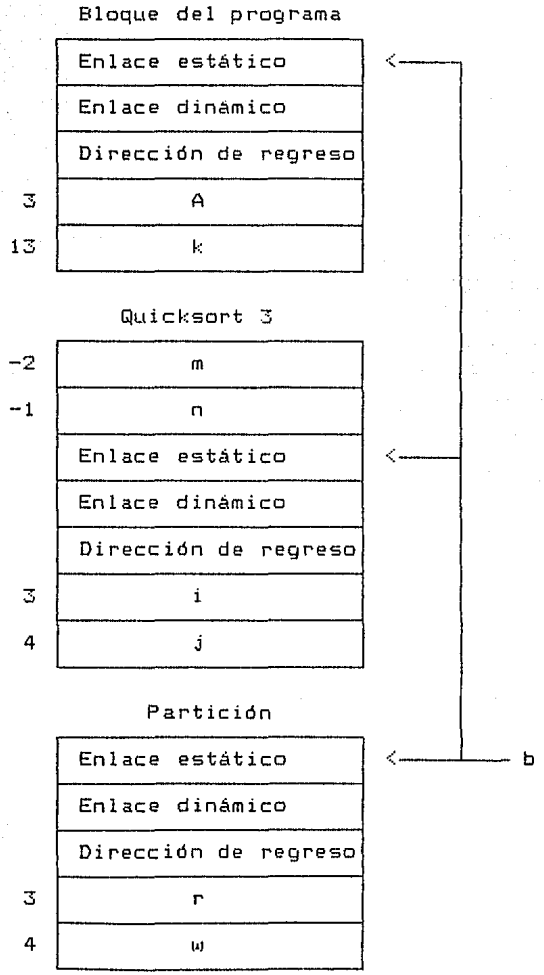
Fig. 6.6 El contexto actual.

6.3 ACCESO A VARIABLES.

La figura 6.7 es una imagen más detallada del contexto mostrado en la figura 6.6. Para accesar una variable en este contexto, el código del programa debe especificar (1) el registro de activación que contiene la variable, y (2) el desplazamiento de la variable dentro del registro.

Durante el acceso a una variable es más conveniente identificar a ésta por un número de nivel que sea relativo al bloque actual. Este número se obtiene restando el número de nivel de la variable del número de nivel del bloque actual. Aqui hay algunos ejemplos de la forma en la que son identificadas las variables dentro del procedimiento partición:

Variable	Número relativo	Desplazamiento
A	2	3
m	1	-2
w	0	4



Variable	Nivel	Desp.
A	1	3
k	1	13
m	2	-2
n	2	-1
i	2	3
j	2	4
r	3	3
w	3	4

Fig. 6.7 Contexto actual en detalle.

Cuando un programa hace referencia a una variable por su identificador, el código correspondiente hace referencia a ésta mediante una instrucción de la forma:

Variable(Nivel, Desplazamiento)

La instrucción consiste de dos partes:

(1) Un código de operación que le indica a la computadora calcular la dirección de una variable.

(2) Dos argumentos que definen el nivel (relativo) y el desplazamiento de la variable.

El código de operación y los argumentos ocupan una palabra cada uno (figura 6.8). Durante la ejecución de la instrucción, el registro de programa p apunta al código de operación.

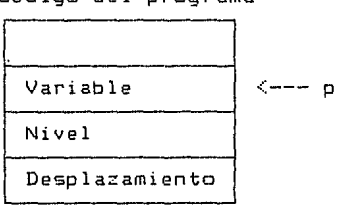


Fig. 6.8 Instrucción Mini-Pascal.

En el procedimiento Partición, las referencias a las variables A, m y w son compiladas en estas instrucciones:

Variable	Instrucción
A	Variable(2,3)
m	Variable(1,-2)
w	Variable(0,4)

La computadora localiza la variable A en cinco pasos:

(1) El registro de pila, s, se incrementa en uno para crear una nueva localidad temporal en el tope de la pila.

(2) La dirección base de la variable se encuentra dos niveles abajo de la cadena estática.

(3) La dirección absoluta de la variable se calcula sumando la dirección base y el desplazamiento 3.

(4) La dirección absoluta se almacena en la nueva localidad temporal.

(5) El registro de programa, p, es incrementado en 3 para hacer que apunte a la siguiente instrucción.

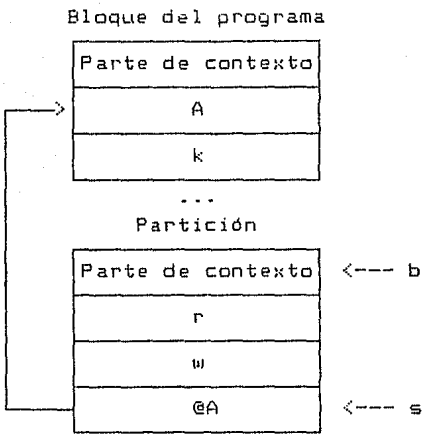


Fig. 6.9 Resultado del acceso a una variable.

La figura 6.9 muestra el resultado de estas acciones. La dirección de la variable arreglo A es la dirección de su primer palabra. Esta dirección se denota como @A y permanece en la pila hasta que se ha usado para su propósito.

El algoritmo 6.1 define la instrucción variable. Los parámetros formales de este procedimiento denotan los argumentos de la instrucción. La variable local x representa un registro de trabajo que se usa durante la ejecución de la instrucción.

```
Procedure Variable(nivel, desp : integer);
var
  x : integer;
begin
  s := s + 1;
  x := b;
  while nivel > 0 do
  begin
    x := St[x];
    nivel := nivel -1
  end;
  St[s] := x + desp;
  p := p + 3
end.
```

Algoritmo 6.1 Instrucción Variable.

Las instrucciones variable se usan sólo para accesar parámetros por valor y variables locales. Los parámetros variables

se accesan en forma diferente. El programa siguiente incluye un parámetro variable x (en el procedimiento Q):

```
program p;
var
  v : integer;
  procedure Q(var x : integer);
  begin
    ...
    x
    ...
  end;
begin
  ...
  Q(v);
  ...
end.
```

El bloque del programa incluye una proposición procedure Q(v) la cual enlaza el parámetro x a la variable v, mientras se ejecuta el procedimiento Q. Esto significa que todas las operaciones sobre x se realizan realmente sobre v. En el registro de activación del procedimiento, el parámetro x está representado por una palabra que contiene la dirección absoluta de la variable v (figura 6.10).

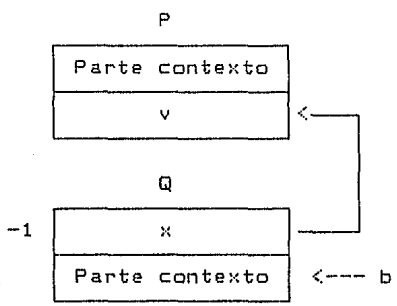


Fig. 6.10 Parámetro variable.

Una referencia al parámetro x dentro de Q es realmente una referencia a la variable v. El efecto de ejecutar la instrucción:

```
Variable(0,-1)
```

es colocar la dirección de la localidad del parámetro x en el tope de la pila. Pero necesitamos la dirección de la variable que apunta a esa localidad. De esta forma debemos introducir otra instrucción:

```
ParamVar(0,-1)
```

la cual está definida por el algoritmo 6.2:

```
ParamVar(nivel, desp : integer);
var
  x : integer;
begin
  s := s + 1;
  x := b;
  while nivel > 0 do
  begin
    x := St[x];
    nivel := nivel - 1
  end;
  St[s] := St[x + desp];
  p := p + 3
end;
```

Algoritmo 6.2 Instrucción ParamVar.

Como hemos visto, las diferentes clases de variables se accesan mediante distintos tipos de instrucciones. Cuando una sentencia en Mini-Pascal puede expresarse en varias formas diferentes, podemos caracterizar todas las formas posibles mediante una gramática.

Para aplicar esta idea poderosa en la generación de código, debemos ver el código Mini-Pascal como un lenguaje. Los símbolos de este lenguaje son instrucciones de computadora, tales como Variable y ParamVar. Las sentencias del código del lenguaje son secuencias de instrucciones que representan sentencias Mini-Pascal. La posible secuencia de instrucciones debe estar definida por reglas sintácticas escritas en notación BNF. Estas reglas sintácticas se llamarán reglas código.

Para aplicar esta idea sistemáticamente, debemos seguir unas cuantas reglas:

Regla 6.1. Para cada regla sintáctica en Mini-Pascal, debemos escribir una regla código que defina las secuencias de instrucciones correspondientes.

Regla 6.2. Cada instrucción preferiblemente, deberá tener el mismo identificador que representa el símbolo Mini-Pascal.

Regla 6.3. Una regla código deberá tener la misma estructura sintáctica en Mini-Pascal.

Como ejemplo, el acceso a una variable tiene la siguiente sintaxis en Mini-Pascal:

```
<Accesa_var> ::= <Id. de variable> {<Selector>}
<Selector>   ::= [<Expresión>] | . <Id. de campo>
```

Las reglas código correspondientes son muy similares:

```
<Accesa_var> ::= <Id. de variable> {<Selector>}
<Id. de variable> ::= Variable | ParamVar
<Selector>   ::= <Expresión> Indice | Campo.
```

El significado de estas reglas es el siguiente:

(1) El código para accesar una variable consiste del código para un identificador de variable, al cual le puede seguir el código para uno o más selectores.

(2) El código para un identificador de variable es una instrucción Variable o una instrucción ParamVar.

(3) El código para un selector es el código para una expresión, seguido por una instrucción Indice o solamente una instrucción Campo.

Para el acceso a los campos de un registro y a los elementos de un arreglo se utiliza un procedimiento similar al que se describió para accesar variables.

6.4 SINTAXIS DEL CODIGO MINI-PASCAL.

La siguiente es una lista de las reglas código, las cuales corresponden a las reglas sintácticas de Mini-Pascal (sección 4.1).

```
<Programa>     ::= Programa <Bloque> FinProg

<Bloque>       ::= {<Def. de procedimiento>} <Prop. compuesta>

<Def. de procedimiento> ::= Procedure <Bloque> FinProc

<Prop.>        ::= <Prop. de asignación> | <Prop. procedure> |
                   <Prop. if> | <Prop. while> |
                   <Prop. compuesta> | <cadena vacía>

<Prop. de asignación> ::= <Accesa var> <Expresión> Asigna

<Prop. procedure>   ::= <Prop. de E/S> | [<Lista param. act>]
                        LlamadaProc

<Prop. de E/S>      ::= <Accesa var> Read | <Expresión> Write

<Lista param. act>  ::= <Parámetro actual>
                        {<Parámetro actual>}

<Parámetro actual>  ::= <Expresión> | <Accesa_var>
<Prop. if>          ::= <Expresión> Do <Prop.> [Goto <Prop.>]
<Prop. while>       ::= <Expresión> Do <Prop.> Goto
<Prop. compuesta>   ::= <Prop.> {<Prop>}
<Expresión> ::= <Exp. simple> [<Exp. simple><Op. relacional>]
<Op. relacional>    ::= Menor_que | Menor_o_igual | Igual |
                        Mayor_que | Mayor_o_igual | Diferente
```

```
<Exp. simple>        ::= <Término> [<Signo>] {<Término>
                         <Op. sumador>}
<Signo>              ::= Menos | <Cadena vacía>
<Op. sumador>        ::= Mas | Resta | Or
<Término>            ::= <Factor> {<Factor> <Op. mult>}
<Op. mult>           ::= Mult | Divide | Módulo | And
<Factor>             ::= Constante | <Accesa_var> Valor |
                         <Expresión> | <Factor> Not

<Accesa_var>         ::= <Id. de variable> {<Selector>}
<Id. de variable>    ::= Variable | ParamVar
<Selector>           ::= <Expresión> Indice | Campo
```

6.5 EJECUCION DE PROPOSICIONES.

Una proposición de asignación produce el siguiente código:

<Prop. de asignación> ::= <Accesa_var> <Expresión> **Asigna.**

El código accesa_var coloca en la pila la dirección de una variable. El código para expresión coloca el valor de una expresión en la dirección de la variable. La instrucción Asigna elimina los dos operandos de la pila y asigna el valor a la variable.

El siguiente ejemplo define dos variables A y B del mismo tipo T:

```
TYPE
  T = ARRAY[1..10] OF INTEGER;
VAR
  A,B : T;
```

La proposición de asignación A:=B produce el siguiente código:

```
Variable(0,3)
Variable(0,13)
Valor(10)
Asigna(10)
```

Una proposición while:

```
while B do S
```

produce código de la forma:

```
L1: B
    Do(L2)
    S
    Goto(L1)
L2:
```

El código se ejecuta en los siguientes pasos:

(1) Se evalua el código de la expresión B para obtener en la pila un valor Booleano.

(2) La instrucción Do elimina de la pila el valor Booleano. Si el valor es verdadero, la computadora procede con el paso 3. En caso contrario, la ejecución de la proposición while termina saltando al punto etiquetado L2.

(3) Se ejecuta el código de la proposición S y salta a L1 para repetir el paso 1.

La siguiente regla describe el código anterior:

<Prop. while> ::= <Expresión> **Do** <Prop.> **Goto**

La computadora Mini-Pascal siempre carga el código del programa al principio de la memoria (Fig. 6.1). Puesto que la computadora conoce la dirección y la longitud de cada instrucción, podría generar instrucciones de salto con direcciones absolutas. Pero en la mayoria de los sistemas, la dirección de un programa se desconoce hasta que éste ha sido cargado mediante el sistema operativo. Dado que un compilador debe tratar con esta incertidumbre, diseñaremos el compilador Mini-Pascal para producir código que pueda ser colocado en cualquier lugar de la memoria. Este se llama código reubicable.

La forma más sencilla de hacer código reubicable es dejar que cada instrucción de salto defina su destino mediante un desplazamiento relativo a la instrucción misma. Cuando se ejecute una instrucción de salto, la dirección destino se obtiene agregando el desplazamiento al registro p.

Lo anterior se logra mediante la instrucción Goto (algoritmo 6.3).

```
Procedure Gotox(desp : integer);
begin
  p := p + desp
end;
```

Algoritmo 6.3 Instrucción **Goto.**

El tipo más simple de la proposición if:

```
if B then S
```

se compila en el siguiente código:

```
B
  Do(L)
  S
L:
```

Si el código de la expresión B da un valor falso, la instrucción Do salta a L. En caso contrario, se ejecuta el código de la proposición S.

Una proposición if de la forma:

```
if B then S1 else S2
```

se compila en el siguiente código:

```
    B
    Do(L1)
    S1
    Goto(L2)
L1: S2
L2:
```

Si el valor de B es verdadero, se ejecuta el código de la proposición S1 antes que la instrucción Goto salte a L2. En caso contrario, la instrucción Do salta a L1, donde será ejecutado el código de la proposición S2.

La siguiente regla código describe ambas formas de la proposición if:

```
<Prop. if> ::= <Expresión> Do <Prop.>
               [Goto <Prop.>]
```

La regla código para una proposición arbitraria es la siguiente:

```
<Prop.> ::= <Prop. de asignación> | <Prop. procedure> |
            <Prop. if> | <Prop. compuesta> | <Prop. vacía>
```

El código para una proposición compuesta:

```
begin
  S1;
  S2;
  ...
  Sn
end;
```

consiste del código para las proposiciones $S_1, S_2, \ldots, S_n$. En otras palabras, la regla código es simplemente:

```
<Prop. compuesta> ::= <Prop.> {<Prop.>}
```

Para la evaluación de expresiones se utiliza la notación postfija, debido a que ésta es apropiada para una computadora con pila y se sigue un procedimiento similar al de la ejecución de proposiciones.

6.6 CODIGOS DE OPERACION.

Cada instrucción generada por el compilador consiste de un código de operación seguido por cero o más argumentos. Los códigos de operación están representados por valores del siguiente tipo:

```
TYPE
  cod_op = (MAS2, AND2, ASIGNA2, CONSTANTE2, DIVIDE2, DO2,
     FINPROC2, FINPROG2,  IGUAL2, CAMPO2, GOTO2, MAYOR_QUE2,
     INDICE2, MENOR_QUE2, MENOS2, MODULO2,MULTIPLICA2, NOT2,
     DIFERENTE2,MENOR_O_IGUAL2,MAYOR_O_IGUAL2,LLAMADA_PROC2,
     OR2,  PROCEDURE2,  PROGRAM2, SUB2,  VALOR2,  VARIABLE2,
     PARAM_VAR2,  READ2, WRITE2, DEFINE_DIREC2, DEFINE_ARG2,
     LLAMADA_GLOBAL2,VALOR_GLOBAL2,VAR_GLOBAL2,VALOR_LOCAL2,
     VARIABLE_LOCAL2, ASIGNACION_SIMPLE2, VALOR_SIMPLE2);
```

El parser emite una instrucción de la forma:

```
Variable(Nivel,Desp)
```

por medio de la proposición:

```
Emite3(Variable, Nivel, Desp);
```

El parser incluye otros tres procedimientos de salida denominados Emite1, Emite2, y Emite5.

6.7 DIRECCIONAMIENTO DE VARIABLES.

El direccionamiento de variables se lleva a cabo mediante números de nivel relativos y desplazamientos. Usaremos el siguiente procedimiento para mostrar como se hace esto:

```
PROCEDURE Quicksort(m,n : INTEGER);
VAR
  i,j : INTEGER;

  PROCEDURE Particion;
  BEGIN
    ...
    j := n;
    ...
END;
```

El procedimiento QuickSort tiene dos parámetros formales m y n y dos variables locales i y j. Partición es un procedimiento local de Quicksort.

Al activarse Quicksort, se crea una instancia de sus parámetros y variables locales en la forma de un registro de activación (ver figura 6.3).

Una referencia dentro de Partición a la variable global j genera la instrucción:

```
Variable(1,4)
```

El primer argumento de la instrucción muestra que la variable j se almacena un nivel arriba de las variables de Partición (figura 6.7). El segundo argumento muestra que j se almacena cuatro palabras abajo de la dirección base del registro de activación.

Para compilar esta instrucción, debemos extender el registro de un objeto para una variable con el número de nivel absoluto y un desplazamiento:

```
TYPE
  Reg_obj = RECORD
              id    : integer;
              ant   : Ptro;
              CASE clase : clas of
                   ...
                 variable, param_val, param_var : (Nivel_var,
                 DespVar : integer);
                   ...
            END;
```

El procedimiento que analiza la lista de variables:

```
i,j : integer;
```

almacena los identificadores y el tipo de variables i y j en dos registros de objetos. El procedimiento también cuenta el número de variables de la lista y devuelve punteros a los registros que describen la última variable j y el tipo entero (algoritmo 5.3).

```
PROCEDURE Lista_variables(clase : clas; var num : integer;
            var ult_var, tipox : Ptro; stop : simbolos);
```

El procedimiento Def_de_variable usa esta información para calcular la longitud de la lista de variables (algoritmo 6.4).

```
PROCEDURE Def_de_variable(VAR ult_var : Ptro;
                          VAR long : INTEGER; stop : simbolos);
VAR
   num : INTEGER;
   tipox : Ptro;
BEGIN
   Lista_de_variables(variable, num, ult_var, tipox,...);
   longitud := num*long_del_tipo(tipox);
   acepta(PUNTOYCOMA, Stop)
END;
```

Algoritmo 6.4 Direccionamiento de variables.

Un valor estándar tiene una longitud de una palabra. La longitud de un valor arreglo se obtiene multiplicando el número de

elementos del arreglo por la longitud de un elemento. La longitud de un valor registro se almacena en la parte variante del registro de objeto que describe el tipo registro:

```
TipoRecord : (Long_reg : integer; ult_campo : Ptro);
```

El procedimiento que analiza la parte de definición de variable:

```
var i,j : integer;
```

está definido por el algoritmo 6.5:

```
PROCEDURE Parte_def_variable(var long_reg:integer;ult_campo:Ptro);
VAR
  ult_var : Ptro;
  mas : integer;
BEGIN
  Acepta(VAR1,...);
  Def_de_variable(ult_var, long,...);
  WHILE simbolo = id DO
  BEGIN
    Def_de_variable(ult_var, long,...);
    long := long + mas
  END;
  Direc_var(long,ult_var);
END;
```

Algoritmo 6.5 Análisis de la parte de def. de variables.

Al final de la parte de definición de variables, el parser llama a un procedimiento que reconoce las variables (algoritmo 6.6) y les asigna los desplazamientos mostrados en la figura 6.3. Las variables son desplazadas mediante la parte contextual del registro de activación, la cual ocupa tres palabras.

```
PROCEDURE Direc_var(long_var : integer; ult_var : Ptro);
VAR
  Desp : integer;
BEGIN
  Desp := 3 + long_var;
  WHILE Desp > 3 DO
  BEGIN
    Desp := Desp - long_del_tipo(ult_var^.tipo_var);
    ult_var^.nivel_var := nivel_block;
    ult_var^.desp_var := Desp;
    ult_var := ult_var^.ant
  END;
END;
```

Algoritmo 6.6 Asignación de desplazamientos a variables.

El direccionamiento de parámetros formales se realiza en forma similar al direccionamiento de variables.

6.8 CODIGO PARA EXPRESIONES.

Consideremos un bloque con dos variables locales:

```
var y,z : integer;
```

y la expresión:

```
y*z div 5
```

Esta se compila en las siguientes instrucciones:

```
Variable(0,3)
Valor(1)
variable(0,4)
Valor(1)
Mult
Constante(5)
Divide
```

Sigamos paso a paso la compilación de la expresión:

(1) Cuando el parser encuentra la expresión, llama a cuatro procedimientos denominados Expresión, ExpresiónSimple, Término y Factor, en este orden. Factor reconoce el identificador de variable y ejecuta las siguientes proposiciones:

```
AccesaVar(Tipox,Stop);
Long := Long_del_tipo(Tipox);
Emite2(Valor2, Long);
Push(Long - 1);
```

La llamada a AccesaVar genera la primer instrucción:

```
Variable(0,3)
```

El procedimiento Factor produce la siguiente:

```
Valor(1)
```

(2) Cuando el procedimiento Término reconoce el operador de multiplicación, éste entra en el siguiente ciclo, el cual almacena temporalmente el operador y llama a Factor nuevamente.

```
Factor(Tipox,...);
while simbolo in simb_multiplicadores do
begin
  operador := simbolo;
  Acepta(simbolo,...);
  Factor(Tipo2,...);
  ...
end;
```

(3) Esta vez, Factor reconoce el identificador de variable z y genera las dos instrucciones siguientes:

```
Variable(0,4)
Valor(1)
```

(4) Posteriormente, el procedimiento Término ejecuta las siguientes proposiciones del ciclo anterior:

```
IF Tipox = TipoEntero THEN
BEGIN
  Verifica_tipos(Tipox,Tipo2);
  CASE operador OF
       POR       : Emite1(MULTIPLICA2);
       DIVISION  : Emite1(DIVIDE2);
       MOD1      : Emite1(MODULO2);
  ELSE { operador = AND1 }
       Error_de_tipo(Tipox);
  END; { CASE }
  Pop(1)
END
```

El efecto de esto es generar el operador de multiplicación como una instrucción Mult.

(5) Cuando el procedimiento Término encuentra el operador de división, éste lo almacena temporalmente y llama a Factor nuevamente. El procedimiento Factor ahora reconoce la constante 5:

```
Constante(Valor, Tipox,Stop);
Emite2(CONSTANTE2,Valor);
Push(1)
```

y emite una instrucción Constante(5).

6) Finalmente, el procedimiento Término genera el operador de división como una instrucción Divide.

El procedimiento término se muestra en el algortimo 6.7.

```
PROCEDURE Termino(VAR tipox : Ptro; Stop : simbolos);
VAR
  operador : t_simb;
  Tipo2    : Ptro;
BEGIN
  Factor(Tipox,Stop);
  WHILE simbolo IN simb_multiplicadores DO
  BEGIN
    operador := simbolo;
    Acepta(simbolo,inic_de_factores + Stop);
    Factor(Tipo2,Stop);
    IF Tipox = TipoEntero THEN
    BEGIN
      Verifica_tipos(Tipox,Tipo2);
      CASE operador OF
           POR       : Emite1(MULTIPLICA2);
           DIVISION  : Emite1(DIVIDE2);
```

```
            MOD1      : Emite1(MODULO2);
        ELSE { operador = AND1 }
            Error_de_tipo(Tipox);
        END; { CASE }
        Pop(1)
      END
      ELSE
        IF Tipox = TipoBoolean THEN
        BEGIN
          Verifica_tipos(Tipox,Tipo2);
          IF operador = AND1 THEN
             Emite1(AND2)
          ELSE { operador aritmetico }
            Error_de_tipo(Tipox);
          Pop(1)
        END
        ELSE
          Error_de_tipo(Tipox)
    END
  END;
```

Algoritmo 6.7. Generación de código para términos.

Durante el análisis de alcance, el parser mantiene una pila de registros de bloques (sección 4.4). Esta pila describe todos los objetos que son conocidos en el bloque actual. Cuando el parser introduce un bloque en el programa fuente, se coloca en la pila un nuevo registro de bloque. Al final del bloque, se borra el registro de la pila.

El registro de un bloque es ampliado con dos nuevos campos denominados Long_temp y num_max_loc:

```
TYPE
  Reg_del_block = RECORD
                    Long_temp, Num_max_loc : INTEGER;
                    ult_objeto : Ptro
                  END;

  Tab_de_blocks = ARRAY[0..num_nivel] OF Reg_del_block;

  VAR
    block : Tab_de_blocks;
    nivel_block, argumento : INTEGER;
```

Cuando el compilador genera una instrucción, éste pronóstica que tanto cambiará el tamaño de la pila cuando se ejecute la instrucción. El parser almacena la longitud actual del área temporal en el campo Long_temp. El número máximo de localidades temporales que se usan en el bloque actual se almacena en el campo num_max_loc.

En el momento que el parser emite una instrucción para una variable, éste llama al procedimiento Push para indicar que la ejecución de esta instrucción incrementará la pila en una palabra.

Después de emitir una instrucción Mult, el parser llama al procedimiento Pop para indicar que la ejecución de esta instrucción decrementará en una palabra el valor de Long_temp del bloque actual.

6.9 CODIGO PARA PROPOSICIONES.

El código de una proposición de asignación tiene la forma:

<Prop. asignación> ::= <Accesa_var> <Expresión> **Asigna**

Anteriormente, discutimos el análisis de tipos para proposiciones de asignación (sección 5.8). El algortimo 6.8 muestra el procedimiento de análisis ampliado con generación de código:

```
PROCEDURE Prop_asignacion(Stop: simbolos);
VAR
  Tipo_de_la_var, Tipo_de_la_expr : Ptro;
  Long : INTEGER;
BEGIN
  Accesa_var(tipo_de_la_var, [ASIGNACION] + inic_expr + Stop);
  Acepta(ASIGNACION, inic_expr + Stop);
  Expresion(Tipo_de_la_expr,Stop);
  Verifica_tipos(tipo_de_la_var,Tipo_de_la_expr);
  Long := Long_del_tipo(Tipo_de_la_expr);
  Emite2(ASIGNA2,Long);
  Pop(1 + Long)
END;
```

Algortimo 6.8 Generación de código para proposiciones.

Durante la ejecución de una proposición de asignación, la computadora coloca en la pila dos valores temporales: (1) la dirección de una variable, y (2) el valor de una expresión. Los valores temporales son eliminados posteriormente mediante la instrucción Asigna.

El código para una proposición while:

```
while B do S
```

incluye dos proposiciones de salto:

```
L1: B
    Do(L2)
    S
    Goto(L1)
L2:
```

La instrucción Do se refiere al punto del programa etiquetado con L2.

El parser explora el programa fuente de izquierda a derecha y genera el código objeto directamente en disco. Cuando el parser está listo para emitir la instrucción Do, aún no se ha compilado la proposición S. Por lo tanto, es imposible generar la dirección de L2.

El Problema de referencias hacia adelante se resuelve en tres pasos:

(1) El parser asigna una etiqueta numérica distinta a cada destino de un salto. Supongamos que a los puntos L1 y L2 del programa se les han asignado las etiquetas 17 y 18. En este caso el parser genera el siguiente código intermedio para la proposición while:

```
DefineDirec(17)
Código para B
Do(18)
Código para S
Goto(17)
DefineDirec(18)
```

(2) El ensamblador explora el código intermedio y se encarga de calcular la dirección de la instrucción actual. La dirección de la instrucción se calcula relativa al inicio del programa.

Supongamos que las instrucciones de la proposición while tienen las siguientes direcciones:

Dirección	Código
279	DefineDirec(17)
279	Código para B
287	DO(18)
289	Código para S
320	Goto(17)
322	DefineDirec(18)

El ensamblador almacena en una tabla la dirección del destino de cada salto. Cuando el ensamblador introduce la instrucción DefineDirec(17), almacena la dirección actual en la entrada 17 de la tabla:

```
Tabla[17] = 279
```

Durante esta exploración del código, el ensamblador define las direcciones de todas las etiquetas, pero no produce código. Las instrucciones DefineDirec sólo sirven para definir direcciones de saltos. cuando se introducen estas pseudoinstrucciones, las direcciones actuales permanecen sin cambio y no se produce nada.

(3) Posteriormente, el ensamblador explora nuevamente el mismo código y utiliza la tabla para producir código final.

En el momento que el ensamblador encuentre la instrucción Do(18), calculará el desplazamiento de la etiqueta 18 relativo a la misma instrucción Do:

```
DespDo = Tabla[18]-287
       = 322 - 287
       = 35
```

y produce la instrucción con este desplazamiento:

```
Do(35)
```

El desplazamiento de la instrucción Goto se calcula de la misma forma.

El código final es el siguiente:

```
Código para B
Do(35)
Código para S
Goto(-41)
```

El parser usa una variable para contar el número de etiquetas creadas hasta el momento:

```
VAR
  Num_etiq : INTEGER;
```

Cuando el parser requiere una nueva etiqueta, ejecuta el algoritmo 6.9, el cual verifica que el número de etiquetas no exceda el límite de la tabla de ensamble.

```
PROCEDURE Etiq_nueva(VAR num : INTEGER);
BEGIN
  verifica_lim(Num_etiq, num_max_etiq);
  Num_etiq := Num_etiq + 1;
  num := Num_etiq
END;
```

Algoritmo 6.9 Creación de una etiqueta.

El análisis de una proposición while resulta ahora muy sencillo (algoritmo 6.10).

```
PROCEDURE Prop_While(Stop: simbolos);
VAR
  Tipo_de_la_expr: Ptro;
  Etiq1, Etiq2 : INTEGER;
BEGIN
  Etiq_nueva(Etiq1);
  Emite2(DEFINE_DIREC2,Etiq1);
  Acepta(WHILE1, inic_expr + [DO1] + inic_de_props + Stop);
  Expresion(Tipo_de_la_expr, [DO1] + inic_de_props + Stop);
  Verifica_tipos(Tipo_de_la_expr, TipoBoolean);
```

```
    Acepta(DO1, inic_de_props + Stop);
    Etiq_nueva(Etiq2);
    Emite2(DO2,Etiq2);
    Pop(1);
    Prop(Stop);
    Emite2(GOTO2,Etiq1);
    Emite2(DEFINE_DIREC2,Etiq2);
  END;
```

Algoritimo 6.10 Generación de código para la prop. while.

El ensamblador guarda la dirección de la instrucción actual en una variable:

```
VAR
  dirección : INTEGER;
```

La dirección se incrementa después de la generación de una instrucción (algoritmo 6.11).

```
PROCEDURE Emite2(Op : cod_op; arg : INTEGER);
BEGIN
  Emite(ORD(op));
  Emite(arg);
  dirección := dirección + 2
END;
```

Algoritmo 6.11 Incrementa la direc. de la instrucción actual.

Las direcciones de las etiquetas se almacenan en una tabla:

```
CONST
   num_max_etiq = 1000;
TYPE
   Tabla_ensamble = ARRAY[1..num_max_etiq ] OF INTEGER;
VAR
   Tabla : Tabla_ensamble;
```

Después de introducir una instrucción DEFINE_DIREC, el ensamblador almacena su dirección en la tabla y lee la siguiente instrucción (algoritmo 6.12).

```
PROCEDURE DEFINE_DIREC(num_etiq : INTEGER);
BEGIN
  Tabla[num_etiq] := dirección;
  Instr_sig
END;
```

Algoritmo 6.12 Almacena en la tabla de ensable la dirección de la instrucción DEFINE_DIREC.

La compilación de las demás proposiciones es muy similar a la de la proposición while.

6.10 CODIGO PARA PROCEDIMIENTOS.

El código del procedimiento Quicksort:

```
PROCEDURE Quicksort(m,n);
VAR
  i,j : INTEGER;
  PROCEDURE Partición;
  BEGIN
    ...
  END;
BEGIN
  SL
END;
```

tiene la siguiente forma:

```
PROCEDURE Quicksort(Long_var, Long_temp, Desp, num_linea)
  Código para partición.
  Código para SL.
FINPROC(Long_param)
```

Cuando el parser alcanza el procedimiento Quicksort, debe generar una instrucción procedure. Pero en este punto, aún no puede calcular los argumentos de la instrucción:

(1) Long_var es la longitud de las variables i y j. Este argumento se conoce sólo después del análisis de la parte de definición de variables:

```
VAR
 i,j : INTEGER;
```

(2) Long_temp es el límite máximo de las localidades temporales para la lista de variables.

(3) Desp es la dirección de la lista de proposiciones relativa a la instrucción procedure. Esta se conoce sólo al principio de SL. Estas referencias hacia adelante son resueltas por el ensamblador. El parser asigna una etiqueta número a cada uno de los argumentos anteriores y genera la instrucción procedure con estas etiquetas:

```
Procedure(Etiq_var, Etiq_temp, Etiq_inic, num_lin);
```

Cuando el parser alcanza un punto en el cual se conoce el valor del argumento, éste genera el valor (y la etiqueta correspondiente) en la forma de una pseudoinstrucción:

```
DEFINE_ARG(Etiq_var,Long_var);
```

Durante la primer fase, el número de nivel y la etiqueta de un procedimiento se almacenan en la parte variante del registro de objeto correspondiente.

En el Capitulo 5 discutimos el análisis de un procedimiento (algoritmo 5.17). El algoritmo 6.13 es la versión final de este procedimiento de análisis.

```
PROCEDURE Def_de_procedimiento(Stop: simbolos);
VAR
  id: INTEGER;
  Proc: Ptro;
  Etiq_proc, Long_param, Etiq_var, Etiq_temp, Etiq_inic : INTEGER;
BEGIN
  Acepta(PROCEDURE1,[IDENT,PARENT_IZQ,PUNTOYCOMA] + inic_de_bloque
         + Stop);
  Acepta_id(id, [PARENT_IZQ, PUNTOYCOMA] + inic_de_bloque + Stop);
  Define(id,procedimiento,Proc);
  Proc^.nivel_proc := Nivel_Block;
  Etiq_nueva(Proc^.etiq_proc);
  BloqueNuevo;
  IF simbolo = PARENT_IZQ THEN
  BEGIN
    Acepta(PARENT_IZQ, inic_param + [PARENT_DER, PUNTOYCOMA] +
           inic_de_bloque+Stop);
    List_param_formal(Proc^.ult_param,Long_param,[PARENT_DER,
                      PUNTOYCOMA]+ inic_de_bloque+Stop);
    Acepta(PARENT_DER, [PUNTOYCOMA] + inic_de_bloque + Stop)
  END
  ELSE { No hay lista de parámetros }
  BEGIN
    Proc^.ult_param := NIL;
    Long_param := 0
  END;
  Etiq_nueva(Etiq_var);
  Etiq_nueva(Etiq_temp);
  Etiq_nueva(Etiq_inic);
  Emite2(DEFINE_DIREC2, Proc^.Etiq_proc);
  Emite5(PROCEDURE2, Etiq_var, Etiq_temp, Etiq_inic, num_lin);
  Acepta(PUNTOYCOMA, [PUNTOYCOMA] + inic_de_bloque + Stop);
  Bloque(Etiq_inic, Etiq_var, Etiq_temp, [PUNTOYCOMA] + Stop);
  Acepta(PUNTOYCOMA, Stop);
  Emite2(FINPROC2, Long_param);
  Final_de_bloque
END;
```

Algoritmo 6.13 Generación de código para procedimientos.

Los argumentos de una instrucción procedure están definidos al final del cuerpo del bloque.

```
PROCEDURE Bloque { Etiq_inic, Etiq_var, Etiq_temp : INTEGER;
                   Stop : simbolos };
VAR
  Long_var : INTEGER;
BEGIN
  ...
  WHILE simbolo = PROCEDURE1 DO
```

```
        Def_de_procedimiento([PROCEDURE1,BEGIN1] + Stop);
    Emite2(DEFINE_DIREC2, Etiq_inic);
    Prop_compuesta(Stop);
    Emite3(DEFINE_ARG2, Etiq_var, Long_var);
    Emite3(DEFINE_ARG2, Etiq_temp, Block[Nivel_Block].num_max_loc);
END;
```

Cuando el ensamblador lee una instrucción LlamadaProc, éste reemplaza la etiqueta del procedimiento por la dirección del código procedure, relativa a la dirección actual (algoritmo 6.14).

```
PROCEDURE Prop_procedure(Stop: simbolos);
VAR
  stop2 : simbolos;
  Proc  : Ptro;
  Long_param : INTEGER;
BEGIN
  Encuentra(argumento, Proc);
  IF Proc^.clase = Proc_estandar THEN
     Prop_ES(Stop)
  ELSE
  BEGIN
    IF Proc^.ult_param <> NIL THEN
    BEGIN
      stop2 := [PARENT_DER] + Stop;
      Acepta(IDENT, [PARENT_IZQ] + inic_expr + Stop2);
      Acepta(PARENT_IZQ,inic_expr + Stop2);
      Lista_de_param_act(Proc^.ult_param, Long_param, Stop2);
      Acepta(PARENT_DER, Stop)
    END
    ELSE { No hay lista de parámetros }
    BEGIN
      Acepta(IDENT,Stop);
      Long_param := 0
    END;
    Emite3(LLAMADA_PROC2,Nivel_Block - Proc^.nivel_proc,
           Proc^.etiq_proc);
    Push(3);
    Pop(Long_param + 3)
  END
END;
```

Algoritmo 6.14 Generación de código para la llamada a un procedimiento.

6.11 OPTIMIZACION DE CODIGO.

El código Mini-Pascal se puede optimizar ampliando las instrucciones estándar de la computadora Mini-Pascal con unas instrucciones extra. Por ejemplo, la proposición de asignación:

```
k := k + 1
```

produce 13 palabras de código:

```
Variable(0,13)
Variable(0,13)
Valor(1)
Constante(1)
Mas
Asigna(1)
```

Este código puede ser reducido a 8 palabras utilizando algunas instrucciones extra:

```
* VarLocal(13)
* ValorLocal(13)
  Constante(1)
  Mas
* Asignación simple
```

Las instrucciones extra están marcadas con un *.

Una instrucción VarLocal reemplaza una instrucción Variable referente al nivel cero. Estableceremos la relación entre la instrucción nueva y la instrucción estándar mediante una regla código parametrizada:

```
VarLocal(Desp) = Variable(0,Desp)
```

a la cual llamaremos regla de optimización.

Cuando la computadora ejecuta una instrucción VarLocal, ésta coloca la dirección de una variable en la pila (algoritmo 6.15).

```
PROCEDURE VarLocal(Desp : INTEGER);
BEGIN
  s := s + 1;
  St[s] := b + Desp;
  p := p + 2
END;
```

Algoritmo 6.15 Instrucción VarLocal.

Una instrucción ValorLocal reemplaza dos instrucciones:

```
Variable(0,Desp) Valor(1)
```

Podemos expresar esto mediante otra regla de optimización:

```
ValorLocal(Desp) = VarLocal(Desp) Valor(1)
```

La ejecución de una instrucción ValorLocal coloca el valor de una variable local simple en la pila.

Las instrucciones correspondientes para variables globales están definidas por las siguientes reglas de optimización:

```
VarGlobal(Desp) = Variable(1,Desp)
ValorGlobal(Desp) = VarGobal(Desp) Valor(1)
```

Una instrucción ValorSimple reemplaza la dirección de una variable simple por su valor.

Una instrucción AsignaciónSimple asigna un valor a una variable simple.

La instrucción extra:

```
LlamadaGlobal(Desp) = LlamadaProc(1,Desp)
```

representa el caso donde un procedimiento Q definido en un bloque llama a otro procedimiento P definido en el mismo bloque.

El uso de reglas sintácticas para describir la optimización de código tiene una interpretación interesante: Las instrucciones extra pueden observarse como unidades sintácticas reconocidas en el código estándar.

La generación de código está definida por un algoritmo de dos pasos:

(1) Durante la primer fase, el parser explora un programa y genera el código que corresponde a la sintaxis del programa mismo.

(2) En la segunda fase, el ensamblador explora el código estándar y reemplaza algunas secuencias de código por código optimizado.

CONCLUSIONES.

En este trabajo se explicaron detalladamente algunas de las técnicas más usuales para la construcción manual de compiladores y se mostró como se pueden programar en un lenguaje de alto nivel.

Es por eso que se considera que las personas que quieran escribir parte de un compilador ó algún otro programa de aplicación, en el cual se utilicen las técnicas expuestas, les será útil consultar esta tesis para que se den cuenta de la estructura de estos programas y puedan aplicar los algoritmos y estructuras de datos apropiadas, o bien para usar parte de las herramientas ya programadas, en la realización de tales proyectos.

En lo expuesto a lo largo de esta tesis, tanto por lo que se refiere a su parte teórica, como a la parte práctica, se hizo énfasis en la construcción de programas para compiladores, sin embargo, la teoría y las técnicas descritas también se utilizan en otras áreas importantes, entre las cuales podemos citar: procesamiento de lenguaje natural, programas de análisis de comandos, procesamiento de textos, traductores fuente-fuente, sistemas operativos y preprocesadores.

Esta tesis puede servir de referencia para estudiantes de la carrera de Ing. en Computación, Lic. en Matemáticas Aplicadas y Computación, Lic. en Informática o para personas con experiencia en el diseño y desarrollo de sistemas de información, ya que en ella encontrarán ideas, algoritmos, técnicas de acceso a información almacenada en tablas y herramientas que les serán de gran ayuda en la realización de programas de aplicación similares a los aquí tratados. También servirá de apoyo a los estudiantes que se inician en el campo de la Computación o Informática a nivel medio superior, debido a que el material expuesto se explica con un lenguaje sencillo y se incluyen bastantes figuras que facilitan la comprensión de los temas desarrollados.

APENDICE A.1 Parser/traductor para el lenguaje definido por la gramática 1.6 (Capitulo 1).

```
PROGRAM Parser_Trad;
TYPE
  cadena = STRING[80];
VAR
  car    : CHAR;                { Carácter delantero }
  linea  : CADENA; indice : INTEGER;

  { Devuelve true si car es un espacio, tab ó nueva línea }
  FUNCTION Es_blanco(car:CHAR): BOOLEAN;
  BEGIN
    Es_blanco := car in[' ',chr(9),chr(13)];
  END;

  { Obtiene un nuevo carácter de la cadena de entrada }
  PROCEDURE Obt_car;
  BEGIN
    indice := indice + 1;
    WHILE (Es_blanco(linea[indice])) do
      indice := indice + 1;     { Ignora los blancos }
    car := linea[indice];
  END;

  PROCEDURE Error(s:cadena);    { Reporta un error }
  BEGIN
    WRITELN; WRITELN(^G,'Error: ',s);
  END;

  { Reporta un error y suspende el proceso }
  PROCEDURE Suspende(s:cadena);
  BEGIN
    Error(s); Halt;
  END;

  { Despliega el token que estaba esperando)
  PROCEDURE Espera(s:cadena);
  BEGIN
    Suspende('Se esperaba ' + s);
  END;

  { Reconoce un carácter alfabético  }
  FUNCTION Es_alfa(c:char): boolean;
  BEGIN
    Es_alfa := UpCase(c) in ['A'..'Z'];
  END;

  FUNCTION Es_digito(c:char):boolean; { Reconoce un dígito }
  BEGIN
    Es_digito := c in ['0'..'9'];
  END;

  { Reconoce un carácter alfanumérico }
  FUNCTION Es_Alfa_num(c:char):boolean;
```

```
BEGIN
  Es_Alfa_num := Es_alfa(c) or Es_digito(c);
END;

FUNCTION Es_op_sum(c:char):boolean; { Reconoce un op. sumador }
BEGIN
  Es_op_sum := c in ['+','-'];
END;

{ Reconoce un op. multiplicador }
FUNCTION Es_op_mul(c:char):boolean;
BEGIN
  Es_op_mul := c in ['*','/'];
END;
              { MANEJO DE ERRORES: Equipara, Obt_ident y Obt_num }
PROCEDURE Equipara(x:char);
BEGIN     { Equipara un carácter especifico de entrada }
  if car <> x then
     Espera('''' + x + '''')
  else
     Obt_car;
END;

FUNCTION Obt_ident:char;     { Obtiene un identificador }
BEGIN
  if not Es_alfa(car) then Espera('Identificador');
  Obt_ident := Upcase(car);
  Obt_car;
END;

FUNCTION Obt_num : INTEGER;   { Obtiene un número }
BEGIN
  if not Es_digito(car) then Espera('Entero');
  Obt_num := Ord(car) - Ord('0');
  Obt_car;
END;

PROCEDURE Escribe(s: cadena); { Escribe una cadena }
BEGIN
  Write(s);
END;

PROCEDURE Escribe_lin(s:cadena);
BEGIN
  Escribe(s);
  Writeln;
END;
       { GENERACION DE CODIGO }

PROCEDURE Carga_cte(n:INTEGER);
BEGIN  { Carga una constante al registro AX }
  Escribe('MOV AX,');
  Writeln(n);
END;
```

```
    { Carga una variable al registro AX }

PROCEDURE Carga_var(Ident:char);
BEGIN
  Escribe('MOV AX,');
  WRITE(ident);
  WRITELN;
END;

PROCEDURE Inserta_en_pila; { Coloca el registro AX en la pila }
BEGIN
  Escribe_lin('PUSH AX    ; Coloca el reg. AX en la pila');
END;

    { Suma el contenido del tope de la pila al reg. AX }

PROCEDURE Extrae_suma;
BEGIN
  Escribe_lin('MOV BX,AX  ; Segundo sumando en BX');
  Escribe_lin('POP AX     ; Primer sumando en AX');
  Escribe_lin('ADD AX,BX  ; AX <-- AX + BX       ');
END;

    { Resta el registro AX del tope del STACK }

PROCEDURE Extrae_resta;
BEGIN
  Escribe_lin('MOV BX,AX  ; Sustraendo en BX');
  Escribe_lin('POP AX     ; Minuendo en AX');
  Escribe_lin('SUB AX,BX  ; AX <-- AX-BX');
END;

{ Multiplica el elemetno superior de la pila por el contenido
  del registro AX }

PROCEDURE Extrae_multiplica;
BEGIN
  Escribe_lin('MOV CX,AX  ; Primer factor en CX, parte baja');
  Escribe_lin('POP AX     ; Segundo factor en AX, parte baja');
  Escribe_lin('MUL CL     ; AX <-- AL * CL');
END;

{ Extrae el elemento superior de la pila y lo divide entre el
  contenido del registro AX }

PROCEDURE Extrae_divide;
BEGIN
  Escribe_lin('MOV CX,AX  ; Divisor en la parte baja de CX');
  Escribe_lin('POP AX     ; Dividendo en la parte baja de AX');
  Escribe_lin('DIV CL     ; Divide AL entre CL ');
  Escribe_lin('             AH = cociente, AL = residuo ');
END;
```

```
{ Almacena el contenido del registro AX en una variable }

PROCEDURE Almacena(ident: char);
BEGIN
  Escribe('MOV ');
  WRITE(ident);
  Escribe(',AX   ; Se almacena el resultado en la variable ');
  WRITE(ident);
END;

                        { ANALISIS SINTACTICO }

PROCEDURE Expr; Forward; { Analiza y traduce una expresión }

PROCEDURE Factor; { Analiza y traduce un factor matemático }
BEGIN
  if car = '(' then
  BEGIN
    Equipara('(');
    Expr;
    Equipara(')');
  END
  else
    if Es_alfa(car) then Carga_var(Obt_ident)
    else Carga_cte(Obt_num);
END;

PROCEDURE Multiplica;    { Reconoce y traduce una mult. }
BEGIN
  Equipara('*');
  Factor;
  Extrae_multiplica;
END;

PROCEDURE Divide;        { Reconoce y traduce una div. }
BEGIN
  Equipara('/');
  Factor;
  Extrae_divide;
END;

PROCEDURE Termino;
BEGIN
  Factor;
  while Es_op_mul(car) do
  BEGIN
    Inserta_en_pila;
    case car of
         '*' : Multiplica;
         '/' : Divide;
    END;
  END;
END;
```

```
  PROCEDURE Suma;          { Reconoce y traduce una suma }
  BEGIN
    Equipara('+');
    Termino;
    Extrae_suma;
  END;

  PROCEDURE Resta;         { Reconoce y traduce una resta }
  BEGIN
    Equipara('-');
    Termino;
    Extrae_resta;
  END;

  PROCEDURE Expr;          { Analiza y traduce una expresión }
  BEGIN
    Termino;
    while Es_op_sum(car) do
    BEGIN
      Inserta_en_pila;
      case car of
           '+' : Suma;
           '-' : Resta;
      END;
    END;
  END;
       { Analiza y traduce una proposición de asignación }

  PROCEDURE Asignacion;
  var
     ident : char;
  BEGIN
    ident := Obt_ident;
    Equipara('=');
    Expr;
    Almacena(ident);
  END;

  PROCEDURE Inicializa;
  BEGIN
    CLRSCR;
    for indice := 1 to 80 do linea[indice] := ' ';
    indice := 0;
    WRITE('Entrada: ');
    READLN(linea);
    GOTOXY(1,3); WRITELN('Salida:');
    GOTOXY(1,5);
    Obt_car;
  END;

BEGIN
   Inicializa;
   Asignacion;
END.
```

APENDICE A.2. Analizador léxico para programas escritos en Mini-Pascal (Capitulo 2).

```
PROGRAM Analizador_lexico;
USES CRT;
CONST
  num_cves = 631;   { Núm. de claves en la tabla hash }
  num_car  = 5000;  { Núm. de caracteres en la tabla de lexemas}
  long_id  = 30;    { Long. max. para un identificador }

                    { Palabras Reservadas }
TYPE
  t_simb = (AND1, ARRAY1, BEGIN1, CONST1,  DIV1, DO1, ELSE1, END1,
     IF1,MOD1,NOT1,OF1, OR1, PROCEDURE1, PROGRAM1, RECORD1,THEN1,
     TO1, TYPE1, VAR1, WHILE1,

                    { Identificadores estándar }

     INTEGER1, BOOLEAN1, FALSE1, TRUE1, READ1, WRITE1,

                    { Operadores / Delimitadores }

     ASIGNACION,  MAS,  MENOS,  POR, DIVISION, IGUAL, DIFERENTE,
     MENOR_QUE, MENOR_O_IGUAL, MAYOR, MAYOR_O_IGUAL, NUEVALINEA,

                    { Otros }

     IDENT,  CTE_ENT,  CORCHETE_IZQ,  PARENT_IZQ,  PARENT_DER,
     CORCHETE_DER, COMENT, COMA, PUNTO, PUNTOYCOMA, DOSPUNTOS,
     PUNTOPUNTO,OTRO);

  t_error = (coment_inc, cte_inc,rango_inc, id_inc);
  cad = STRING[30];
  arr_lexemas = ARRAY[1..num_car] OF CHAR;
  Ptro_a_palabra  = ^Reg_palabra;

  Reg_palabra = RECORD
                  palabra_sig : Ptro_a_palabra;
                  es_id       : BOOLEAN;
                  indice,
                  long,
                  ult_car     : INTEGER;
                END;

  TablaHash = ARRAY[1..num_cves] OF Ptro_a_palabra;

VAR
  lexemas       : arr_lexemas;
  hash          : TablaHash;
  long,
  apun_car,     { índice del carácter actual }
  num_lin,      { núm. de líneas del programa de entrada }
  caracteres,   { núm. caracteres en la tabla de lexemas }
  num_ids,      { núm. de identificadores    }
```

```
num_errs,      { núm. de errores detectados durante el análisis}
cont_reng      : INTEGER;
arch_ent, temp1,
temp2          : TEXT;
car            : CHAR;
texto, descr_token : cad;

nom_arch_ent   : STRING[15];
linea          : ARRAY[1..80] OF CHAR;
fin_linea,
fin_arch,
lin_correcta, todo_bien, ha_habido_errores, arch_ok : BOOLEAN;
{$I impr.pas} {$I error.pas} {$I inicio.pas}

FUNCTION cve_hash(texto : cad; long:INTEGER) : INTEGER;
CONST
  w = 32641; {32768-127 = 32641 para prevenir error de overflow}
  n = num_cves;
VAR
  sum, i : INTEGER;
BEGIN
  sum := 0; i:= 1;
  WHILE i <= long DO
  BEGIN
    sum := (sum + ORD(texto[i])) MOD w;
    i := i + 1
  END;
  cve_hash := sum mod n+1;
END;

PROCEDURE verifica_lim(long,max : INTEGER);
BEGIN
  IF long >= max THEN
  BEGIN
    WRITELN('Programa demasiado grande '); HALT;
  END;
END;

PROCEDURE inserta(es_id : BOOLEAN; texto : cad; long, indice,
                  num_cve : INTEGER);
VAR
  Ptro : Ptro_a_palabra; m,n  : INTEGER;
BEGIN    { Inserta la palabra en la tabla de lexemas }
  caracteres := caracteres + long;
  verifica_lim(caracteres,num_car);
  m := long; n := caracteres - m;
  WHILE m > 0 DO
  BEGIN
    lexemas[m+n] := texto[m];
    m := m - 1
  END;

    { Inserta la palabra en la lista de palabras }
```

```
  NEW(Ptro);
  Ptro^.palabra_sig := hash[num_cve];
  Ptro^.es_id       := es_id;
  Ptro^.indice      := indice;
  Ptro^.long        := long;
  Ptro^.ult_car     := caracteres;
  hash[num_cve]     := Ptro
END;

FUNCTION encont(texto : cad; long : INTEGER;
                Ptro  : Ptro_a_palabra):BOOLEAN;
VAR
  misma : BOOLEAN;
  m,n   : INTEGER;
BEGIN
  IF Ptro^.long <> long THEN misma := FALSE
  ELSE
    BEGIN
      misma := TRUE;
      m := long;
      n := ptro^.ult_car - m;
      WHILE (misma = TRUE) AND ( m > 0) DO
      BEGIN
        misma := texto[m] = lexemas[m+n];
        m := m - 1
      END;
    END;
  encont := misma
END;

PROCEDURE Instala(es_id : BOOLEAN; texto : cad; long,
                  indice : INTEGER);
BEGIN
  inserta(es_id,texto, long, indice, cve_hash(texto, long))
END;

PROCEDURE busca(texto:cad; long : INTEGER; VAR es_id : BOOLEAN;
                VAR indice : INTEGER);
VAR
  num_cve   : INTEGER;
  Ptro      : Ptro_a_palabra;
  realizado : BOOLEAN;
BEGIN
  num_cve   := cve_hash(texto,long);
  Ptro      := hash[num_cve];
  realizado := FALSE;
  WHILE NOT realizado DO
  IF Ptro = NIL THEN
  BEGIN              { La lista no contiene la palabra }
    es_id   := TRUE;
    num_ids := num_ids + 1;
    indice  := num_ids;
    inserta(TRUE, texto, long, indice, num_cve);
    realizado := TRUE
```

```
  END
  ELSE
    IF encont(texto,long,Ptro) THEN
    BEGIN            { La lista contiene la palabra }
      es_id     := Ptro^.es_id;
      indice    := Ptro^.indice;
      realizado := TRUE
    END
    ELSE             { La lista ya contiene otra palabra }
      Ptro := Ptro^.palabra_sig
END;

PROCEDURE Nueva_lin(num : INTEGER);
BEGIN
  num_lin := num;
  lin_correcta := TRUE
END;

PROCEDURE Empieza_lin(num : INTEGER );
BEGIN
  Nueva_lin(num);
END;

PROCEDURE Graba_linea;
BEGIN
  WRITE(temp2,num_lin,'   ',linea);
  WRITELN(temp2);
END;

PROCEDURE Fin_de_linea;
BEGIN
  IF todo_bien THEN WRITELN(temp1);
  Empieza_lin(num_lin + 1)
END;

PROCEDURE limpia_linea;
VAR
  i : INTEGER;
BEGIN
  for i := 1 to 80 DO
      linea[i] := ' ';
END;

{ Procedimiento para efectuar la verificación de fin de archivo
  y regresar el sig. carácter del archivo de entrada }

PROCEDURE lee_linea;
BEGIN
  IF NOT EOF(arch_ent) THEN  { lee_linea }
  BEGIN
    Fin_de_linea;
    long := 0;
    WHILE NOT eoln(arch_ent) DO
    BEGIN
```

```
      long := long + 1;
      read(arch_ent,linea[long]);
    END;
    Graba_linea;
    READLN(arch_ent);
    fin_linea := (apun_car > long)
  END;
END;

PROCEDURE Obt_car;
BEGIN
  IF fin_linea THEN
    IF EOF(arch_ent) THEN fin_arch := TRUE
    ELSE lee_linea
  ELSE
    BEGIN
      apun_car := apun_car + 1;
      IF apun_car > long THEN fin_linea := TRUE
    END;
  car := linea[apun_car];
  IF car IN ['a'..'z'] THEN car := UPCASE(car);
END;

PROCEDURE comentario;
BEGIN
  Obt_car;
  WHILE NOT ( car IN ['}'] )  AND (fin_linea = FALSE) DO
    IF car = '{' THEN comentario
    ELSE Obt_car;
  IF car = '}' THEN Obt_car
  ELSE error(coment_inc)
END;

{Proc. para clasificar un token según su primer carácter}

PROCEDURE Explora_simbolo;
VAR
  es_id : BOOLEAN;
  long, indice, valor, digito : INTEGER;
BEGIN
  texto := '                                        ';
  WHILE car = ' ' DO Obt_car;
  CASE car OF
   'A'..'Z': BEGIN
               { Análisis de un id. o una palabra reservada }
               long := 0;
               WHILE (car IN ['0'..'9','A'..'Z','_']) DO
               BEGIN
                 IF long < long_id THEN
                 BEGIN
                   long := long + 1;
                   texto[long] := car;
                 END
                 ELSE
```

```
            IF lin_correcta = TRUE THEN Error(id_inc);
            Obt_car;
          END;
          busca(texto,long,es_id,indice);
          IF es_id THEN
          BEGIN
            IF car = '.' THEN
            BEGIN
              descr_token := 'Variable registro';
              impr(texto,descr_token,IDENT);
            END
            ELSE
              BEGIN
                descr_token := 'Identificador';
                impr(texto,descr_token,IDENT);
              END
          END
          ELSE
            BEGIN
              descr_token := 'Palabra reservada';
              impr(texto,descr_token,t_simb(indice))
            END
        END;

'0'..'9' : BEGIN    { Análisis de una constante entera }
           valor := 0;
           WHILE car IN ['0'..'9'] DO
           BEGIN
              digito := ORD(car) - ORD('0');
              IF valor <= (MAXINT - digito) DIV 10
              THEN
                 BEGIN
                    valor := 10*valor + digito;
                    Obt_car
                 END
              ELSE
                 BEGIN
                    error(cte_inc);
                    WHILE car IN ['0'..'9'] DO
                          Obt_car
                 END
           END;
           descr_token := 'Constante entera';
           str(valor,texto);
           impr(texto,descr_token,CTE_ENT)
         END;

              { Operadores/delimitadores }
'+' : BEGIN
       descr_token := 'Mas';
       texto := car;
       impr(texto,descr_token,MAS);
       Obt_car
     END;
```

```
'-' : BEGIN
        descr_token := 'Menos';
        texto := car;
        impr(texto,descr_token,MENOS);
        Obt_car
      END;

'*' : BEGIN
        descr_token := 'Por';
        texto := car;
        impr(texto,descr_token,POR);
        Obt_car
      END;

'/' : BEGIN
        texto := car;
        descr_token := 'División';
        impr(texto,descr_token,DIVISION);
        Obt_car;
      END;

'<' : BEGIN
         texto := car;
         Obt_car;
         IF car = '=' THEN
         BEGIN
            descr_token := 'Menor o igual';
            texto := texto + car;
            impr(texto,descr_token,MENOR_O_IGUAL);
            Obt_car
         END
         ELSE
            IF car = '>' THEN
            BEGIN
               descr_token := 'Diferente a';
               texto := texto + car;
               impr(texto,descr_token,DIFERENTE);
               Obt_car
            END
            ELSE
              BEGIN
                 descr_token := 'Menor que';
                 impr(texto,descr_token,MENOR_QUE)
              END;
      END;

'=' : BEGIN
        descr_token := 'Igual a';
        texto := car;
        impr(texto,descr_token,IGUAL);
        Obt_car;
      END;
```

```
')' : BEGIN
        texto := car;
        Obt_car;
        IF car = '=' THEN
        BEGIN
          descr_token := 'Mayor o igual';
          texto := texto + car;
          impr(texto,descr_token,MAYOR_O_IGUAL);
          Obt_car;
        END
        ELSE
          BEGIN
            descr_token := 'Mayor';
            impr(texto,descr_token,MAYOR);
          END;
      END;

':' : BEGIN
        texto := car;
        Obt_car;
        IF car = '=' THEN
        BEGIN
          descr_token := 'Asignación';
          texto := texto + car;
          impr(texto,descr_token,ASIGNACION);
          Obt_car
        END
        ELSE
          BEGIN
            descr_token := 'Dos puntos';
            impr(texto,descr_token,DOSPUNTOS);
          END;
      END;

'(' : BEGIN
        texto := car;
        descr_token := 'Parentesis izquierdo';
        impr(texto,descr_token,PARENT_IZQ);
        Obt_car
      END;

')' : BEGIN
        texto := car;
        descr_token := 'Parentesis derecho';
        impr(texto,descr_token,PARENT_DER);
        Obt_car
      END;

'[' : BEGIN
        texto := car;
        descr_token := 'Corchete izquierdo';
        impr(texto,descr_token,CORCHETE_IZQ);
        Obt_car
      END;
```

```
']' : BEGIN
        texto := car;
        descr_token := 'Corchete derecho';
        impr(texto,descr_token,CORCHETE_DER);
        Obt_car
      END;

',' : BEGIN
        texto := car;
        descr_token := 'Coma';
        impr(texto,descr_token,COMA);
        Obt_car
      END;

'.' : BEGIN
       texto := car;
       Obt_car;
       IF car = '.' THEN
       BEGIN
         descr_token := 'Punto punto';
         texto := texto+car;
         impr(texto,descr_token,PUNTOPUNTO);
         Obt_car
       END
       ELSE
         IF (CAR IN ['A'..'Z']) and (not fin_arch) THEN
         BEGIN
           descr_token  := 'Punto';
           impr(texto,descr_token,PUNTO);
         END
           ELSE
             BEGIN
               descr_token  := 'Punto';
               impr(texto,descr_token,PUNTO);
               fin_arch :=  TRUE;
             END;
     END;

'{' : comentario;

';' : BEGIN
       texto := car;
       descr_token := 'Punto y coma';
       impr(texto,descr_token,PUNTOYCOMA);
       Obt_car
    END;
            { Caracteres ilegales }
    ELSE
      BEGIN
        texto := car;
        descr_token := 'Carácter inválido ';
        impr(texto,descr_token,OTRO);
        Obt_car;
```

```
        END;
  END;
END;

PROCEDURE inicializa_tabla;
VAR
  i : INTEGER;
BEGIN
  i := 1;    { Inicialización de la tabla Hash }
  WHILE i <= num_cves DO
  BEGIN
    hash[i] := NIL ;
    i := i + 1
  END;
  caracteres := 0;

  { Inserta la palabras reservadas }

  Instala(FALSE, 'AND',      3,ORD(AND1));
  Instala(FALSE, 'ARRAY',    5,ORD(ARRAY1));
  Instala(FALSE, 'BEGIN',    5,ORD(BEGIN1));
  Instala(FALSE, 'CONST',    5,ORD(CONST1));
  Instala(FALSE, 'DIV',      3,ORD(DIV1));
  Instala(FALSE, 'DO',       2,ORD(DO1));
  Instala(FALSE, 'ELSE',     4,ORD(ELSE1));
  Instala(FALSE, 'END',      3,ORD(END1));
  Instala(FALSE, 'IF',       2,ORD(IF1));
  Instala(FALSE, 'MOD',      3,ORD(MOD1));
  Instala(FALSE, 'NOT',      3,ORD(NOT1));
  Instala(FALSE, 'OF',       2,ORD(OF1));
  Instala(FALSE, 'OR',       2,ORD(OR1));
  Instala(FALSE, 'PROCEDURE',9,ORD(PROCEDURE1));
  Instala(FALSE, 'PROGRAM',  7,ORD(PROGRAM1));
  Instala(FALSE, 'RECORD',   6,ORD(RECORD1));
  Instala(FALSE, 'THEN',     4,ORD(THEN1));
  Instala(FALSE, 'TO',       2,ORD(TO1));
  Instala(FALSE, 'TYPE',     4,ORD(TYPE1));
  Instala(FALSE, 'VAR',      3,ORD(VAR1));
  Instala(FALSE, 'WHILE',    5,ORD(WHILE1));

    { Inserta los identificadores estándar }

  Instala(TRUE, 'INTEGER',  7,ORD(INTEGER1));
  Instala(TRUE, 'BOOLEAN',  7,ORD(BOOLEAN1));
  Instala(TRUE, 'FALSE',    5,ORD(FALSE1));
  Instala(TRUE, 'TRUE',     4,ORD(TRUE1));
  Instala(TRUE, 'READ',     4,ORD(READ1));
  Instala(TRUE, 'WRITE',    5,ORD(WRITE1));
END;

PROCEDURE proceso;
BEGIN
  WHILE NOT fin_arch DO
  BEGIN
```

```
      limpia_linea;
      apun_car := 0;
      lee_linea;
      Obt_car;
      WHILE NOT fin_linea DO
           Explora_simbolo;
    END;
  END;

  PROCEDURE fin;
  BEGIN
    WRITELN(temp2);
    IF num_errs <> 0 THEN
       WRITELN(temp2,'     ',num_errs,' Error(es) encontrado(s)')
    ELSE
       WRITELN(temp2,'  Análisis lexico terminado sin errores ');
    WRITELN(temp2);
    WRITELN(temp2,'  Número de identificadores usados: ',num_ids);
    CLOSE(temp1);
    CLOSE(temp2);
  END;

BEGIN               { Programa principal }
  inicio;
  inicializa_tabla;
  proceso;
  fin;
END.
```

APENDICE A.3. Analizador sintáctico para programas escritos en Mini-Pascal (Capítulos 3, 4 y 5).

```
PROGRAM Analizador_sintactico;
USES CRT;
CONST
  num_cves = 631;  { Núm. de claves en la tabla hash     }
  num_car  = 5000; { Núm. de caracteres en la tabla hash }
  long_id  = 30;   { Long. máx. para un idwentificador   }
  no_hay_id = 0;   { Valor para un id. de una constante incorrecta}
  num_nivel = 10;  { Núm. máx. de bloques                }

                   { Palabras Reservadas                 }
TYPE
  t_simb = (AND1, ARRAY1, BEGIN1, CONST1, DIV1, DO1, ELSE1, END1,
     IF1, MOD1, NOT1, OF1, OR1, PROCEDURE1, PROGRAM1, RECORD1,
     THEN1, TO1, TYPE1, VAR1, WHILE1,

                   { Identificadores estándar }

     INTEGER1, BOOLEAN1, FALSE1, TRUE1, READ1, WRITE1,

                   { Operadores / Delimitadores }

     ASIGNACION, MAS, MENOS, POR, DIVISION, IGUAL, DIFERENTE,
     MENOR_QUE, MENOR_O_IGUAL, MAYOR, MAYOR_O_IGUAL, NUEVALINEA,

                   {            Otros           }

     APOSTROFO,IDENT,CTE_ENT,CORCHETE_IZQ, PARENT_IZQ,PARENT_DER,
     CORCHETE_DER, COMENT, COMA, PUNTO, PUNTOYCOMA, DOSPUNTOS,
     PUNTOPUNTO,OTRO);

  cad = STRING[30];
  arr_lexemas = ARRAY[1..num_car] OF CHAR;
  Ptro_a_palabra  = ^Reg_palabra;
  Reg_palabra = RECORD
                  palabra_sig : Ptro_a_palabra;
                  es_id       : BOOLEAN;
                  indice,
                  long,
                  ult_car     : INTEGER;
                END;

  TablaHash = ARRAY[1..num_cves] OF Ptro_a_palabra;

  t_error = (dup,coment_inc,clase_id_inc,cte_inc,rango_inc,
             error_sintactico,tipo_inc,tipo_id_inc,
             id_indef, id_inc, desconocido);

  simbolos = SET OF t_simb;
  clas = (Constantex, TipoEstandar, TipoArray, TipoRecord, Campo,
          Variable, Param_val,Param_var, Procedimiento,
          Proc_estandar, Indefinido);
```

```
  clases = SET OF clas;

  Ptro = ^Reg_obj;

  Reg_obj = RECORD
              id        : INTEGER;
              ant       : Ptro;
              CASE clase : clas OF
                 constantex : (valor_de_la_cte : INTEGER;
                               tipo_de_la_cte : Ptro);
                 TipoArray  : (Limite_inf, Limite_sup : INTEGER;
                               TipoIndice, TipoElemento : Ptro);
                 TipoRecord : (ult_campo  : Ptro);
                 Campo      : (Tipo_campo : Ptro);
                 Variable,Param_val,Param_var:(tipo_de_var:Ptro);
                 Procedimiento : (ult_param : Ptro);
            END;

  Reg_del_block = RECORD
                    ult_objeto : Ptro
                  END;

  Tab_de_blocks = ARRAY[0..num_nivel] OF Reg_del_block;

VAR
  simbolo : t_simb;
  simb_sumadores, inic_de_bloque, inic_ctes,inic_expr, inic_param,
  inic_de_factores, simb_multiplicadores, simb_relacionales,
  simb_selectores, inic_expr_simple, inic_de_props, signos,
  inic_term:simbolos;

  Tipos, variables, Procedimientos : clases;
  TipoUniversal, TipoEntero, TipoBoolean, TipoChar : Ptro;
  block : Tab_de_blocks;
  nivel_block, argumento : INTEGER;
  lexemas      : arr_lexemas;
  hash         : TablaHash;
  long, apun_car,
  num_lin,     { Núm. de líneas del programa de entrada }
  caracteres,
  num_ids,     { Núm. de identificadores del prog. fuente       }
  num_errs,    { Núm. de errores detectados durante el análisis }
  cont_reng    : INTEGER;
  arch_ent,
  temp1, temp2 : TEXT;
  car          : CHAR;
  texto        : cad;
  nom_arch_ent : STRING[15];
  linea        : ARRAY[1..80] OF CHAR;
  fin_linea, fin_arch, lin_correcta, todo_bien, ha_habido_errores,
  arch_ok : BOOLEAN;  {$I LIMPIA_LINEA.PAS} {$I VERIFICA_LIM.PAS}
  {$I EXPLORA_SIMBOLO.PAS} {$I ERROR.PAS}  {$I TERMINA.PAS}
  {$I INICIALIZA.PAS} {$I DEFINE.PAS} {$I CONSTANTE.PAS}
```

```
        {   A N A L I S I S   D E   A L C A N C E   }

PROCEDURE Busca_id(id, num_nivel : INTEGER; VAR encont : boolean;
                   VAR objeto : Ptro);
VAR
  mas : BOOLEAN;
BEGIN { Se determina si un id. ya ha sido definido en un bloque }
  mas := TRUE;
  objeto := Block[num_nivel].ult_objeto;
  WHILE mas DO
    IF objeto = nil THEN
    BEGIN
      mas := false; encont := false;
    END
    ELSE
      IF objeto^.id = id THEN
      BEGIN
        mas := false; encont := TRUE
      END
      ELSE
        objeto := objeto^.ant
END;

PROCEDURE Define(id:INTEGER; clase:clas; VAR objeto : Ptro);
VAR
  encont : boolean; otro : Ptro;
BEGIN
  IF id <> no_hay_id THEN
  BEGIN
    Busca_id(id,nivel_block, encont,otro);
    IF encont THEN error(dup)
  END;
  NEW(objeto);  objeto^.id := id; objeto^.clase := clase;
  objeto^.ant := Block[nivel_block].ult_objeto;
  Block[nivel_block].ult_objeto := objeto;
END;

PROCEDURE Encuentra(id : INTEGER; VAR objeto : Ptro);
VAR
  mas, encont : boolean; num_nivel : INTEGER;
BEGIN
  mas := true; num_nivel := nivel_block;
  WHILE mas DO
  BEGIN
    busca_id(id, num_nivel, encont, objeto);
    IF encont or (num_nivel = 0) THEN mas := false
    ELSE num_nivel := num_nivel -1;
  END;
  IF not encont THEN
  BEGIN
    Error(id_indef);
    Define(id,Indefinido,objeto)
  END
END;
```

```
PROCEDURE BloqueNuevo;
BEGIN
  Verifica_lim(nivel_block,Num_nivel);
  nivel_block := nivel_block + 1;
  Block[nivel_block].ult_objeto := nil
END;

PROCEDURE Final_de_bloque;
BEGIN
  nivel_block := nivel_block - 1
END;

PROCEDURE Block_estandar;
VAR
  Constx, Proc : Ptro;
BEGIN
  nivel_block := - 1;
  BloqueNuevo;
  Define(no_hay_id, TipoEstandar, TipoUniversal);
  Define(ord(INTEGER1), TipoEstandar, TipoEntero);
  Define(ord(BOOLEAN1), TipoEstandar, TipoBoolean);
  Define(ord(FALSE1), Constantex, Constx);
  Constx^.valor_de_la_cte := ord(false);
  Constx^.tipo_de_la_cte := TipoBoolean;
  Define(ord(TRUE1), Constantex, Constx);
  Constx^.valor_de_la_cte := ord(true);
  Constx^.tipo_de_la_cte := TipoBoolean;
  Define(ord(READ1),Proc_estandar, Proc);
  Define(ord(WRITE1), Proc_estandar, Proc);
END;
          {    A N A L I S I S     D E     T I P O S     }

PROCEDURE Verifica_tipos(VAR tipo1 : Ptro; tipo2: Ptro);
BEGIN
  IF tipo1 <> tipo2 THEN
  BEGIN
    IF (tipo1 <> TipoUniversal) and (tipo2 <> TipoUniversal) THEN
       Error(tipo_id_inc);
    tipo1 := TipoUniversal
  END
END;

PROCEDURE Error_de_tipo(VAR Tipox: Ptro);
BEGIN
  IF Tipox <> TipoUniversal THEN
  BEGIN
    Error(tipo_id_inc); Tipox := TipoUniversal
  END
END;

PROCEDURE Error_de_clase(Objeto: Ptro);
BEGIN
  IF objeto^.clase <> Indefinido THEN Error(clase_id_inc)
END;
```

```
{    A N A L I S I S    S I N T A C T I C O    }

PROCEDURE Error_de_sintaxis(Stop: simbolos);
BEGIN
  Error(error_sintactico);
  WHILE not (simbolo in Stop) DO Explora_simbolo;
END;

PROCEDURE Verifica_sintaxis(Stop: simbolos);
BEGIN
  IF not (simbolo in Stop + [COMENT]) THEN
     Error_de_sintaxis(Stop)
END;

PROCEDURE Acepta(simbolo_esperado: t_simb; Stop: simbolos);
BEGIN
  IF simbolo = simbolo_esperado THEN
     Explora_simbolo
  ELSE
     Error_de_sintaxis(Stop);
  Verifica_sintaxis(Stop);
END;

PROCEDURE Acepta_id(VAR id: INTEGER; Stop : simbolos);
BEGIN
  IF simbolo = IDENT THEN
  BEGIN
    id := argumento; Explora_simbolo
  END
  ELSE
    BEGIN
      id := no_hay_id;
      Error_de_sintaxis(Stop)
    END;
  Verifica_sintaxis(Stop)
END;

PROCEDURE Id_de_tipo(VAR Tipox: Ptro; Stop: simbolos);
VAR
  Objeto: Ptro;
BEGIN
  IF simbolo = IDENT THEN
  BEGIN
    Encuentra(argumento, Objeto);
    IF objeto^.clase IN tipos THEN Tipox := objeto
    ELSE
       BEGIN
         Error_de_clase(objeto); Tipox := TipoUniversal
       END
  END
  ELSE
    Tipox := TipoUniversal;
  Acepta(IDENT, Stop)
END;
```

```
PROCEDURE Definicion_de_cte(Stop: simbolos);
VAR
  id, valor: INTEGER; Constx, Tipox: Ptro;
BEGIN
  Acepta_id(id, [IGUAL, PUNTOYCOMA] + inic_ctes + Stop);
  Acepta(IGUAL, inic_ctes + [PUNTOYCOMA] + Stop);
  Constante(valor, Tipox, [PUNTOYCOMA] + Stop);
  Define(id, Constantex, Constx);
  constx^.valor_de_la_cte := valor;
  constx^.tipo_de_la_cte := Tipox;
  Acepta(PUNTOYCOMA, Stop)
END;

PROCEDURE Parte_de_def_ctes(Stop: simbolos);
VAR
  Stop2: simbolos;
BEGIN
  Stop2 := [IDENT] + Stop;
  Acepta(CONST1, Stop2);
  Definicion_de_cte(Stop2);
  WHILE simbolo = IDENT DO
        Definicion_de_cte(Stop2)
END;

PROCEDURE TipoArreglo(id: INTEGER; Stop: simbolos);
VAR
  Tipo, tipo_del_lim_inf, tipo_del_lim_sup, TipoElemento : Ptro;
  Limite_sup, Limite_inf : INTEGER;
BEGIN
  Acepta(ARRAY1, [CORCHETE_IZQ,CORCHETE_DER, Of1, IDENT]
        + inic_ctes + Stop);
  Acepta(CORCHETE_IZQ,[CORCHETE_DER,OF1,IDENT]+inic_ctes + Stop);
  Constante(Limite_inf,tipo_del_lim_inf,[PUNTOPUNTO,CORCHETE_DER,
          OF1,IDENT] + inic_ctes + Stop);
  Acepta(PUNTOPUNTO,[CORCHETE_DER,OF1,IDENT]+inic_ctes + Stop);
  Constante(Limite_sup,tipo_del_lim_sup,[CORCHETE_DER, OF1,
          IDENT] + Stop);
  Verifica_tipos(tipo_del_lim_inf,tipo_del_lim_sup);
  IF Limite_inf > Limite_sup THEN
  BEGIN
    Error(rango_inc);
    Limite_inf := Limite_sup
  END;
  Acepta(CORCHETE_DER, [OF1, IDENT] + Stop);
  Acepta(OF1, [IDENT] + Stop);
  Id_de_tipo(TipoElemento, Stop);
  Define(id ,TipoArray , Tipo);
  Tipo^.Limite_inf := Limite_inf;
  Tipo^.Limite_sup := Limite_sup;
  Tipo^.TipoIndice  := tipo_del_lim_inf;
  Tipo^.TipoElemento := TipoElemento
END;
```

```
PROCEDURE Seccion_de_regs(VAR ult_campo,Tipox:Ptro;Stop:simbolos);
VAR
  id : INTEGER; campox: Ptro;
BEGIN
  Acepta_id(id, [COMA, DOSPUNTOS] + Stop);
  Define(id, campo, campox);
  IF simbolo = COMA THEN
  BEGIN
    Acepta(COMA,[IDENT] + Stop);
    Seccion_de_regs(ult_campo, Tipox, Stop)
  END
  ELSE
    BEGIN
      Acepta(DOSPUNTOS, [IDENT] + Stop);
      Id_de_tipo(Tipox, Stop);
      ult_campo := campox
    END;
  campox^.Tipo_campo := Tipox
END;

PROCEDURE lista_de_campos(VAR ult_campo: Ptro; Stop : simbolos);
VAR
  Stop2: simbolos; Tipox: Ptro;
BEGIN
  Stop2 := [PUNTOYCOMA] + Stop;
  Seccion_de_regs(ult_campo, Tipox, Stop2);
  WHILE simbolo = PUNTOYCOMA DO
  BEGIN
    Acepta(PUNTOYCOMA, [IDENT] + Stop2);
    Seccion_de_regs(ult_campo,Tipox, Stop2);
  END;
END;

PROCEDURE TipoRegistro(id: INTEGER;  Stop: simbolos);
VAR
  Tipo, ult_campo: Ptro;
BEGIN
  BloqueNuevo;
  Acepta(RECORD1, [IDENT, END1] + Stop);
  lista_de_campos(ult_campo,[END1] + Stop);
  Acepta(END1, Stop);
  Final_de_bloque;
  Define(id, TipoRecord, Tipo);
  Tipo^.ult_campo := ult_campo
END;

PROCEDURE Def_de_tipo(Stop: simbolos);
VAR
  Stop2: simbolos; id: INTEGER; Objeto: Ptro;
BEGIN
  Stop2 := [PUNTOYCOMA] + Stop;
  Acepta_id(id, [IGUAL, ARRAY1,RECORD1] + stop2);
  Acepta(IGUAL,[ARRAY1, RECORD1] + Stop2);
  IF simbolo = ARRAY1 THEN TipoArreglo(id, Stop2)
```

```
  ELSE
     IF simbolo = RECORD1 THEN TipoRegistro(id, Stop2)
     ELSE
       BEGIN
         Define(id, Indefinido, Objeto);
         stop2 := stop2 - [IDENT];
         Error_de_sintaxis(Stop2);
       END;
  Acepta(PUNTOYCOMA, Stop)
END;

PROCEDURE Parte_def_tipos(Stop: simbolos);
VAR
  Stop2: simbolos;
BEGIN
  Stop2 := [IDENT] + Stop;
  Acepta(TYPE1, Stop2);
  Def_de_tipo(Stop2);
  WHILE simbolo = IDENT DO
    Def_de_tipo(Stop2)
END;

PROCEDURE lista_variables(clase : clas; VAR ult_var,Tipox : Ptro;
                          Stop: simbolos);
VAR
  id   : INTEGER; Varx : Ptro;
BEGIN
  Acepta_id(id, [COMA, DOSPUNTOS] + Stop);
  Define(id,clase,Varx);
  IF simbolo = COMA THEN
  BEGIN
    Acepta(COMA, [IDENT] + Stop);
    lista_variables(clase, ult_var, Tipox, Stop)
  END
  ELSE
    BEGIN
      ult_var := Varx;
      IF simbolo = DOSPUNTOS THEN
      BEGIN
        Acepta(DOSPUNTOS, [IDENT] + Stop);
        Id_de_tipo(Tipox, Stop);
      END
      ELSE
        BEGIN
          Error_de_sintaxis(Stop);
          Tipox := TipoUniversal
        END
    END;
  Varx^.tipo_de_var := Tipox
END;

PROCEDURE def_de_variable(Stop: simbolos);
VAR
  ult_var, Tipox : Ptro;
```

```
BEGIN
  lista_variables(Variable, ult_var, Tipox,[PUNTOYCOMA] + Stop);
  Acepta(PUNTOYCOMA, Stop)
END;

PROCEDURE Parte_def_var(Stop : simbolos);
VAR
  Stop2: simbolos; ult_var: Ptro;
BEGIN
  Stop2 := [IDENT] + Stop;
  Acepta(VAR1, Stop2);
  def_de_variable(Stop2);
  WHILE simbolo = IDENT do
       def_de_variable(Stop2);
END;

PROCEDURE Def_de_param(VAR ult_param: Ptro; Stop: simbolos);
VAR
  Tipo : clas; Tipox : Ptro;
BEGIN
  Verifica_sintaxis([VAR1, IDENT] + stop);
  IF simbolo = VAR1 THEN
  BEGIN
    tipo := param_var;
    Acepta(VAR1, [IDENT] + Stop);
  END
  ELSE
    tipo := param_val;
  lista_variables(tipo,ult_param,Tipox,Stop);
END;

PROCEDURE List_param_formal(VAR ult_param : Ptro;Stop: simbolos);
VAR
  Stop2: simbolos;
BEGIN
  Stop2 := [PUNTOYCOMA] + Stop;
  Def_de_param(ult_param,Stop2);
  WHILE simbolo = PUNTOYCOMA do
  BEGIN
    Acepta(PUNTOYCOMA,inic_param + Stop2);
    Def_de_param(ult_param,Stop2);
  END;
END;

PROCEDURE Bloque(Stop : simbolos); forward;

PROCEDURE Def_de_procedimiento(Stop: simbolos);
VAR
  id: INTEGER; Proc: Ptro;
BEGIN
  Acepta(PROCEDURE1,[IDENT,PARENT_IZQ,PUNTOYCOMA]
        + inic_de_bloque+Stop);
  Acepta_id(id,[PARENT_IZQ, PUNTOYCOMA] + inic_de_bloque + Stop);
  Define(id,procedimiento,Proc);
```

```
  BloqueNuevo;
  IF simbolo = PARENT_IZQ THEN
  BEGIN
    Acepta(PARENT_IZQ, inic_param + [PARENT_DER, PUNTOYCOMA] +
          inic_de_bloque+Stop);
    List_param_formal(Proc^.ult_param,[PARENT_DER,PUNTOYCOMA]
                      + inic_de_bloque+Stop);
    Acepta(PARENT_DER, [PUNTOYCOMA] + inic_de_bloque + Stop)
  END
  ELSE { No hay lista de parámetros }
    Proc^.ult_param := NIL;
  Acepta(PUNTOYCOMA, [PUNTOYCOMA] + inic_de_bloque + Stop);
  Bloque([PUNTOYCOMA] + Stop);
  Acepta(PUNTOYCOMA, Stop);
  Final_de_bloque
END;

PROCEDURE Expresion(VAR Tipox:Ptro; Stop: simbolos); forward;

PROCEDURE selector_de_indice(VAR  Tipox : Ptro; stop : simbolos);
VAR
  Tipo_de_la_expr : Ptro;
BEGIN
  Acepta(CORCHETE_IZQ, inic_expr + [CORCHETE_DER] + Stop);
  Expresion(Tipo_de_la_expr,[CORCHETE_DER]+Stop);
  IF tipox^.clase = TipoArray THEN
  BEGIN
    Verifica_tipos(Tipo_de_la_expr, Tipox^.TipoIndice);
    Tipox := Tipox^.TipoElemento
  END
  ELSE
    BEGIN
      Error_de_clase(Tipox);
      Tipox := TipoUniversal;
    END;
  Acepta(CORCHETE_DER,Stop);
END;

PROCEDURE Selector_de_campo(VAR Tipox: Ptro; Stop: simbolos);
VAR
  encont: Boolean; campox: Ptro;
BEGIN
  Acepta(PUNTO, [IDENT] + Stop);
  IF simbolo = IDENT THEN
  BEGIN
    IF Tipox^.clase = TipoRecord THEN
    BEGIN
      encont := false;
      campox := Tipox^.ult_campo;
      WHILE not encont and (campox <> nil) do
        IF campox^.id <> argumento THEN campox := campox^.ant
        ELSE encont := true;
      IF encont THEN Tipox := campox^.Tipo_campo
      ELSE
```

```
        BEGIN
          Error(id_indef);
          Tipox := TipoUniversal
        END
    END
    ELSE
      BEGIN
        Error_de_clase(Tipox);
        Tipox := TipoUniversal
      END;
    Acepta(IDENT, Stop)
  END
  ELSE
    BEGIN
      Error_de_sintaxis(Stop);
      Tipox := TipoUniversal
    END
END;

PROCEDURE Accesa_var(VAR Tipox: Ptro; Stop: simbolos);
VAR
  Stop2 : simbolos; objeto: Ptro;
BEGIN
  IF simbolo = IDENT THEN
  BEGIN
    Stop2 := simb_selectores + simb_multiplicadores + Stop;
    Encuentra(argumento, objeto);
    Acepta(IDENT, Stop2);
    IF objeto^.clase in Variables THEN
       Tipox := objeto^.tipo_de_var
    ELSE
      BEGIN
        Error_de_clase(objeto);
        Tipox := TipoUniversal
      END;
    WHILE simbolo in simb_selectores DO
      IF simbolo = CORCHETE_IZQ THEN
         Selector_de_indice(Tipox, Stop2)
      ELSE {simbolo = PUNTO}
         Selector_de_campo(Tipox, Stop2)
  END
  ELSE
    BEGIN
      Error_de_sintaxis(Stop);
      Tipox := TipoUniversal
    END
END;

PROCEDURE Factor(VAR Tipox: Ptro; Stop: simbolos);
VAR
  Objeto: Ptro; valor: INTEGER;
BEGIN
  CASE simbolo OF
    CTE_ENT : Constante(valor,Tipox,simb_multiplicadores + Stop);
```

```
    IDENT   : BEGIN
                Encuentra(Argumento,Objeto);
                IF objeto^.clase = Constantex THEN
                   Constante(valor,Tipox, Stop)
                ELSE
                  IF objeto^.clase IN Variables THEN
                     Accesa_var(Tipox,Stop)
                  ELSE
                     BEGIN
                       Error_de_clase(Objeto);
                       Tipox := TipoUniversal;
                       Acepta(IDENT,Stop)
                     END
              END;

    PARENT_IZQ: BEGIN
                  Acepta(PARENT_IZQ,inic_expr+[PARENT_DER]+Stop);
                  Expresion(Tipox, [PARENT_DER] + Stop);
                  Acepta(PARENT_DER, simb_multiplicadores+Stop);
                END;

    NOT1 :      BEGIN
                  Acepta(NOT1, inic_de_factores + Stop);
                  Factor(Tipox,Stop);
                  Verifica_tipos(Tipox, TipoBoolean)
                END;
    ELSE        BEGIN
                  Error_de_sintaxis(Stop);
                  Tipox:=TipoUniversal
                END
  END  {CASE}
END;   {Factor}

PROCEDURE Termino(VAR tipox : Ptro; Stop : simbolos);
VAR
  operador : t_simb; Tipo2    : Ptro;
BEGIN
  Factor(Tipox,Stop);
  WHILE simbolo IN simb_multiplicadores DO
  BEGIN
    operador := simbolo;
    Acepta(simbolo,inic_de_factores + Stop);
    Factor(Tipo2,Stop);
    IF Tipox = TipoEntero THEN
    BEGIN
      Verifica_tipos(Tipox,Tipo2);
      IF operador = AND1 THEN Error_de_tipo(Tipox)
    END
    ELSE
      IF Tipox = TipoBoolean THEN
      BEGIN
        Verifica_tipos(Tipox,Tipo2);
        IF operador <> AND1 THEN Error_de_tipo(Tipox)
      END
```

```
      ELSE
        Error_de_tipo(Tipox)
  END
END;

PROCEDURE ExpresionSimple(VAR Tipox:Ptro; Stop : simbolos);
VAR
  Stop2 : simbolos; operador : t_simb; Tipo2 : Ptro;
BEGIN
  Stop2 := simb_sumadores + Stop;
  Verifica_sintaxis(signos + inic_term + Stop2);
  IF simbolo IN signos THEN
  BEGIN
    Acepta(simbolo,inic_term + Stop2);
    Termino(Tipox,Stop2);
    Verifica_tipos(Tipox,TipoEntero);
  END
  ELSE
    Termino(Tipox,Stop2);
  WHILE simbolo IN simb_sumadores DO
  BEGIN
    operador := simbolo;
    Acepta (simbolo,inic_term + Stop2);
    Termino(Tipo2,Stop2);
    IF Tipox = TipoEntero THEN
    BEGIN
      Verifica_tipos(Tipox,Tipo2);
      IF (operador <> MAS) AND (operador <> MENOS) THEN
         Error_de_tipo(Tipox);
    END
    ELSE
      IF Tipox = TipoBoolean THEN
      BEGIN
        Verifica_tipos(Tipox,Tipo2);
        IF operador <> OR1 THEN Error_de_tipo(Tipox);
      END
      ELSE
        Error_de_tipo(Tipox)
  END
END;

PROCEDURE Expresion (VAR Tipox:Ptro; Stop: simbolos);
VAR
  operador : t_simb; Tipo2 : Ptro;
BEGIN
  ExpresionSimple(Tipox, simb_relacionales + Stop);
  IF simbolo IN simb_relacionales THEN
  BEGIN
    operador := simbolo;
    Acepta(simbolo,inic_expr_SIMPLE + Stop);
    ExpresionSimple(Tipo2, Stop);
    IF Tipox^.clase = TipoEstandar THEN
       Verifica_tipos(Tipox, Tipo2)
    ELSE
```

```
      Error_de_tipo (Tipox);
    Tipox := TipoBoolean;
  END
END;

PROCEDURE Prop_ES(Stop : simbolos);
VAR
  id : INTEGER; Tipox : Ptro; Stop2 : simbolos;
BEGIN
  Stop2 := [PARENT_DER] + Stop; id := argumento;
  Acepta(IDENT, inic_expr + Stop2);
  Acepta(PARENT_IZQ, inic_expr + Stop2);
  IF id = ORD(READ1) THEN Accesa_var(Tipox,Stop2)
  ELSE Expresion(Tipox,Stop2);
  Verifica_tipos(Tipox,TipoEntero);
  Acepta(PARENT_DER,Stop2)
END;

PROCEDURE Lista_de_param_act(ult_param : Ptro; Stop: simbolos);
VAR
  Tipox : Ptro;
BEGIN
  IF ult_param^.ant <> NIL THEN
  BEGIN
    Lista_de_param_act(ult_param^.ant,[COMA]+inic_expr+stop);
    Acepta(COMA,inic_expr + Stop );
  END;
  IF ult_param^.clase = Param_val THEN Expresion(Tipox,Stop)
  ELSE Accesa_var(Tipox,Stop);
  Verifica_tipos(Tipox,ult_param^.tipo_de_var)
END;

PROCEDURE Prop_procedure(Stop: simbolos);
VAR
  Stop2 : simbolos; Proc : Ptro;
BEGIN
  Encuentra(argumento, Proc);
  IF Proc^.clase = Proc_estandar THEN Prop_ES(Stop)
  ELSE
     IF Proc^.ult_param <> NIL THEN
     BEGIN
       stop2 := [PARENT_DER] + Stop;
       Acepta(IDENT, [PARENT_IZQ] + inic_expr + Stop2);
       Acepta(PARENT_IZQ,inic_expr + Stop2);
       Lista_de_param_act(Proc^.ult_param,Stop2);
       Acepta(PARENT_DER, Stop)
     END
     ELSE ( No hay lista de parámetros )
        Acepta(IDENT,Stop)
END;

PROCEDURE Prop_asignacion(Stop: simbolos);
VAR
  tipo_de_la_var, Tipo_de_la_expr : Ptro;
```

```
BEGIN
  Accesa_var(tipo_de_la_var, [ASIGNACION] + inic_expr + Stop);
  Acepta(ASIGNACION, inic_expr + Stop);
  Expresion(Tipo_de_la_expr,Stop);
  Verifica_tipos(tipo_de_la_var,Tipo_de_la_expr)
END;

PROCEDURE Prop(Stop : simbolos); Forward;
PROCEDURE Prop_IF(Stop: simbolos);
VAR
  tipo_de_la_expr : Ptro;
BEGIN
  Acepta(IF1, inic_expr + [THEN1, ELSE1] + inic_de_props + Stop);
  Expresion(tipo_de_la_expr,[THEN1,ELSE1] +inic_de_props + Stop);
  Verifica_tipos(tipo_de_la_expr,TipoBoolean);
  Acepta(THEN1, inic_de_props + [ELSE1] + Stop);
  Prop([ELSE1] + Stop);
  IF simbolo = ELSE1 THEN
  BEGIN
    Acepta(ELSE1, inic_de_props + Stop);
    Prop(Stop)
  END
END;

PROCEDURE Prop_While(Stop: simbolos);
VAR
  Tipo_de_la_expr: Ptro;
BEGIN
  Acepta(WHILE1, inic_expr + [DO1] + inic_de_props + Stop);
  Expresion(Tipo_de_la_expr, [DO1] + inic_de_props + Stop);
  Verifica_tipos(Tipo_de_la_expr, TipoBoolean);
  Acepta(DO1, inic_de_props + Stop);
  Prop(Stop);
END;

PROCEDURE Prop_compuesta(Stop: simbolos); forward;

PROCEDURE Prop (Stop: simbolos);
VAR
  Objeto : Ptro;
BEGIN
  CASE simbolo OF
      IDENT  : BEGIN
                 Encuentra(argumento,Objeto);
                 IF objeto^.clase in Variables THEN
                    Prop_asignacion(stop)
                 ELSE
                    IF objeto^.clase in procedimientos THEN
                      Prop_PROCEDURE(stop)
                    ELSE
                      BEGIN
                        Error_de_clase(objeto);
                        Acepta(IDENT,Stop)
                      END
```

```
                END;
        IF1    : Prop_if(Stop);
        WHILE1 : Prop_WHILE(Stop);
        BEGIN1 : Prop_compuesta(Stop);
        ELSE     Verifica_sintaxis(Stop)
    END;
  END;

  PROCEDURE Prop_compuesta ( Stop: simbolos);
  BEGIN
    Acepta(BEGIN1, inic_de_props + [PUNTOYCOMA, END1] + Stop);
    Prop([PUNTOYCOMA, END1] + Stop);
    WHILE simbolo = PUNTOYCOMA DO
    BEGIN
      Acepta(PUNTOYCOMA,inic_de_props + [PUNTOYCOMA, END1] + Stop);
      Prop([PUNTOYCOMA, END1] + Stop)
    END;
    Acepta(END1, Stop)
  END;

  PROCEDURE Bloque  ( Stop: simbolos );
  BEGIN
    Verifica_sintaxis(inic_de_bloque + Stop);
    IF simbolo = CONST1 THEN
    Parte_de_def_ctes([TYPE1,VAR1,PROCEDURE1, BEGIN1] + Stop);
    IF simbolo = TYPE1 THEN
    Parte_def_tipos([VAR1,PROCEDURE1,BEGIN1] + Stop);
    IF simbolo = VAR1 THEN
       Parte_def_var([PROCEDURE1,BEGIN1] + Stop);
    WHILE simbolo = PROCEDURE1 DO
         Def_de_procedimiento([PROCEDURE1,BEGIN1] + Stop);
    Prop_compuesta(Stop);
  END;

  PROCEDURE Programa(Stop: simbolos);
  BEGIN
    Acepta(PROGRAM1,[IDENT,PUNTOYCOMA,PUNTO]+inic_de_bloque+Stop);
    Acepta(IDENT,[PUNTOYCOMA,PUNTO]+inic_de_bloque + Stop);
    Acepta(PUNTOYCOMA, [PUNTO] + inic_de_bloque + Stop);
    BloqueNuevo;
    Bloque([PUNTO] + Stop);
    Final_de_bloque;
    Acepta(PUNTO,Stop);
  END;

BEGIN
  Inicializa;
  Explora_simbolo;
  Block_estandar;
  Programa([PUNTO]);
  Termina;
END.
```

APENDICE A.4 Programas de prueba.

```
{ Mini-Pascal. Prueba1: Símbolos correctos }
Program prueba1;

and array begin const div do else
end if mod not of or procedure
program record then type var while

{ Identificadores estándar }

integer Boolean false true read write

dna dna

{ { Comentario } }

alfa1 x1 x2

0 32767

+ - * /

= < <= > >= <> :=
() [] , : ; ..
' ( )
.
```

```
{ Mini-Pascal. Prueba2: Análisis sintáctico }
Program prueba2;
const
   a = 1;
   b = a;
type
   T = array [1..2] of integer;
   U = record
         f,g : integer;
         h   : boolean
       end;
   V = record
         f : integer
       end;
var
   x,y : T;
   z : U;

   procedure P(var x:integer; y : boolean);
   const
      a = 1;
      procedure Q(x : integer);
      type
         T = array [1..2] of integer;
      begin
         x := -1;
         x := x;
         x := (2 - 1) * (2 + 1) div 2 mod 2;
         if x < x then
            while x = x do Q(x);
         if x > x then
            while x <= x do P(x,false)
         else
            if not (x <> x) then { vacio }
      end;
   begin
      if x >= x then y := true
   end;

   procedure R;
   var
     x : T;
   begin
      x[1] := 5
   end;
begin
   z.f := 6
end.
```

```
{ Mini-Pascal. Prueba3: Errores sintácticos }
program Prueba3;
const
   a := 1;
   b = 2;
   c = ;
   d = 4;
type
   s = recrod
         f, g : integer
       end;
   T = array [1..2] of integer;
var
   x : integer;
begin
   if = 2 then
      x := 1
end.
```

```
{ Mini-Pascal. Prueba 4: Análisis de alcance }
program Prueba4;
type
  S = record
        f, g : boolean
      end;
var
    v : S;

    procedure P(x: integer);
    const
        n = 10;
    type
        T = array[1..n] of integer;
    var
        y, z : T;

        procedure Q;
        begin
           read(x);
           v.g := false
        end;
    begin
      y := z;
      Q;
      P(5);
      write(x)
    end;
begin
   v.f := true;
   P(5)
end.
```

```
{ Mini-Pascal. Prueba 5: Errores de alcance }
program prueba5;
const
    {a} = 1;
    b   = b;
type
   T = array[1..10] of T;
   U = record
         f, g : U
       end;
var
  x, y, x : integer;
begin
  x := a;
  y := a
end.

{ Mini-Pascal. Prueba 6: Análisis de tipos }
program prueba6;
const
   a = 10;
   b = false;
type
   T1 = array[a..a] of integer;
   T2 = record
          f, g : integer;
          h    : Boolean
        end;
var
   x, y : integer;
   Z    : Boolean;

   procedure Q(var x: T1; z: T2);
   begin
      x[10] := 1;
      z.f := 1;
      Q(x,z)
   end;
   procedure P;
   begin
      Read(x);
      Write(x+1)
   end;
begin
   P;
   x := 1;
   x := a;
   x := y;
   x := - (x+1) * (y-1) div 9 mod 9;
   z := not b;
   z := z or z and z;
   if x <> y then
      while x < y do { vacía }
end.
```

```
{ Mini-Pascal. Prueba 7: Errores de tipos }
program prueba7;
type
   T = array[1..10] of integer;
var
   x : integer;
   y : Boolean;
   z : T;

   procedure P(x : integer);
   begin
   end;

begin
   y := not 1 and 2 and 3;
   y := false * true div false;
   z := z mod z;
   x := 1 or 2 or 3;
   y := false + true - true;
   z := z - z;
   if z <> z then
      P(true)
end.
```

```
{ Mini-Pascal. Prueba 8: Errores de clase }
program prueba8;
const
   a = integer;
type
   T = array[2..1] of integer;
   U = record
         f : integer
       end;
var
   x : integer;
   y : U;
   z : false;

   procedure P(var x : integer y : true);
   begin
   end;

begin
   x[1] := 1;
   x.f := 1;
   P(false, true);
   x := P;
   false := true;
   y.g := 1
end.
```

APENDICE B. GLOSARIO.

ALCANCE. Característica de la mayoría de los lenguajes de programación modernos en la que el nombre de una variable es conocido en una parte perfectamente definida del lenguaje fuente. Ver BLOQUE.

ANALISIS LEXICO. Fase inicial de un compilador que realiza el análisis lexicográfico del programa fuente, fragmentándolo en símbolos o tokens que facilitan su proceso posterior.

ARBOL DE RECONOCIMIENTO SINTACTICO. Es sinónimo de árbol sintáctico.

ARBOL SINTACTICO. Arbol cuya raíz es el axioma y que refleja la estructura sintáctica de una sentencia empleando para ello subárboles sintácticos.

Cada subárbol sintáctico representa gráficamente una regla BNF de la gramática implicada. Es una representación muy útil del resultado del análisis de una sentencia.

AXIOMA. Es la variable de una gramática que sirve de arranque para todos los reconocimientos, formando la raíz del árbol sintáctico. Se denomina también símbolo inicial.

BLOQUE. En lenguajes de programación tipo PASCAL, grupo de sentencias y declaraciones que suelen ir entre las palabras reservadas begin y end. El bloque supone dos características esenciales:

- al entrar en él, se consigue memoria para sus variables, que se libera al salir de él.

- es un límite para el alcance de las variables locales del bloque. El bloque tiene una relación muy estrecha con el alcance de las variables.

COMPILADOR. Programa que es capaz de realizar la traducción total y definitiva de un programa escrito en lenguaje fuente a un lenguaje objeto distinto y de menor nivel.

DESCENSO RECURSIVO. Parser del tipo descendente, en el que cada no terminal de la gramática se escribe como un procedimiento. Al realizar el reconocimiento es casi inevitable que se realicen llamadas recursivas de los procedimientos, de donde proviene su nombre.

DIRECCION ABSOLUTA. Una dirección real en el almacenamiento de una unidad particular de datos; una dirección que la unidad de control puede interpretar directamente.

DIRECCION DE MEMORIA. Toda palabra en memoria tiene una sola dirección. Una palabra se puede definir como un conjunto de bits comprendiendo la unidad de información direccionable más larga en una memoria programable. La dirección de una palabra es su posición en la memoria.

DIRECCION SIMBOLICA. Es la identificación arbitraria de una palabra particular, función u otra información sin considerar la posición de la información.

ENSAMBLADOR. Programa que traduce un lenguaje simbólico de bajo nivel a lenguaje máquina. En algunos casos realiza un paso antes de expansión de macroinstrucciones antes de la traducción.

Existen unas instrucciones de control llamadas directivas de ensamblador que crean un ambiente adecuado para la traducción, así: fijan el origen de direcciones, eligen los registros de base, controlan el formato del listado resultante, etc.

GENERADOR DE CODIGO. Fase de un compilador que genera el código máquina resultado final de la traducción. Este código suele ser para la misma máquina en la que funciona el compilador. Esta fase constituye la diferencia esencial entre un compilador y un intérprete.

GRAMATICA. Forma teórica de definición de un lenguaje o conjunto de cadenas formadas con un cierto vocabulario.

GRAMATICA DE CONTEXTO LIBRE. Es el Tipo 2 de la Clasificación de Chomsky de las gramáticas. Las reglas de la gramática son del tipo A ---> alfa, siendo A una variable o no terminal y alfa una cadena formada por variables y/o no terminales.

GRAMATICA LL(k). Una gramática de contexto libre se denomina LL(k), si se puede decidir qué regla aplicar en una derivación a izquierdas, observando sólo k símbolos a la derecha del símbolo actual. Caracteriza los análisis sintácticos descendentes.

HASHING. Forma de conseguir accesos muy rápidos a tablas mediante la obtención de una dirección especial, que se pretende sea lo más única o específica posible. Dicha dirección no guarda ninguna relación con los datos originales, salvo para su localización.

INTERPRETE. Traductor que en vez de actuar una sóla vez en la vida del programa a traducir, va traduciendo justo la sentencia que se va a ejecutar. Tiene unas ventajas indudables frente a los compiladores. Actualmente hay

productos que reunen lo mejor de los intérpretes y de los compiladores.

LENGUAJE ENSAMBLADOR. Lenguaje de programación simbólico que tiene una instrucción por línea. La instrucción puede tener cuatro partes separadas entre sí por espacios:

- una etiqueta opcional,
- un código de operación,
- los operandos,
- comentario opcional.

El resultado de la traducción con ensamblador puede ser:

a) objeto, que servirá para su ulterior montaje (con el montador de enlaces) y ejecución,

b) un listado en donde dará los errores cometidos (si los hubiere) y,

c) un listado de referencias cruzadas.

LENGUAJE FUENTE. Es el lenguaje de programación que entiende y es capaz de traducir un compilador.

LENGUAJE INTERMEDIO. Lenguaje que de acuerdo con su nombre se encuentra entre el lenguaje fuente y el lenguaje objeto de un compilador. Mediante él, se simplifica el proceso de traducción.

LENGUAJE OBJETO. Lenguaje resultado de la compilación de un programa fuente. Suele coincidir con el código máquina de la computadora en que se realiza la compilación.

LENGUAJE. Conjunto de cadenas de símbolos o caracteres que se forman con el vocabulario llamado terminal. Salvo en los casos muy simples, para la definición de un lenguaje se tendrá que recurrir a una gramática adecuada para ello (p. ej. una gramática de contexto libre).

LISTA ENCADENADA. Lista de elementos, cada uno de ellos tiene un campo de enlace para localizar al siguiente elemento de la lista.

Es una estructura dinámica de información y se recomienda su uso en los casos en que la variabilidad de la información y el desconocimiento del número de elementos hacen difícil el tener una estructura preplanificada.

MACROINSTRUCCION. Sentencia o instrucción de un lenguaje que, a modo de abreviatura, se expansiona en varias

sentencias del mismo lenguaje, facilitando así la escritura de un programa. El paso de expansión de la macro suele ser previo al de compilación.

MONTADOR DE ENLACES. Programa de utilidad que realiza el montaje de los módulos objeto suministrando un "módulo cargable" listo para su ejecución.

NOTACION POLACA INVERSA. Notación debida a Lukasiewicz que permite escribir las expresiones aritméticas sin paréntesis y de una forma no ambigua. Es fácil ampliarla para representar las sentencias de un lenguaje de programación.

PARAMETRO. Si no lleva calificativo se sobreentiende parámetro ficticio, o sea el parámetro que aparece en la cabecera del subprograma y que le sirve a éste para definir o escribir el algoritmo de cálculo.

Posteriormente, cuando se realice la llamada del subprograma, el parámetro se sustituye por el argumento o expresión correspondiente de la llamada.

PARSER. Es un reconocedor sintáctico que además construye el árbol sintáctico de la sentencia analizada.

El reconocimiento es o bien el árbol sintáctico, o bien una secuencia de dígitos indicando las reglas BNF que hay que ir aplicando (y en qué orden) para derivar la sentencia en cuestión. Si la sentencia fuera incorrecta proporcionará mensajes adecuados para su mejoramiento.

Se complementa con rutinas semánticas para que además del análisis sintáctico realice la traducción del lenguaje fuente a lenguaje intermedio.

PASCAL. Lenguaje diseñado por Niklaus Wirth en 1970 y pensado especialmente para la enseñanza de la programación, objetivo que ha conseguido con creces.

PREPROCESADOR. Traductor que permite realizar pequeños cambios a un lenguaje fuente, normalemente como paso previo a su compilación.

Algunos ejemplos son:

1. El RATFOR que consigue un lenguaje parecido al C para un compilador p. ej. de FORTRAN;

2. El SIMPL/I es un lenguaje de simulación, similar al GPSS, obtenido con un preprocesador del PL/I;

PROGRAMA REUBICABLE. Se trata de un programa qué se escribe de modo que pueda ser localizado y ejecutado desde muchas áreas en memoria.

RECONOCEDOR. Algoritmo o programa que dada una cadena de símbolos terminales la acepta o la rechaza según que pertenezca o no al lenguaje correspondiente.

RECURSIVIDAD. Característica de las reglas de una gramática que permite ampliar su campo de aplicación. Facilita que una gramática con un número limitado de reglas recursivas represente un lenguaje con un número infinito de sentencias.

TABLA DE SIMBOLOS. Estructura de información que sirve para almacenar las variables y otros objetos del lenguaje fuente durante el proceso de compilación. A veces perdura después de la compilación para facilitar mensajes más inteligibles al usuario en la ejecución del programa traducido.

TOKEN. Es el símbolo resultado del análisis lexicográfico. Al funcionar el analizador léxico va fragmentando el programa, escrito en lenguaje fuente, entregando símbolos que facilitan considerablemente el trabajo posterior del compilador.

TRADUCTOR. Término genérico referido a cualquier programa o máquina teórica que es capaz de traducir una cadena de caracteres escrita en un lenguaje fuente a otra escrita en lenguaje objeto.

BIBLIOGRAFIA.

AHO, ALFRED V. AND J.D. ULLMAN.
Principles of Compiler Design.
Addison-Wesley, Reading, Mass.

AHO, ALFRED V., RAVHI SETHI AND J. D. ULLMAN.
Compilers Principles, Techniques, and Tools.
Addison-Wesley Publishing Company.

AHO, ALFRED V. AND J. D. ULLMAN.
The Theory of Parsing, Translation and Compiling,
Vol. I: Parsing.
Prentice-Hall, Englewood Cliffs, N. J.

AHO, ALFRED V. AND J.D. ULLMAN.
The Theory of Parsing, Translation and Compiling,
Vol. II: Compiling.
Prentice-Hall, Englewood Cliffs, N. J.

BACKHOUSE, R.C.
Syntax of Programming Languages: Theory and Parctice.
Prentice, Englewood Cliffs, N.J.

BRINCH HANSEN, P.
On Pascal Compilers.
Prentice, Englewood Cliffs, N.J.

DAVID GRIES.
Compiler Design.
Springer-Verlag.

HOPCROFT, J. E. AND J. D. ULLMAN.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, Reading, Mass.

HOPCROFT, J. E. AND J. D. ULLMAN.
Formal Languages and Their Relation to Automata.
Addison-Wesley, Reading, Mass.

JEAN PAUL TREMBLAY AND PAUL SORENSON.
Compiler Writing.
Mc Graw-Hill.

WAITE AND GOSS.
Compiler Construction.
Springer-Verlag.

WELSH, J., AND MCKEAG, M.
Structured System Programming.
Prentice, Englewood Cliffs, N.J.