UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS PROFESIONALES
Y DE POSGRADO
DEL COLEGIO DE CIENCIAS Y HUMANIDADES

INSTITUTO DE INVESTIGACIONES EN MATEMATICAS
APLICADAS Y EN SISTEMAS

DISEÑO E IMPLANTACION DE HERRAMIENTAS DE SOFTWARE PARA COMPILADORES

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

CIRO ARAUJO RAMIREZ

DIRECTOR: ING. MARIO RODRIGUEZ MANZANERA

MEXICO, D. F. ABRIL DE 1996

Para "Alis"

Por su motivación,

apoyo,

cariño

y

amor.

AGRADECIMIENTOS.

- A mi familia por su cariño, comprensión y apoyo.

- Al Ing. Mario Rodríguez Manzanera, por haber dirigido esta tesis. A la Doctora Hanna Oktaba, a la M. en C. Sofía Galicia Haro, a la M. en C. Guadalupe Ibargüengoitia González y al M. en. C. Javier García García por haber aceptado ser mis sinodales en el examen de grado.

- A mis profesores y amigos de la maestría.

Ciro Araujo Ramírez

México, D.F. abril de 1996.

CONTENIDO.

6. GENERACION DE CODIGO.

INTRODUCCION.

En algunas situaciones, dentro del área de desarrollo de sistemas de información, se presenta el problema de escribir manualmente un programa que se comporte como un compilador a fin incluirlo en alguna otra aplicación, como por ejemplo un programa para analizar sintácticamente ecuaciones matemáticas. Para poder escribir un programa con estas características, se requiere contar con técnicas y herramientas de software que simplifiquen dicha tarea y que puedan utilizarse 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 utilizadas en la construcción de un compilador.

Esta tesis no intenta cubrir diferentes métodos de compilación, como lo hacen la mayor parte de los textos que existen sobre compiladores, ya que esto con frecuencia confunde al lector principiante. En vez de eso, aquí se explica en detalle el desarrollo completo de un compilador utilizando el método de descenso recursivo.

En el primer capítulo se discuten las fases de un compilador, su interacción con la tabla de símbolos y el manejo de errores. Concluye con el diseño y programación de un parser/traductor.

En el segundo capítulo se describe la función del analizador léxico y se define el modelo del lenguaje Mini-Pascal, un subconjunto del Pascal, para el cual se desarrollará la programación de un analizador léxico.

El tercer capítulo está dedicado al análisis sintáctico. Primeramente, se discuten dos clases de análisis sintáctico: ascendente y descendente, haciendo énfasis en el análisis sintáctico descendente. A continuación se define la sintaxis del lenguaje Mini-Pascal y se desarrolla su analizador sintáctico.

El cuarto capítulo trata el análisis de alcance, el cual es una extensión del análisis sintáctico. Para realizar el análisis de alcance, primero se definen las reglas de alcance y los objetos de un programa en Mini-Pascal y después de describen los algoritmos que se utilizarán para este tipo de análisis.

El quinto capítulo describe la forma en la que el compilador utiliza 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.

CAPITULO 1. INTRODUCCION A LOS COMPILADORES.

En este Capítulo se da una idea general del proceso de compilación mediante la descripción de los componentes de un compilador. Se concluye con la construcción de un pequeño parser/traductor para analizar y traducir a lenguaje ensamblador para el µp 80x86 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.

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 primera se encarga de separar el programa fuente en varios componentes y crear una representación intermedia del programa fuente, mientras que la segunda construye el programa objeto correspondiente a partir de la representación intermedia[Aho-Ullman, 1986].

Durante el análisis, se determinan las operaciones contenidas en el programa fuente y se registran en un árbol sintáctico, en el cual cada nodo interno representa una operación, mientras que los nodos externos 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.

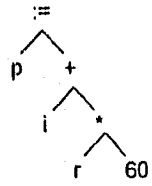


Fig. 1.1 Arbol sintáctico para la proposición p := i+r*60.

1.1.2 El contexto de un compilador.

Además del compilador, se requieren otros programas para crear un programa objeto. Un programa fuente se puede dividir en varios módulos almacenados en archivos separados. La tarea de conformar el programa fuente es algunas veces realizada por un programa distinto, llamado preprocesador.

La figura 1.2 muestra el esquema de una compilación típica. El compilador de esta figura crea código en lenguaje ensamblador, el cual posteriormente se ensambla para generar el código máquina que será encadenado con algunas bibliotecas y pueda realmente, ejecutarse en la computadora.

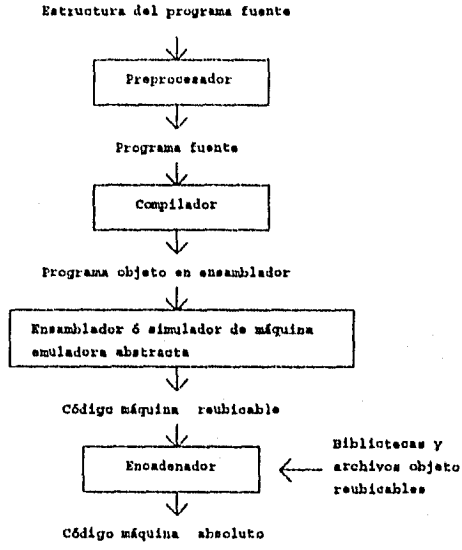


Fig. 1.2 Esquema típico de la compilación de un programa.

1.2 Análisis del programa fuente.

En la compilación, el análisis consiste de tres fases:

i) Análisis lineal, en el que 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 tokens se agrupan jerárquicamente en colecciones anidadas con un significado colectivo.

iii) Análisis semántico, en éste se verifica que los componentes de un programa tengan un significado real.

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, se pueden agrupar varias fases en una.

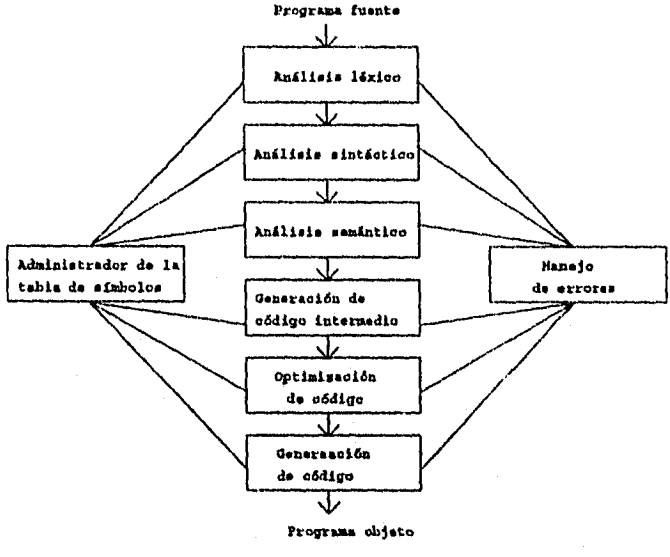


Fig. 1.3 Fases de un compilador.

En la figura 1.3 también se muestran otras dos actividades adicionales, la creación de la tabla de símbolos y el manejo de errores, interactuando con las seis fases.

1.3.1 Análisis léxico.

En un compilador, el análisis lineal se denomina análisis léxico, en el cual los caracteres de la proposición (1.1) se agrupan en los siguientes tokens: El identificador p, el símbolo de asignación, el identificador i, el signo +, el identificador r, el signo * y el número 60.

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

1.3.2 Análisis sintáctico.

El análisis jerárquico se llama análisis sintáctico. Consiste en agrupar los tokens del programa fuente en frases gramaticales que utiliza el compilador para sintetizar la salida. Usualmente, las frases gramaticales del programa fuente se representan por un árbol de reconocimiento sintáctico como el de la figura 1.4.

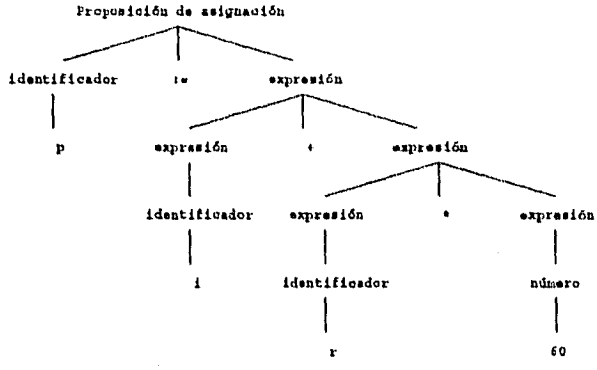


Fig. 1.4 Arbol de reconocimiento sintáctico para p:=i+r*60.

Debido a la estructura jerárquica que representa un programa, usualmente se expresa mediante reglas recursivas. Por ejemplo, podemos tener las siguientes reglas como parte de la definición de expresiones aritméticas:

Regla 1. Cualquier identificador es una expresión.
Regla 2. Cualquier número es una expresión.
Regla 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 la regla 3 define expresiones en términos de operadores aplicados a otras expresiones. Entonces para la proposición (1.1), 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 que r*60 es una expresión y que i+r*60 también lo es.

En forma similar, varios lenguajes definen proposiciones recursivamente mediante reglas. Por ejemplo:

i) Si id_1 es un identificador, y $expresión_2$ es una expresión, entonces

$$id_1 := expresión_2$$

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 arbitraria. Usualmente, se selecciona una división que simplifique la tarea completa de análisis. Un factor en determinar tal división es verificar si las construcciones del lenguaje fuente son recursivas o no lo son. Las construcciones léxicas no requieren recursión, mientras que las construcciones semánticas si.

Por ejemplo, no se requiere recursión para reconocer identificadores, los cuales por lo regular son cadenas de letras y digitos. Normalmente, reconoceremos identificadores mediante una exploración simple de la secuencia de entrada, continuando hasta que se encuentre un carácter que no sea una letra o un digito, y después los agrupamos en un token de identificador. Los caracteres asi agrupados se registran en la tabla de símbolos, eliminándolos de la entrada de tal forma que se pueda empezar el procesamiento del siguiente token.

Este tipo de exploración lineal no es eficiente para analizar expresiones o proposiciones. Por ejemplo, no podemos agrupar paréntesis adecuadamente en expresiones, o el begin y el end en proposiciones sin considerar algún mecanismo de precedencia o estructura anidada en la entrada.

El árbol de reconocimiento sintáctico de la figura 1.4 describe la estructura sintáctica de la proposición p:=i+r*60. 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 las hojas de ese nodo.

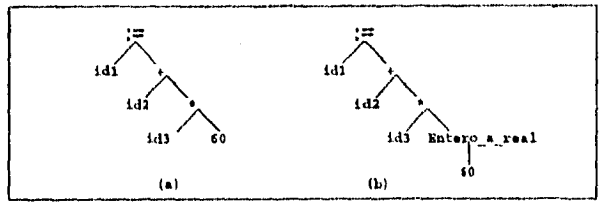


Fig. 1.5 Análisis semántico para insertar la conversión de entero a real.

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. Utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

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, algunas definiciones de lenguajes de programación requieren del compilador para que reporte un error cada vez que un número real se utilice como un índice en un arreglo. Sin embargo, la especificación del lenguaje puede permitir algunas conversiones de operandos, por ejemplo, cuando un operador aritmético binario se aplica a un entero y a un real. En este caso, el compilador puede necesitar convertir el entero a real.

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 * se aplica 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 entero_a_real que explícitamente realiza la conversión.

1.3.4 Administación de la tabla de símbolos.

La función inicial de un compilador es registrar los identificadores utilizados en el programa fuente y reunir información 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, y, en el caso de procedimientos, algunas características como el número y tipo de sus argumentos, el método para transferirlos, por ejemplo, por referencia, y el tipo devuelto, si es que existe.

Una tabla de símbolos es una estructura de datos que contiene un registro para cada identificador, con campos para sus atributos.

Cuando se detecta un identificador en el programa fuente mediante el análisis léxico, se coloca en la tabla de símbolos.

Las fases restantes introducen información referente a los identificadores dentro de la tabla de símbolos y después la utilizan de varias formas. Por ejemplo, cuando se realiza el análisis semántico y la generación de código intermedio, se requiere saber el tipo de los identificadores, para poder verificar que el programa fuente los use en forma válida, y así pueda generar las operaciones apropiadas con ellos. El generador de código típicamente introduce y utiliza información detallada acerca de la memoria asignada a los identificadores.

1.3.5 Detección y reporte de errores.

En cada fase se pueden detectar errores. Por lo tanto, es necesario que una fase se encargue de atenderlos, de tal forma que la compilación pueda proceder, permitiendo que los errores subsiguientes del programa también puedan ser detectados.

Las fases del análisis sintáctico y semántico usualmente manejan gran parte de los errores detectados por el compilador. La fase lexicográfica puede detectar errores donde los caracteres que quedan en la entrada no forman un token válido para el lenguaje. Los errores donde el token viola las reglas sintácticas del lenguaje se detectan en la fase del análisis sintáctico.

Durante el análisis semántico el compilador trata de detectar construcciones que tienen la estructura sintáctica correcta, pero que 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, deberá marcarse error de semántica.

1.3.6 Generación de código intermedio.

Después del análisis sintáctico y del semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Esta representación intermedia debe 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 éstas se conoce como "código de tres direcciones", en la cual cada localidad de memoria puede actuar como un registro y cada instrucción tiene a lo más tres operandos. La proposición (1.1) puede representarse 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 tiene a lo más un operador, además del de asignación. Por lo tanto, cuando se generan estas instrucciones, el compilador tiene que decidir, el orden en el cual se efectuarán las operaciones. Segunda, el compilador debe generar un nombre de variable y un espacio temporal para almacenar el valor calculado por cada instrucción. Tercera, algunas instrucciones tienen menos de tres operandos, por ejemplo, la primera y última instrucción en (1.2).

1.3.7 Optimización de código.

Esta fase intenta mejorar y minimizar el código intermedio, de tal forma que resulte un código de máquina más rápido de ejecutar. Por ejemplo, un algoritmo puede generar el código intermedio mostrado en (1.2), utilizando una instrucción para cada operador en el árbol representado en la figura (1.6) después del análisis semántico, aunque existe una forma mejor para realizar el mismo cálculo, utilizando dos instrucciones:

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

En este caso, el compilador deduce que la conversión del número 60 de entero a real puede hacerse desde la fase de compilación, por lo tanto, la operación de conversión de entero a real puede eliminarse. Además, temp3 se utiliza sólo una vez, para transmitir su valor a id1. Por lo tanto, se puede sustituir id1 por temp3, con lo cual la última proposición de (1.2) no se necesita y el resultado es el código de (1.3).

1.3.8 Generación de código.

La fase final del compilador es la generación de código objeto, que por lo general consiste de código de la máquina en donde puede ejecutarse ó código ensamblado. Se seleccionan las localidades de memoria para cada una de las variables utilizadas en 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, utilizando los registros 1 y 2, la traducción del código de (1.3) es como se muestra en (1.4).

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

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. El # significa que el número 60.0 va a ser tratado como una constante. La tercer instrucción transfiere el valor de id2 a R1. La cuarta instrucción suma a R1 el valor previamente calculado en R2. Finalmente, el valor de R1 se pasa a la dirección de id2, implementando así el código de la proposición de asignación de la figura 1.6.

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 (1.1).

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 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. La secuencia de caracteres que forman un token se llama lexema.

A ciertos tokens se les agregará un "valor léxico". Así, cuando se encuentra un identificador, como por ejemplo r, el analizador léxico no sólo genera un token, escrito id, sino también introduce el lexema r en la tabla de símbolos, si es que aún no está allí.

Utilizaremos id1, id2, e id3 para p, i, y r respectivamente, para enfatizar que la representación interna de un identificador es diferente de la secuencia de caracteres que lo forman. La representación de p:=i+r*60 después del análisis léxico es como se ilustra en (1.5).

```
id1 := id2 + id3 * 60          (1.5)
```

También debemos crear tokens para el operador multicarácter := y el número 60 para reflejar su representación interna.

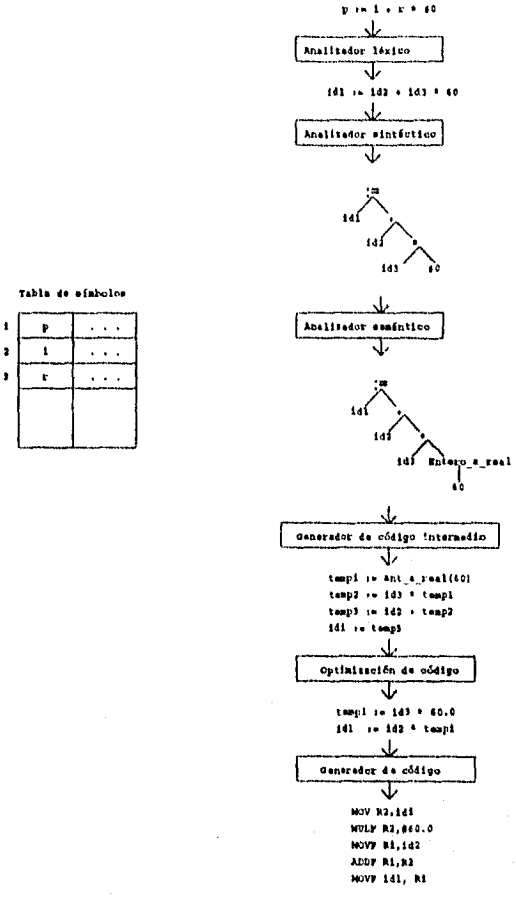


Fig. 1.6 Ejemplo de la compilación de una proposición.

1.4 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 utilizaremos en el contexto de análisis y traducción de expresiones aritméticas, pero las técnicas también se aplican a la traducción de otros elementos.

1.4.1 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:

```
x = 2*7+8-4/2
y = (5+6) + (5*2)/(4-2)
```

1.4.2 Caracteristicas del parser/traductor.

El parser para esta gramática tiene las características siguientes:

i) Entada a través del teclado y salida por el 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) Utiliza análisis sintáctico de descenso recursivo.

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 80x86 conforme se realiza el análisis sintáctico.

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.

1.4.3 Diseño.

Empezaremos con la definición de una clase general, la cual denominaremos "Nodo", que proporcione la estructura y funcionalidad común para todos los tipos de nodos del árbol sintáctico que genera el compilador durante la traducción. De esta clase se derivarán otras dos: una que refleje las similitudes de los nodos binarios y otra las similitudes de los nodos unarios. La primera se llamará "OpBin" y la segunda "OpUnario".

Utilizaremos clases derivadas de la clase OpBin para crear los distintos tipos de nodos para los operadores +, -, *, y /.

Crearemos una entrada en la tabla de símbolos para un identificador la primera vez que se utilice, para lo cual definiremos las clases "Id" y "Entrada". También definiremos una clase "Asignación" para almacenar el resultado de la expresión.

El diseño completo se muestra en el diagrama de clases de la figura 1.7.

1.4.4 Programación.

El lenguaje que utilizaremos para la programación del parser/traductor será C++, debido a las características que tiene como lenguaje orientado a objetos.

La declaración de la clases Nodo y OpBin se muestran a continuación:

```
class Nodo
{
   public:
     Nodo() {}
     virtual ~Nodo() {}
     virtual int evalua()=0;
     virtual void impr(ostream& salida,int ) = 0;
};

class OpBin : public Nodo
{
   protected:
    Nodo *izquierdo, *derecho;
   pubiic:
    ~OpBin() { delete izquierdo; delete derecho;}
     OpBin(Nodo* izq,  Nodo* der) { izquierdo = izq; derecho   = der;}
     virtual void impr_op(ostream& ,int)=0;
     void impr(ostream& salida,int profundidad) {
          izquierdo->impr(salida,profundidad+3);
          impr_op(salida,profundidad);
          derecho->impr(salida,profundidad+3);
     }
};
```

Los métodos evalua() e impr() se declaran virtuales en la clase Nodo. Utilizar métodos virtuales permite que las clases derivadas proporcionen su propia versión de éstos.

La abstracción de similitudes de un grupo de nodos relacionados se muestra en el método impr_op() de la clase base OpBin. Los detalles de como imprimir objetos específicos están encapsulados en sus clases derivadas:

```
class Mas : public OpBin
{
   public:
      Mas(Nodo* izq, Nodo* der) : OpBin(izq,der) { }
      int evalua() { return izquierdo->evalua() + derecho->evalua(); }
      void impr_op(ostream& salida,int profundidad)
      {
         impr_esp(profundidad);
         salida << "+" << "\n";
      }
};

class Menos : public OpBin
{
   public:
      Menos(Nodo* izq, Nodo* der) : OpBin(izq,der) { }
      int evalua() { return izquierdo->evalua() - derecho->evalua(); }
      void impr_op(ostream& salida,int profundidad)
      {
         impr_esp(profundidad);
         salida << "-"  << "\n";
      }
};
```

La interacción entre las instancias de estas clases inicia cuando un objeto parser envía el mensaje Obt_token() a un objeto analizador léxico, el cual lo obtiene de la cadena de entrada. Los métodos del parser: Expresión(), Factor() y Termino() son reconocedores: aceptan un token del tipo correspondiente y envían nuevamente el mensaje Obt_token() al analizador léxico para obtener el siguiente token antes de regresar.

Para generar el código en ensamblador, se utilizan las clases generador de código y pantalla, cuyos métodos se encargan de desplegar el código resultante de la traducción.

La razón por la cual utilice los nodos del árbol sintáctico como clases se debe a que un nodo puede representar una variedad de construcciones de un lenguaje de programación. Cada uno de estos tipos de nodos comparte propiedades comunes con los otros, pero cada uno tiene propiedades adicionales que lo distinguen de los otros. Por lo tanto, conviene crear una jerarquía de clases que pueda ser reutilizada para otras construcciones.

1.4.5 Prueba.

Al ejecutar el programa del parser/traductor obtenemos:

Teclea la expresión de entrada: 2*7+8-4/2

```
MOV AX,2    ; AX = 2
PUSH AX     ; Coloca el valor 2 en la pila
MOV AX,7    ; AX = 7

; Se efectua la multiplicación:
MOV CX,AX   ; Segundo factor en el registro CX, esto es, CX=7
POP AX      ; Se saca de la pila el primer factor, AX=2
MUL CX      ; AX = AX*CX, es decir, AX = 14

PUSH AX     ; Coloca el valor 14 en la pila
MOV AX,8    ; AX = 8

; Se realiza la suma:
MOV BX,AX   ; Segundo sumando en BX, es decir BX=8
POP AX      ; El Primer sumando, lo extrae de la pila, AX=14
ADD AX,BX   ; AX = AX + BX, es decir, AX=22

PUSH AX     ; Coloca el valor 22 en la pila
MOV AX,4    ; AX = 4
PUSH AX     ; Coloca el valor 4 en la pila
MOV AX,2    ; AX = 2

; Se efectua la división:
MOV CX,AX   ; Divisor en CX, es decir, CX =2
POP AX      ; El dividendo lo saca de la pila, AX=4
DIV CX      ; AX = AX/CX, en este caso AX= 2


; Se efectua la resta:
MOV BX,AX   ; El sustraendo en BX, en este caso BX= 2
POP AX      ; El minuendo lo saca de la pila, AX=22
SUB AX,BX   ; AX = AX-BX, esto es, AX= 20
```

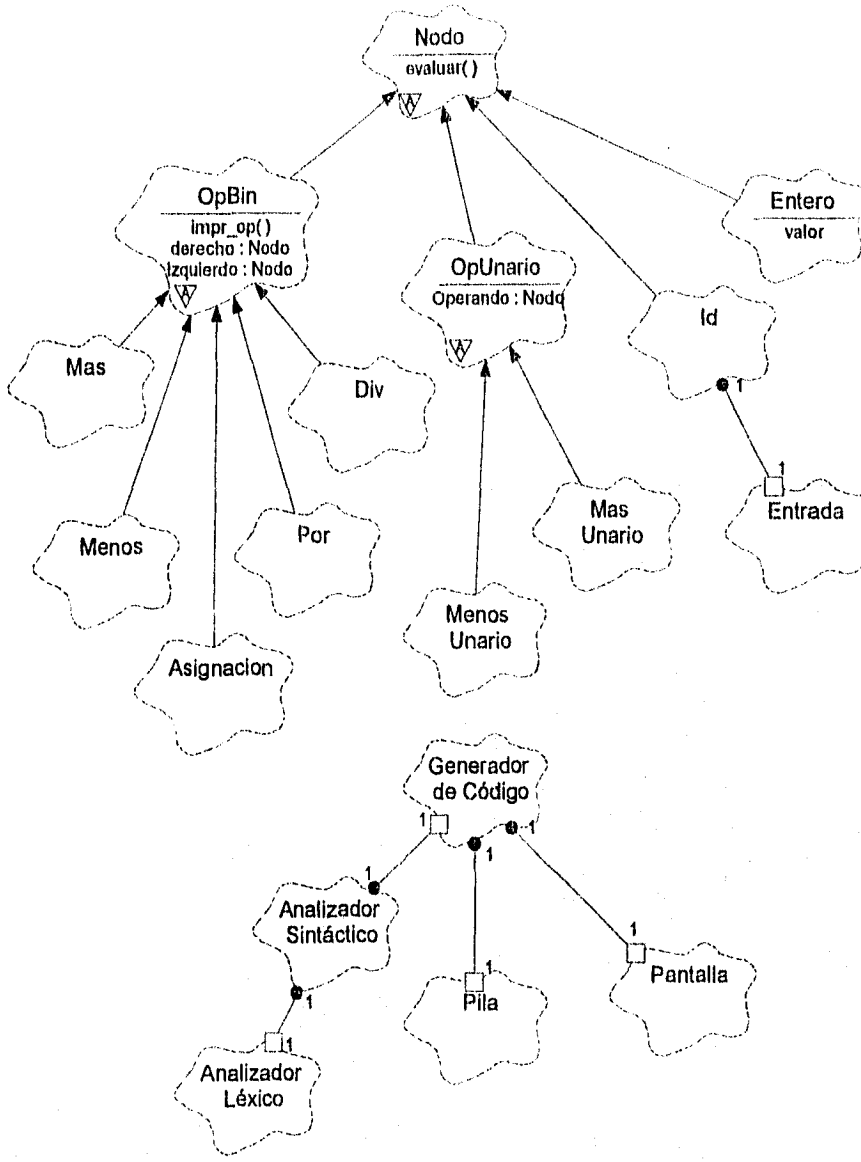


Fig. 1.7 Diagrama de clases para el parser/traductor de la gramática 1.

CAPITULO 2. ANALISIS LEXICO.

Introducción.

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

2.1. El papel del analizador léxico.

La tarea principal del analizador léxico 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 el mensaje "obtener el siguiente token", enviado por el parser, el analizador léxico lee caracteres de entrada hasta formar el siguiente token.

Puesto que el analizador léxico es la parte del compilador que lee el programa fuente, realiza tareas como saltar los comentarios y los espacios en blanco en la forma de carácter tabulador, blanco y nuevalínea. Otra tarea es relacionar los mensajes de error del compilador con el programa fuente.

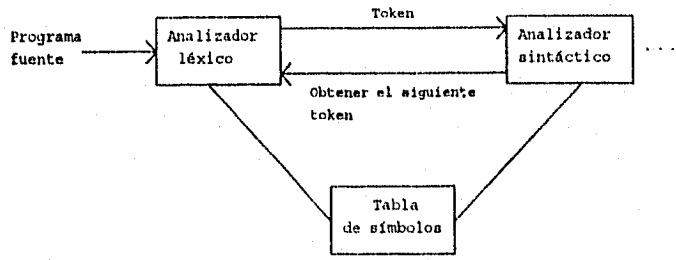


Fig. 2.1 Interacción del analizador léxico con el parser.

2.1.1 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 como salida el mismo token. Este conjunto de cadenas se describe mediante una regla llamada patrón asociado al token. 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 y dig.
num		Cualquier constante numérica
literal		Cualquier serie de caracteres encerrados entre comillas simples excepto '.

Fig. 2.2 Ejemplo de tokens.

En varios lenguajes de programación, las siguientes construcciones se tratan 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).

2.2 Definición del lenguaje Mini-Pascal.

En esta sección se define el lenguaje de programación para el cual desarrollaremos el analizador léxico, el 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. El lenguaje definido, el cual se llama Mini-Pascal, es un subconjunto del lenguaje Pascal que se utiliza en esta tesis para ilustrar las técnicas y problemas que surgen en la construcción de compiladores [Welsh-McKeag, 1980].

2.2.1 Resumen del Lenguaje.

Un programa de computadora consiste de dos partes esenciales, una descripción de acciones que van a efectuarse, y una descripción de los datos que serán manipulados por éstas. Las acciones se describen 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 le asocia un identificador y un tipo de dato.

Las proposiciones de asignación, lectura, escritura, y de procedimientos son los componentes de bloques de proposiciones estructuradas. La ejecución secuencial de proposiciones se especifica mediante una proposición compuesta, la ejecución condicional o selectiva por la proposición **if** y la ejecución de ciclos por la proposición **while**.

A un procedimiento se le asociará un nombre, a través de su declaración, la cual puede contener adicionalmente, un conjunto de declaraciones de variables, o declaraciones de procedimientos, para formar un bloque. Puesto que pueden declararse procedimientos dentro los bloques que definen otros procedimientos, los bloques podrán anidarse. Esta estructura de bloques anidados determina el alcance de los identificadores de variables, procedimientos, constantes, y también determinan la vida de las variables.

2.2.2 Notación, terminología y vocabulario.

Se utilizará la forma de Backus-Naur (BNF) ampliada, para describir las construcciones sintácticas.

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
<Digito> ::= 0|1|2|3|4|5|6|7|8|9

<Símbolos 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.

```
<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 y entero, pueden tomar.

Se utiliza la notación decimal usual para constantes de tipo entero.

```
<Constante entera> ::= <Digito> [<Digito>]
```

2.2.5 Tipos de datos.

Los tipos estándar de Mini-Pascal serán: Entero y boolean.

El tipo arreglo es una estructura que consiste de un número fijo de componentes del mismo tipo, llamado tipo de componente. Los elementos de un arreglo se designan por índices, los valores pertenecen a un cierto rango de índices. La definición de tipo arreglo especifica el tipo de componente y también el rango de los índices.

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

2.2.6 Declaración de variables.

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

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

Una denotación de variable designará a una variable entera o una variable indexada.

```
<Variable> ::= <Variable entera> | <Variable indexada>
<Variable entera> ::= <Identificador de variable>
<Identificador de variable> ::= <Identificador>
```

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>
```

2.2.7 Expresiones.

Las expresiones consisten de operandos (esto es, variables y constantes) y operadores.

El operador NOT tiene la precedencia más alta, le siguen los operadores multiplicadores, después los operadores sumadores, y finalmente, con la precedencia más baja, los operadores de relación. Las secuencias de operadores de la misma precedencia se ejecutan de izquierda a derecha. 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	Términos	Expresiones simples	Expresiones
x	x*y	-x	x = 1
15	(x<=y) AND (y<z)	i*j+k	p <= q
(x+y+z)			(i<j) = (j<k)
NOT p			

2.2.8 Operadores.

Operador not. Denota negación de su operador booleano.

Operadores Multiplicadores:

Operador	Operación	Tipo de operandos	Tipo de resultado
*	Multiplicación	Ambos enteros	Entero
DIV	División con truncamiento	Ambos enteros	Entero
AND	"Y" lógico	Ambos Boolean	Boolean

Operadores sumadores:

Operador	Operación	Tipo de operandos	Tipo de resultado
+	Suma	Ambos enteros	Entero
-	Resta	Ambos enteros	Entero
OR	"o" lógico	Ambos Boolean	Boolean

Cuando el signo "-" se utiliza con un operando (entero), denota el inverso de éste.

Operadores relacionales.

Operador	Tipo de operandos	Tipo de resultado
=, <>		
<>	Ambos enteros	Boolean
<=, >=		

2.2.9 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 Programación del analizador léxico.

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> ::= <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 [Aho-Ullman, 1986], 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 de la siguiente forma:

```
CASE car OF
 'A'..'Z' : Explora identificador ó 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 Interfaz entre el analizador léxico y el parser.

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

¿Qué es un símbolo? Nuestra definición de los símbolos, como las palabras reservadas que integran un programa, sugiere que un símbolo sea 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,(...));
```

La interfaz 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 dato:

```
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, llama al siguiente procedimiento:

```
PROCEDURE Error(tipo_error : t_error);
VAR
  texto : STRING[35];
BEGIN
                ...
  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;
            ...
END;
```

Procedimiento 2.1 Manejo de mensajes de error.

Al principio de cada línea del programa fuente, las fases de análisis llaman al procedimiento Nueva_lin para registrar un número de línea e indicar que aún no se han encontrado errores en ésta:

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

Al indicar un error, se generará un mensaje y la línea se marca como incorrecta para suprimir mensajes adicionales relacionados a esa línea (procedimiento 2.1).

2.3.3 Búsqueda.

EL analizador léxico construye una tabla de todas las palabras utilizadas 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 booleano que indica si ésta es un identificador o palabra reservada, y 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, se inserta y se describe como un identificador con un nuevo índice de identificador.

2.3.4 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 utilizar 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 utiliza 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.

```
FUNCTION cve_hash(texto:cad; long:INTEGER) : INTEGER;
CONST
  w = 32641;
  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;
```

Procedimiento 2.2 Cálculo de la clave hash.

La función hash debe elegirse con cuidado para reducir las colisiones en la tabla de símbolos. Se ha demostrado[Brinch Hansen, 1985], que el procedimiento 2.2 trabaja bien. Suma los caracteres individuales de una palabra, divide la suma entre la longitud de la palabra y utiliza el residuo como clave hash. Es conveniente que la longitud de la tabla sea un número primo.

2.3.5 Tabla de símbolos.

Antes de implementar la tabla de símbolos, 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 [Aho-Ullman, 1986]. Cada registro de una palabra define la longitud de la palabra y el índice de su último carácter en la tabla de lexemas.

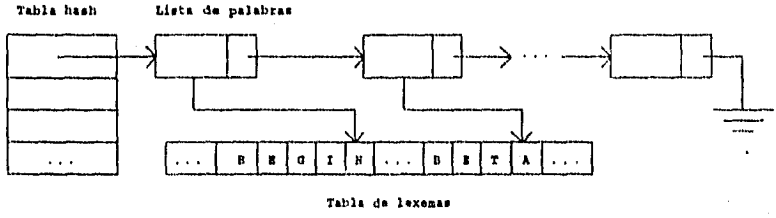


Fig. 2.3 Tabla de símbolos.

La tabla hash es un arreglo de apuntadores:

```
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 vacía, el apuntador correspondiente de la tabla hash tiene el valor nil; en caso contrario, éste apunta a la primera palabra registrada en la lista.

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 procedimiento 2.3 define la inserción de una nueva palabra en la tabla de símbolos.

```
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;
  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;
```

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

Las palabras reservadas y los identificadores estándar se insertan por medio del procedimiento 2.4.

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

Procedimiento 2.4 Inserta palabras clave e identificadores.

Este procedimiento se llama de la siguiente forma:

```
Instala(FALSE,'AND',    3,ORD(AND1));
                  ...
Instala(FALSE,'WHILE', 5,ORD(WHILE1));
Instala(TRUE,'INTEGER',7,ORD(INTEGER1));
                  ...
```

El analizador léxico utiliza el procedimiento 2.5 para buscar una palabra particular en la tabla de símbolos. Utiliza la clave hash de ésta para seleccionar la lista en la cual buscarla. Si se encuentra, se recuperará de la tabla el valor del símbolo; en caso contrario, la insertará 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;
```

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

El procedimiento 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;
```

Procedimiento 2.6 Verifica si una palabra dada ya existe en la tabla de símbolos.

2.3.6 Prueba.

Cada fase del compilador se prueba permitiéndole compilar pequeños programas construidos específicamente.

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 programa Prueba1 (ver apéndice A) se construyó especialmente para utilizarlo 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 utilizados: 5
```

CAPITULO 3. ANALISIS SINTACTICO.

Introducción.

Analizar sintácticamente una cadena de tokens es encontrar para ella un árbol sintáctico que tenga como raíz el símbolo inicial de la gramática 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 y se indica el error.

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 para este tipo de análisis: reducción desplazamiento, precedencia simple, precedencia de operadores y lenguaje de producciones. El análisis sintáctico ascendente puede manejar una clase mayor de gramáticas [Aho-Ullman, 1986]. En sistemas UNIX se cuenta con herramientas basadas en el método de análisis ascendente para generar un traductor a partir de la gramática del lenguaje.

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 el descenso recursivo y parsing LL(1) con tabla.

Debido a que en esta tesis nuestro enfoque es la construcción manual de un compilador, desarrollaremos un analizador sintáctico basado en el método de descenso recursivo para analizar programas escritos en Mini-Pascal.

3.1 Descenso recursivo.

Una de las formas de reconocimiento sintáctico más utilizadas hoy en dia es la de descenso recursivo. Con este nombre se quiere indicar que se realiza una construcción descendente del árbol sintáctico como ya se indicó.

Comenzaremos definiendo la sintaxis del lenguaje Mini-Pascal, clasificada en orden descendente recursivo [Welsh-McKeag, 1980]. Los siguientes metasímbolos pertenecen al formalismo BNF.

::=	Significa "se define como".
{}	Significa que su contenido se puede repetir cero o más veces.
<>	Indica que su contenido es una construcción sintáctica BNF.
|	Denota la posible selección entre una u otra alternativa.
[]	Representa una construcción opcional.

El resto de los símbolos forma parte parte del lenguaje. Cada construcción sintáctica aparece entre paréntesis angulares, 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 vacia>

<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 vacia>
```

```
<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 | Digito}
```

3.2 Construcción del analizador sintáctico para Mini-Pascal.

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

1) Se 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.

3.2.1 Algoritmos.

En el Capítulo anterior, traducimos las reglas sintácticas para símbolos en fragmentos equivalentes de código. Podemos utilizar 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;
```

Procedimiento 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;
```

Procedimiento 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 transformar la regla sintáctica de cada construcción sintáctica en un procedimiento, cuya acción sea analizar la secuencia de símbolos de entrada, y verificar que sean de la forma permitida.

Podemos ilustrar nuestro proceso de transformación 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 $\alpha|\beta|\ldots|\delta$, la acción requerida es alguna selección entre las acciones apropiadas a cada alternativa:

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

¿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|...|δ) ---> case símbolo of
                               iniciadores(α) : T(α)
                               iniciadores(β) : T(β)
                                   ...
                               iniciadores(δ) : T(δ)
                           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|...|δ|' <cadena vacia>) ---> case símbolo 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.

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. Esto lo expresamos en la siguiente regla.

Regla 5. T({α}) ---> while símbolo 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 en cuestión 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 en el mismo nivel. A estos símbolos mayores les llamaremos símbolos de parada y entre ellos están los siguientes:

```
símbolos de parada := [CONST1, TYPE1, VAR1, PROCEDURE1,
                       BEGIN1, IF1, WHILE1, PUNTO]
```

El parser utiliza el procedimiento 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 (símbolo in stop) DO
        Explora_simbolo
END;
```

Procedimiento 3.2.3 Recuperación de errores de sintaxis.

La idea de saltar símbolos hasta que el parser alcance un símbolo de parada es buena, pero si utilizamos pocos 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), 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 utilizar 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 utilizaremos 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 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, para esto utiliza el procedimiento 3.2.4.

```
PROCEDURE Verifica_sintaxis(stop : simbolos);
BEGIN
  if not (simbolo in stop) then error_de_sintaxis(stop)
END;
```

Procedimiento 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:

```
Procedure Error_de_sintaxis(stop : simbolos);
begin
  Error(error_sintactico);
  while not(simbolo in stop) do
        Explora_simbolo;
end;
```

Procedimiento 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 utilizando como parámetro a los símbolos de parada esta sentencia:

```
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.

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).

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 utilizados: 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 utilizados: 11
```

CAPITULO 4. ANALISIS DE ALCANCE.

Introducción.

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

4.1 Bloques.

Un programa en Mini-Pascal utiliza identificadores para referirse a constantes, tipos de datos, campos, variables y procedimientos. Estas entidades forman los objetos del programa.

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

Cualquier otro objeto que se utilice en un programa debe especificarse 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.

Un programa combina proposiciones y definiciones relacionadas en unidades sintácticas llamadas bloques. Hay tres clases de bloques: 1) El bloque estándar, en el cual se definen los objetos estándar, 2) bloque de programa principal y 3) procedimientos.

4.2 Reglas de alcance.

Regla 1. Todos los objetos definidos en el mismo bloque deben tener nombres diferentes.

Regla 2. Una constante, tipo o variable definida en un bloque se conoce desde el final de su definición hasta el final del bloque. Un procedimiento definido en un bloque B se conoce desde el principio hasta el final de éste.

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 se desconoce en el alcance del segundo objeto.

4.3 Método de compilación.

Utilizaremos el siguiente programa de ejemplo para explicar cómo realiza el compilador el análisis de alcance [Brinch Hansen, 1985].

```
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 utiliza 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 durante la compilación del programa de ejemplo.

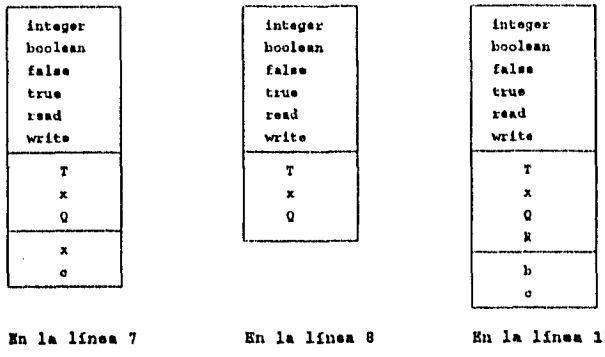


Fig. 4.1 La pila durante la compilación de un programa.

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 están representados mediante una lista encadenada de registros. Cada registro describe un objeto mediante el índice del identificador y un apuntador.

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

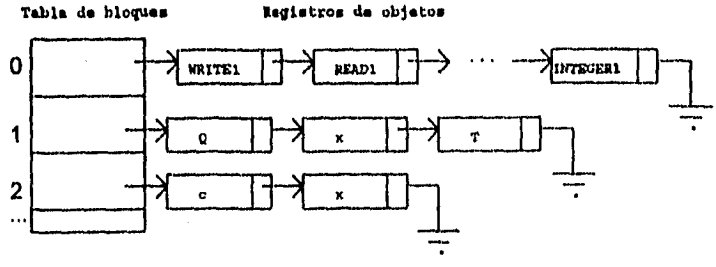


Fig. 4.2 Pila de identificadores representados por listas encadenadas.

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 utiliza 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 apuntador 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 Procedimientos.

Necesitamos un procedimiento 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;
```

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

El parser utiliza el procedimiento 4.2 para definir un nuevo objeto. Si el bloque actual ya tiene definido otro objeto con el mismo nombre, el identificador se reporta como duplicado. En caso contrario, se enlaza al otro objeto definido en el mismo bloque.

```
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;
```

Procedimiento 4.2 Definición de un objeto nuevo en un bloque.

Cuando se utiliza un identificador para referirse a un objeto, el compilador primero 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 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;
```

Proc. 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;
```

Procedimiento 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 (procedimiento 4.5).

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

Procedimiento 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 procedimiento 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;
```

Procedimiento 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 parser 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), incluye todos estos casos.

La salida del parser para este programa es la siguiente:

```
1   { Mini-Pascal. Prueba4: 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 utilizados: 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). A continuación se muestra la salida proporcionada por el analizador sintáctico para este programa.

```
1    { Mini-Pascal. Prueba5: 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 utilizados: 9
```

CAPITULO 5. ANALISIS DE TIPOS.

Introducción.

Este capítulo explica la forma en la que el compilador utiliza 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. Sin embargo, para efectuar el análisis de tipos, el compilador debe ser capaz de distinguir diferentes clases de objetos. Utilizaremos el siguiente tipo de dato para clasificar estos 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 ampliar 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 acceder sus registros a través de apuntadores.

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 apuntador a este registro. El parser utiliza el apuntador para llamar la parte variante del registro con información adicional referente al objeto.

Cuando 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, crea un registro de una clase ficticia, llamada indefinida. En ambos casos, el procedimiento devuelve un apuntador al registro seleccionado, el cual utiliza el parser para acceder a la información disponible referente al objeto.

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

5.2 Tipos estándar.

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 utiliza 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 apuntador al registro correspondiente. Estos apuntadores 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.

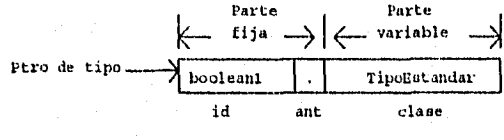


Fig. 5.1 Representación de datos del tipo Boolean.

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.

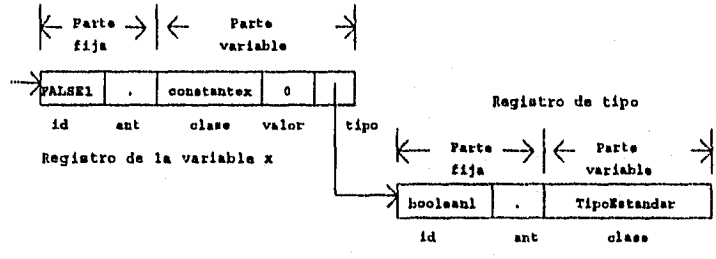


Fig. 5.2 Representación de datos de la constante false.

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 la proporciona el procedimiento 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;
```

Procedimiento 5.1 Análisis de una constante.

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

(1) Cuando el parser encuentre un identificador que no se refiere a una constante, lo reportará 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.

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

Procedimiento 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.

El siguiente procedimiento define un parámetro por valor u, un parámetro por referencia v, y dos variables locales x, y. Estas son las diferentes clases de variables que se utilizan en Mini-Pascal:

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

Durante la compilación, cada variable se describe 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.

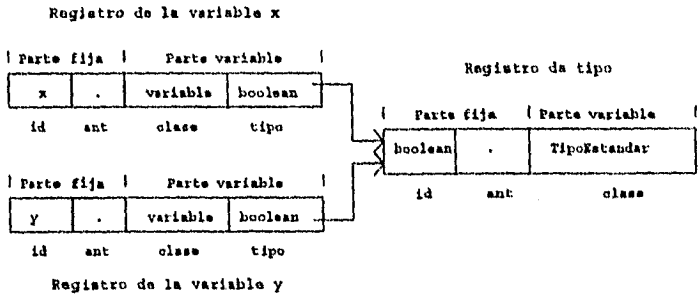


Fig. 5.3 Representación de datos de dos variables.

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

```
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;
```

Procedimiento 5.3 Análisis de tipos para variables.

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

El procedimiento 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 se maneja por una sola llamada separada al procedimiento. La última de éstas ocurre cuando el procedimiento alcanza el identificador de tipo y obtiene un apuntador al registro del tipo correspondiente. Este apuntador se asigna 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 apuntador de tipo.

```
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;
```

Procedimiento 5.4 Análisis de un identificador de tipo.

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

El compilador utiliza 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 procedimiento 5.5.

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

Proc. 5.5 Análisis de tipos para la definición de variables.

5.5 Arreglos.

La parte variante de un tipo arreglo 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 registros que describen los tipos entero y boolean.

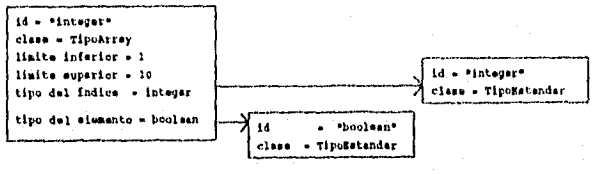


Figura 5.4 Representación de datos de un tipo arreglo.

Una definición 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 procedimiento 5.6, el cual analiza la definición de un tipo arreglo y construye el registro correspondiente.

```
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;
```

Procedimiento 5.6 Análisis de la definición de un tipo arreglo.

El parser utiliza el procedimiento 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;
```

Procedimiento 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 apuntador 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;
```

Procedimiento 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 realizarse 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 se denota como x.f y se conoce sólo en el alcance de x.

Un tipo registro está descrito por un registro de objeto de la siguiente clase:

```
Reg_obj = RECORD
            id          : INTEGER;
            ant         : Ptro;
            CASE clase : clas OF
                   ...

                TipoRecord : (ult_campo : Ptro);

                   ...
          END;
```

La parte variante apunta a otro registro que describe el último campo del tipo registro. Los campos están descritos por otra clase de registro de objeto:

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

La parte fija enlaza un campo a los campos anteriores del mismo tipo. La parte variante apunta al tipo del campo.

El parser debe verificar que los campos de un tipo registro tengan identificadores diferentes y debe enlazarlos en forma separada. Los procedimientos de análisis de alcance realizarán esto automáticamente si tratamos un nuevo registro como un bloque (ver procedimiento 5.9).

```
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;
```

Procedimiento 5.9 Análisis de tipos para registros.

La lista de campos se analiza con el procedimiento 5.10.

```
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;
```

Procedimiento 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 apuntador a su tipo R (procedimiento 5.8). Después, ejecuta el procedimiento 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
       ...
  END;
END;
```

Procedimiento 5.11 Análisis de tipos para un selector de campo.

5.7 Expresiones.

El tipo de una expresión está determinado por cuatro procedimientos: Expresión, Expresión Simple, Término y Factor. La figura 5.5 muestra el orden en el que se pueden llamar.

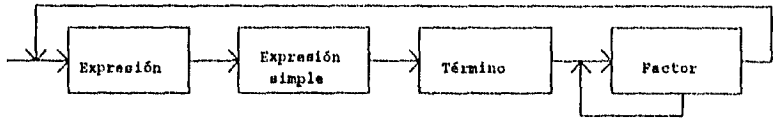


Fig. 5.5 Diagrama de dependencia para expresiones.

Observemos primero los factores (procedimiento 5.12). Este procedimiento ilustra la naturaleza recursiva de la compilación.

```
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:  { ERROR }
  END  {CASE}
END;     {Factor}
```

Procedimiento 5.12 Análisis de tipos para factores.

Un término se analiza mediante el procedimiento 5.13 utilizando el siguiente conjunto de símbolos:

simbolos_multiplicadores = [AND1, MULT, DIV1, MOD1]

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

```
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;
```

Procedimiento 5.13 Análisis de tipos para términos.

Los procedimientos para el análisis de expresiones son similares.

5.8 Proposiciones.

El procedimiento 5.14 se utiliza 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, debe ser una proposición de asignación. Si empieza con un identificador de procedimiento, debe ser una proposición procedure.

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

```
PROCEDURE Prop (Stop: simbolos);
VAR
  Objeto : Ptro;
BEGIN
  CASE simbolo OF
       IDENT : BEGIN
                 ...
                 Acepta(IDENT,Stop)
               END
               END;
       IF1   : Prop_if(Stop);
       WHILE1: Prop_WHILE(Stop);
       BEGIN1: Prop_compuesta(Stop);
       ELSE    Verifica_sintaxis(Stop)
  END;
END;
```

Procedimiento 5.14 Análisis de tipos para una proposición.

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 procedimiento 5.15.

```
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;
```

Procedimiento 5.15 Análisis de tipos de una proposición 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;
```

Proc. 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.

El procedimiento 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;
```

Proc. 5.17 Creación de un registro para un procedimiento.

El análisis de una proposición procedure se muestra en el procedimiento 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.

```
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;
```

Procedimiento 5.18 Análisis de una proposición procedure.

La lista de parámetros actuales se analiza mediante el procedimiento 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 se analiza mediante 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;
```

Proc. 5.19 Análisis de la lista de parámetros actuales.

Los registros de los procedimientos estándar se crean 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 se analizan por medio del procedimiento 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
     Expresion(Tipox,Stop2);
  Verifica_tipos(Tipox,TipoEntero);
  Acepta(PARENT_DER,Stop2)
END;
```

Proc. 5.20 Análisis de tipos para READ y WRITE.

5.10 Prueba.

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

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 (procedimiento 5.12). Para probar sistemáticamente las proposiciones de este procedimiento, debemos usar factores de las siguientes clases:

a) Un número.
b) Un identificador de constante.
c) Una variable.
d) Una expresión entre paréntesis.
f) 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 sus partes. Esto nos permite desarrollarlo gradualmente, es decir, en fases. En cada fase, podremos ver que los procedimientos desarrollados en etapas anteriores funcionan, de tal forma que podamos concentrarnos en probar la parte más recientemente desarrollada.

A continuación se muestran los resultados de estos tres programas de prueba.

```
1   { Mini-Pascal. Prueba6: 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
```

```
1   { Mini-Pascal. Prueba7: 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
```

```
1   { Mini-Pascal. Prueba8: 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
```

CAPITULO 6. GENERACION DE CODIGO.

Introducción.

Este capítulo 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.

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.

Esta computadora es ideal en el siguiente sentido:

(1) Sus instrucciones 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. Por lo tanto, el generador de código es una extensión del parser.

6.2 La pila.

La memoria de la computadora Mini-Pascal es un arreglo de enteros.

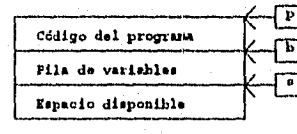


Fig. 6.1 La memoria y sus registros.

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 utiliza 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 acceder a 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
  memoria = ARRAY[min..max] OF INTEGER;
VAR
  st     : memoria;
  p,b,s  : INTEGER;
```

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

Las variables de un bloque se guardan en el segmento de pila conocido como registro de activacion [Brinch Hansen, 1985; Fischer-LeBlanc, 1991], el cual consiste de cuatro partes: La parte de parámetros, la parte de contexto, la parte de variables y la parte temporal (fig. 6.2).

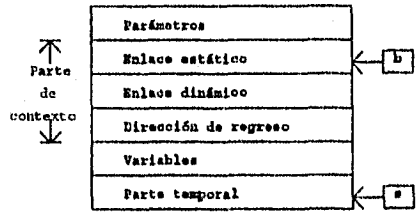


Fig. 6.2 Estructura de un registro de activación.

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.

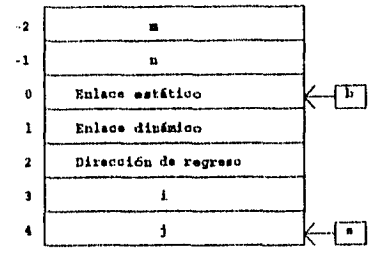


Fig. 6.3 Ejemplo de un registro de activación.

Si un procedimiento se activa 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 accedido 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.

```
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 después de que se han activado 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 termina un procedimiento, 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.

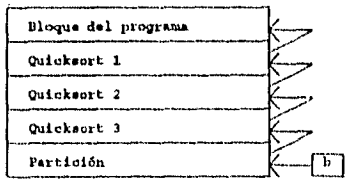


Fig. 6.4 La cadena dinámica.

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.

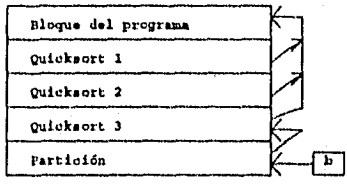


Fig. 6.5 La cadena estática.

En general, cada activación de un bloque puede tomar lugar en un contexto diferente. Por tanto, 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.

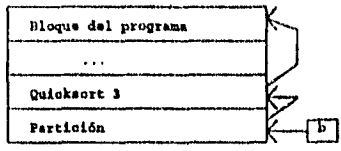
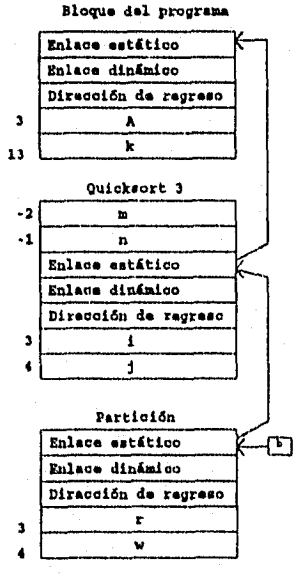


Fig. 6.6 El contexto actual.

6.3 Acceso a variables.

La figura 6.7 es una imagen más detallada del contexto mostrado en la figura 6.6. Para acceder a 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 identificarla 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. Aquí hay algunos ejemplos de la forma en la que son identificadas las variables dentro del procedimiento partición:



variable	nivel	desplazamiento
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.

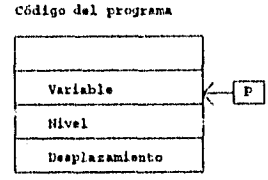


Fig. 6.8 Instrucción Mini-Pascal.

En el procedimiento Partición, las referencias a las variables A, m y w se compilan en las siguientes 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) Incrementa en uno al registro de pila, s, para crear una nueva localidad temporal en el tope de la pila.

(2) Busca la dirección base de la variable, la cual encuentra dos niveles abajo de la cadena estática.

(3) Calcula la dirección absoluta de la variable sumando la dirección base y el desplazamiento.

(4) Almacena la dirección absoluta en la nueva localidad temporal.

(5) Incrementa en 3 al registro de programa, p, para que apunte a la siguiente instrucción.

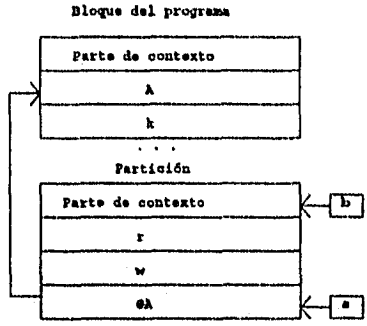


Fig. 6.9 Resultado de acceder a una variable.

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 utilizado para su propósito.

El procedimiento 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 utiliza 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.
```

Procedimiento 6.1 Instrucción Variable.

Las instrucciones "**Variable**" se utilizan sólo para acceder a parámetros por valor y variables locales. Los parámetros por variable se acceden en forma diferente. El programa siguiente incluye un parámetro por referencia 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).

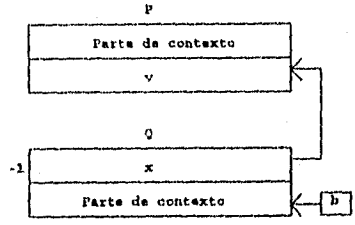


Fig. 6.10 Parámetro variable.

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 procedimiento 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;
```

Procedimiento 6.2 Instrucción ParamVar.

Como hemos visto, las diferentes clases de variables se acceden mediante distintos tipos de instrucciones. Podemos caracterizar todas las formas posibles mediante una gramática.

Para aplicar esta idea en la generación de código, debemos ver el código Mini-Pascal como un lenguaje, en el cual sus símbolos son instrucciones de computadora, tales como **Variable** y **ParamVar**. La posible secuencia de instrucciones debe estar definida por reglas sintácticas escritas en BNF. Estas reglas sintácticas se llamarán reglas código [Brinch Hansen, 1985].

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 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) Para acceder a una variable se requiere 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 acceder a los campos de un registro y a los elementos de un arreglo se utiliza un procedimiento similar al que se describió para acceder a variables.

6.4 Sintaxis del código Mini-Pascal.

La siguiente es una lista de las reglas código, las cuales corresponden a las reglas sintácticas de Mini-Pascal (sección 3.1).

```
<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 vacia>
<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 Ejecución 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 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 cual se ejecuta en los siguientes pasos:

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

(2) La instrucción Do elimina de la pila este valor. 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

6.5 Ejecución 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 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 cual se ejecuta en los siguientes pasos:

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

(2) La instrucción **Do** elimina de la pila este valor. 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

Puesto que la computadora Mini-Pascal conoce la dirección y la longitud de cada instrucción, podría generar instrucciones de salto con direcciones absolutas. Pero en la mayoría de los sistemas, la dirección de un programa se desconoce hasta que el sistema operativo lo cargue. Diseñaremos el compilador Mini-Pascal para producir 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 con la instrucción **Goto**:

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

Procedimiento 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 ,...,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 es apropiada para una computadora con pila y se sigue un procedimiento similar al de la ejecución de proposiciones.

6.6 Códigos de operación.

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 = (ADD2,AND2,ASIGNA2,CONSTANTE2,DIV2,DO2,FINPROC2,
                              ...
            VALOR2, VARIABLE2, PARAM_VAR2, READ2, WRITE2,
            DEFINE_DIREC2, DEFINE_ARG2,...);
```

El parser emite una instrucción de la forma:

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

por medio de la proposición:

```
Emite3(VARIABLE2, Nivel, Desp);
```

6.7 Direccionamiento de variables.

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

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

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

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

El procedimiento Def_de_variable utiliza esta información para calcular la longitud de la lista de variables (procedimiento 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;
```

Procedimiento 6.4 Direccionamiento de variables.

La longitud de un valor arreglo se obtiene multiplicando el número de elementos de éste 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 6.5 analiza la parte de definición de las variables i y j.

```
PROCEDURE Parte_def_variable(varlong_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;
```

Procedimiento 6.5 Análisis de la parte de definición de variables.

Al final de la parte de definición de variables, el parser llama al procedimiento 6.6 para reconocerlas y asignarles los desplazamientos mostrados en la figura 6.3.

```
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;
```

Procedimiento 6.6 Asignación de desplazamientos a variables.

6.8 Código para expresiones.

Consideremos un bloque con dos variables locales:

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

y la expresión:

```
y*z div 5;
```

la cual 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 los procedimientos Expresión, ExpresiónSimple, Término y Factor, en este orden. Factor reconoce el identificador de variable y ejecuta las siguientes proposiciones:

```
AccesaVar(Tipox,Stop);
Emite2(Valor2, Long);
```

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, 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
       MULT     : Emitel(MULT2);
       DIV1     : Emitel(DIV2);
       MOD1     : Emitel(MOD2);
  ELSE { operador = AND1 }
       Error_de_tipo(Tipox);
  END; { CASE }
  Pop(1)
END
```

El efecto de esto es generar el operador de multiplicación.

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

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

y emite una instrucción Constante(5).

6) Finalmente, el procedimiento Término genera el operador de división.

```
PROCEDURE Termino(VAR tipox : Ptro; Stop : simbolos);
BEGIN {...}
  Factor(Tipox,Stop);
  WHILE simbolo IN simb_multiplicadores DO
  BEGIN {...}
    Factor(Tipo2,Stop);
    IF Tipox = TipoEntero THEN
    BEGIN
      Verifica_tipos(Tipox,Tipo2);
      CASE operador OF
           MULT     : Emitel(MULT2);
           DIV1     : Emitel(DIV2);
           MOD1     : Emitel(MOD2);
           ...
      END; { CASE }
    END
 END;
```

Procedimiento 6.7 Generación de código para términos.

Durante el análisis de alcance, el parser mantiene una pila de registros de bloques (sección 4.4). Ahora ampliaremos el registro de un bloque 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 utilizan en el bloque actual se almacena en el campo num_max_loc.

En el momento que el parser emite una instrucción para una variable, 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 Código 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 procedimiento 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;
```

Procedimiento 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 se eliminan 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, no es posible generar en este punto 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 código.

(3) Posteriormente, el ensamblador explora nuevamente el mismo código y utiliza la tabla para producir el 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 utiliza una variable para contar el número de etiquetas creadas hasta el momento.

Cuando el parser requiere una nueva etiqueta, ejecuta el procedimiento 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;
```

Procedimiento 6.9 Creación de una etiqueta.

El análisis de una proposición **while** resulta ahora muy sencillo (procedimiento 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;
```

Proc. 6.10 Generación de código para la proposición **while**.

El ensamblador guarda la dirección de la instrucción actual en una variable denominada dirección, la cual se incrementa después de la generación de una instrucción (procedimiento 6.11).

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

Procedimiento 6.11 Incrementa la dirección.

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 (procedimiento 6.12).

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

Procedimiento 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 Código 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.

(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. El ensamblador resuelve estas referencias hacia adelante. El parser asigna una etiqueta númerica a cada uno de los argumentos anteriores y genera la instrucción procedure con estas etiquetas.

Cuando el parser alcanza un punto en el cual se conoce el valor del argumento, 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 capítulo 5 discutimos el análisis de un procedimiento (procedimiento 5.17). El procedimiento 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;
  Emite2(FINPROC2, Long_param);
  Final_de_bloque
END;
```

Procedimiento 6.13 Generación de código para procedimientos.

Cuando el ensamblador lee una instrucción **LlamadaProc**, reemplaza la etiqueta del procedimiento por la dirección del código procedure, relativa a la dirección actual (procedimiento 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;
```

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

6.11 Optimización de código.

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

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

produce 13 palabras de código:

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

Este código puede reducirse a 8 palabras utilizando algunas instrucciones extra:

```
* VariableLocal(13)
* ValorLocal(13)
  Constante(1)
  Suma
* AsignaciónSimple
```

Las instrucciones extra están marcadas con un * y representan casos especiales que ocurren con frecuencia en programas en Pascal.

Estudios experimentales [Brinch Hansen, 1985] muestran que, en general, un bloque utiliza sus propias variables locales y (en menor grado) variables globales definidas en el bloque inmediato en el que está contenido. Esto es, un bloque tiende a acceder variables en los niveles (relativos) 0 y 1. Las instrucciones VariableLocal y ValorLocal toman ventaja de esto.

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

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

a la cual llamaremos regla de optimización.

Cuando la computadora ejecuta una instrucción VariableLocal, coloca la dirección de una variable local en la pila (procedimiento 6.15).

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

Procedimiento 6.15 Instrucción VariableLocal.

Una instrucción ValorLocal reemplaza dos instrucciones:

Variable(0,Desp) Valor(1)

Podemos expresar esto mediante otra regla de optimización:

ValorLocal(Desp) = VariableLocal(Desp) Valor(1)

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

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.

Visto lo anterior, podemos decir que la generación de código se realiza en dos pasos:

(1) Durante la primera 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.

Veamos como realiza el ensamblador la optimización de código. Primero, debemos ampliar el conjunto de instrucciones con las instrucciones extra:

```
TYPE
  cod_op = (ADD2...LLAMADA_GLOBAL2,VALOR_GLOBAL2,VARIABLE_GLOBAL2,
            VALOR_LOCAL2, VARIABLE_LOCAL2, ASIGNACION_SIMPLE2,
            VALOR_SIMPLE2);
```

El ensamblador utiliza el procedimiento 6.16 para leer la siguiente instrucción estándar y almacenarla en un conjunto de variables.

```
VAR
  op : cod_op;
  arg1, arg2, arg3, arg4 : INTEGER;

  PROCEDURE Prop_sig;
  BEGIN
    READ(temp1,simb);
    op := Busca_op(simb);
    IF op IN sin_args THEN
       { saltarla }
    ELSE
       IF op IN un_arg THEN
          READ(temp1,arg1)
       ELSE
          IF op IN dos_args THEN
          BEGIN
            READ(temp1,arg1);
            READ(temp1,arg2)
          END
          ELSE { op IN cuatro_args }
          BEGIN
            READ(temp1,arg1);
            READ(temp1,arg2);
            READ(temp1,arg3);
            READ(temp1,arg4)
          END;
END;
```

Procedimiento 6.16 Lectura de la siguiente instrucción.

Este procedimiento utiliza conjuntos para definir el número de argumentos que le siguen a un código de operación:

```
VAR
  sin_args, un_arg, dos_args, cuatro_args : SET OF cod_op;
```

Después de leer una instrucción Variable, el ensamblador llama al procedimiento del mismo nombre:

```
PROCEDURE Variable(Nivel, Desp : INTEGER);
BEGIN
  Prop_sig;
  WHILE Optimiza(op = CAMPO2) DO
  BEGIN
    Desp :=  Desp + arg1;
    Prop_sig
  END;
  IF Optimiza(Nivel = 0) THEN
     IF (op = VALOR2) AND (arg1 = 1) THEN
     BEGIN
       Emite2(VALOR_LOCAL2, Desp);
       Prop_sig
     END
     ELSE
       Emite2(VARIABLE_LOCAL2,Desp)
  ELSE
     IF Optimiza(Nivel = 1) THEN
        IF (op = VALOR2) AND (arg1 = 1) THEN
        BEGIN
          Emite2(VALOR_GLOBAL2, Desp);
          Prop_sig
        END
        ELSE
          Emite2(VARIABLE_GLOBAL2, Desp)
     ELSE
        Emite3(VARIABLE2,Nivel,Desp)
END;
```

Tan pronto como ha reconocido la instrucción Variable y haber llamado al procedimiento anterior, lee la siguiente instrucción para ver si la puede combinar esta primera.

6.12 Prueba.

Debemos probar el compilador cuando genera código estándar y cuando produce código optimizado. Para simplificar esta prueba dual, es conveniente hacer condicional la optimización de código. En lugar de escribir una condición de optimización como:

```
IF Nivel = 0 THEN ...
```

ésta se expresa como un parámetro de una función:

```
IF Optimiza( Nivel = 0 ) THEN ...
```

Si el código debe ser optimizado, la función devuelve el valor booleano de la condición de optimización:

```
FUNCTION Optimiza(CasoEspecial: BOOLEAN) : BOOLEAN;
BEGIN
   Optimiza := CasoEspecial
END;
```

La optimización puede suprimirse haciendo que la función siempre devuelva el valor falso.

La generación de código se realizó compilando el programa Prueba10 (ver apéndice A), con y sin optimización. El resultado del programa es la siguiente secuencia de valores enteros:

```
0 1 2 3 4 5 6 7 -8 9 10 11 12 13
1 0 1 1 0 0 1 1 0 14 15 16 17
```

CONCLUSIONES.

En esta tesis se presentaron algunas de las técnicas y algoritmos más importantes para la construcción manual de compiladores y se mostró como se pueden programar en un lenguaje de alto nivel.

No obstante que, especialmente en sistemas UNIX, existen herramientas para generar compiladores, éstas algunas veces no están disponbles para personas que no cuentan con este equipo; desconocen su uso o no manejan el lenguaje en el cual se generan los programas. Es por esto que se consideran de gran utilidad los resultados obtenidos en este trabajo para que las personas que quieran escribir parte de un compilador ó alguna otra aplicación similar utilicen las técnicas expuestas.

La implementación de los algoritmos se realizó principalmente en el lenguaje de programación Pascal, debido a que es uno de los lenguajes que está más al alcance de todos. Al realizar la implementación en este lenguaje, no encontramos mayor dificultad, dada su facilidad para manejar las estructuras de datos que utilizamos.

Un siguiente trabajo sería definir una biblioteca de clases para hacer la implementación en un lenguaje orientado a objetos, como por ejemplo C++, puesto que los elementos del lenguaje que utilizamos como modelo fueron considerados como tipos de datos abstractos que se implementaron con registros variantes de este lenguaje, logrando con esto que su uso sea más general. La conversión al modelo de objetos no es difícil, puesto que cada tipo de dato abstracto se puede mapear a un objeto de una jerarquía de clases ya definida.

Por su contenido, este trabajo podrá 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.

Por lo anterior, se lograron los objetivos señalados en título de esta obra "DISEÑO E IMPLANTACION DE HERRAMIENTAS DE SOFTWARE PARA COMPILADORES".

APENDICE A. 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 Test3;
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. Prueba4: 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. Prueba5: 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, z : integer;
begin
  x := a;
  y := a
end.

{ Mini-Pascal. Prueba6: 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 { vacio }
end.
```

```
{ Mini-Pascal. Prueba7: 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. Prueba8: 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.
```

```
( Mini-Pascal. Prueba9: Programa de ejemplo (capitulo 6) )
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
        r := A[(m+n) div 2];
        i := m;
        j := n;
        while i <= j do
        begin
           while A[i] < r do
                 i := i + 1;
           while r < A[j] do
                 j := j -1;
           if i <= j then
           begin
              w    := A[i];
              A[i] := A[j];
              A[j] := w;
              i := i + 1;
              j := j - 1;
           end
        end
      end;
   begin
      if m < n then
      begin
         Particion;
         Quicksort(m,j);
         Quicksort(i,n);
      end
   end;
begin
   k := 1;
   while k <= max do
   begin
     Read(A[k]);
     k := k + 1
   end;
   Quicksort(1,max);
   k := 1;
   while k <= max do
   begin
     write(A[k]);
     k := k + 1;
   end
end.
```

```
{ Mini-Pascal. Prueba10: Generación de código }
Program prueba10;
const
   dos = 2;
type
   S = array[1..10] of integer;
   T = record
          f, g : integer
       end;
var
   a : integer;
   b,c : S;
   d,e : T;

   procedure EscribeBoolean(x : boolean);
   begin
      if x then write(1)
      else write(0)
   end;

   procedure Eco;
   begin
      read(a);
      write(a)
   end;

   procedure R(u : integer; var v : integer);
   var
      x : integer;
   begin
      Eco;
      write(u);
      v := 3;
      write(a);
      x := 4;
      write(x)
   end;

   procedure Q;
   begin
      write(5)
   end;
```

```
begin
  write(0);
  R(dos,a);
  Q;
  b[10] := 6;
  c := b;
  write(c[10]);
  d.g := 7;
  e := d;
  write(e.g);
  write(-8);
  write(8+1);
  write(11-1);
  write(22 div 2);
  write(6*2);
  write(27 mod 14);
  EscribeBoolean(not false);
  EscribeBoolean(false and true);
  EscribeBoolean(false or true);
  EscribeBoolean(1 < 2);
  EscribeBoolean(1 = 2);
  EscribeBoolean(1 > 2);
  EscribeBoolean(1 <= 2);
  EscribeBoolean(1 <> 2);
  EscribeBoolean(1 >= 2);
  if true then write(14);
  if false then write(0)
  else write(15);
  a := 16;
  while a <= 17 do
  begin
     Write(a);
     a := a + 1
  end
end.
```

APENDICE B. Lista de los nombres de procedimientos principales.

Capitulo 2.

Nombre: cve_hash. (Pág. 24)
Parámetros: Cadena y longitud.
Objetivo: Calcular la clave hash de la cadena que recibe como parámetro.

Nombre: Inserta. (Pág. 26)
Parámetros: es_id, cadena, longitud, índice y num_cve.
Objetivo: Inserta una nueva palabra en la tabla de símbolos.

Nombre: Busca. (Pág. 27)
Parámetros: cadena, longitud, es_id, índice.
Objetivo: Buscar una palabra en la tabla de símbolos.

Nombre: Encont. (Pág. 28)
Parámetros: cadena, longitud y un apuntador a una palabra.
Objetivo: Verifica si una palabra dada ya existe en la tabla de símbolos.

Capitulo 3.

Nombre: Bloque. (Pág. 33)
Parámetros: ninguno
Objetivo: Análisis de un bloque.

Nombre: Error_de_sintaxis. (Pág. 36)
Parámetros: un conjunto de símbolos.
Objetivo: Recuperación de errores de sintaxis.

Nombre: Acepta. (Pág. 37)
Parámetros: simbolo_esperado, conjunto de símbolos.
Objetivo: Verifica el símbolo actual.

Capitulo 4.

Nombre: Busca_id. (Pág. 45)
Parámetros: id, num_nivel, encont, objeto.
Objetivo: Determina si un identificador ya ha sido definido en un bloque.

Nombre: Define. (Pág. 45)
Parámetros: clase y objeto.
Objetivo: Definicion de un objeto nuevo en un bloque.

Nombre: Encuentra. (Pág. 46)
Parámetros: id y objeto.
Objetivo: Verifica si existe un objeto en un bloque dado.

Nombre: Programa. (Pág. 47)
Parámetros: Un conjunto de símbolos.
Objetivo: Análisis de un programa.

Capítulo 5.

Nombre: Constante. (Pág. 53)
Parámetros: valor, tipo y un conjunto de símbolos.
Objetivo: Análisis de una constante.

Nombre: Lista_variables. (Pág. 55)
Parámetros: clase, ult_var, tipo y un conjunto de símbolos.
Objetivo: Análisis de tipos para variables.

Nombre: Id_de_tipo. (Pág. 56)
Parámetros: tipo y un conjunto de símbolos.
Objetivo: Análisis de un identificador de tipo.

Nombre: Tipo Arreglo. (Pág. 58)
Parámetros: id y un conjunto de símbolos.
Objetivo: Análisis de la definición de un tipo arreglo.

Nombre: Tipo Registro. (Pág. 61)
Parámetros: id y un conjunto de símbolos.
Objetivo: Análisis de tipos para registros.

Nombre: Lista _de_campos. (Pág. 62)
Parámetros: Un apuntador al tipo del factor y un conjunto de símbolos.
Objetivo: Análisis de tipos para campos.

Nombre: Factor. (Pág. 63)
Parámetros: Un apuntador al tipo del factor y un conjunto de símbolos.
Objetivo: Análisis de tipos para factores.

Nombre: Termino (Pág. 64)
Parámetros: Un apuntador al tipo del término y un conjunto de símbolos.
Objetivo: Análisis de tipos para términos.

Nombre: Prop. (Pág. 65)
Parámetros: Un conjunto de símbolos.
Objetivo: Análisis de tipos para una proposición.

Nombre: Prop_asignación. (Pág. 65)
Parámetros: Un conjunto de símbolos.
Objetivo: Análisis de tipos para una proposición de asignación.

Nombre: Prop_while. (Pág. 65)
Parámetros: Un conjunto de símbolos.
Objetivo: Análisis de tipos para una proposición while.

Nombre: Prop_procedure. (Pág. 67)
Parámetros: Un conjunto de símbolos.
Objetivo: Análisis de tipos para una proposición procedure.

Nombre: Prop_Es. (Pág. 68)
Parámetros: Un conjunto de símbolos.
Objetivo: Análisis de tipos para los procedimientos estándar READ y WRITE.

Capitulo 6.

Nombre: Variable. (Pág. 80)
Parámetros: Nivel y desplazamiento.
Objetivo: Definición de la instrucción "variable".

Nombre: Def_de_variable. (Pág. 88)
Parámetros: Un apuntador a la última variable, su longitud y un conjunto de símbolos.
Objetivo: Calcula la longitud de una lista de variables.

Nombre: Direc_var. (Pág. 88)
Parámetros: long_var y un apuntador a la última variable.
Objetivo: Asignación de desplazamientos a variables.

Nombre: Prop_asignación. (Pág. 91)
Parámetros: Un conjunto de símbolos.
Objetivo: Generación de código para proposiciones.

Nombre: Prop_while. (Pág. 94)
Parámetros: Un conjunto de símbolos.
Objetivo: Generación de código para la proposición while.

Nombre: Def_de_procedimiento. (Pág. 96)
Parámetros: Un conjunto de símbolos.
Objetivo: Generación de código para procedimientos.

Nombre: Prop_procedure. (Pág. 96)
Parámetros: Un conjunto de símbolos.
Objetivo: Generación de código para la llamada a un procedimiento.

BIBLIOGRAFÍA.

AHO, ALFRED V. AND J.D. ULLMAN.
Principles of Compiler Design.
Addison-Wesley, Reading, Mass. 1977.

AHO, ALFRED V., RAVHI SETHI AND J. D. ULLMAN.
Compilers Principles, Techniques, and Tools.
Addison-Wesley Publishing Company. 1986.

AHO, ALFRED V. AND J. D. ULLMAN.
The Theory of Parsing, Translation and Compiling,
Vol. I: Parsing.
Prentice-Hall, Englewood Cliffs, N. J. 1972.

AHO, ALFRED V. AND J.D. ULLMAN.
The Theory of Parsing, Translation and Compiling,
Vol. II: Compiling.
Prentice-Hall, Englewood Cliffs, N. J. 1973.

BACKHOUSE, R.C.
Syntax of Programming Languages: Theory and Practice.
Prentice, Englewood Cliffs, N.J. 1979.

BRINCH HANSEN, P.
On Pascal Compilers.
Prentice, Englewood Cliffs, N.J. 1985.

CHARLES N. FISCHER AND RICHARD J. LEBLANC JR.
Crafting a Compiler with C.
The Benjamin/Cummings Publishing Company Inc. 1992.

HOPCROFT, J. E. AND J. D. ULLMAN.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, Reading, Mass. 1979.

HOPCROFT, J. E. AND J. D. ULLMAN.
Formal Languages and Their Relation to Automata.
Addison-Wesley, Reading, Mass. 1969.

JEAN PAUL TREMBLAY AND PAUL SORENSON.
Compiler Writing.
Mc Graw-Hill. 1985.

JENSEN, K. AND N. WIRTH.
Pascal User Manual and Report.
Springer-Verlag, New York. 1975.

WELSH, J., AND MCKEAG, M.
Structured System Programming.
Prentice, Englewood Cliffs, N.J. 1980.