FACULTAD DE INGENIERIA
Universidad de Buenos Aires

75.16 - LENGUAJES DE PROGRAMACIÓN
Cátedra: LIC. MÓNICA KUHN
Trabajos Prácticos: M. ING. DIEGO CORSI

Entre los lenguajes de alto nivel y el código de máquina

Construcción de un compilador de PL/0 para Linux

Introducción

En este curso desarrollaremos un compilador para el lenguaje de programación PL/0, presentando una introducción general de la estructura y operación de los compiladores.

1. Comentarios generales sobre la teoría de compiladores

El diseño de programas para resolver problemas complejos es mucho más sencillo utilizando *lenguajes de alto nivel*, ya que se requieren menos conocimientos sobre la estructura interna del computador, aunque es obvio que éste sólo entiende el *código de máquina*. Por lo tanto, para que un computador pueda ejecutar programas escritos en un lenguaje de alto nivel, éstos deben ser traducidos a código de máquina. A este proceso se lo denomina *compilación*, y la herramienta que la lleva a cabo se llama *compilador*. Por ende, los compiladores son fundamentales para la computación, y su importancia se mantendrá en el futuro.

La entrada del compilador es el *código fuente*, es decir, el programa escrito en un lenguaje de alto nivel. El compilador analiza esta entrada y genera a su salida el *código objeto*. Existen distintas formas de código objeto, siendo una de las diferencias más destacables la que existe entre el código absoluto (el código de máquina con direcciones de memoria absolutas) y el código relocalizable (el código de máquina con desplazamientos de direcciones, y por lo tanto enlazable con otros módulos compilados por separado).

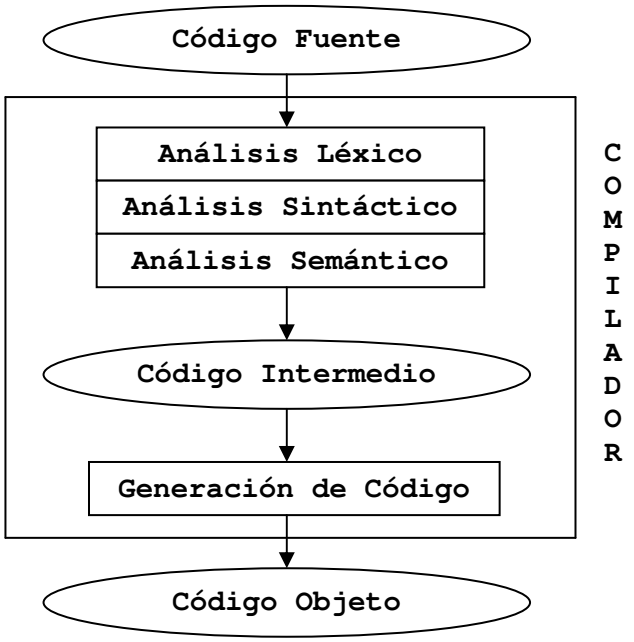
De acuerdo con los diferentes tipos de código y las diversas formas de funcionamiento, podemos distinguir entre los siguientes tipos de sistemas de compilación:

- **Ensamblador: Traduce programas escritos en lenguaje ensamblador a código de máquina. El lenguaje ensamblador se caracteriza por el uso de mnemónicos para representar instrucciones y direcciones de memoria.**
- **Compilador: Traduce programas escritos en un lenguaje de alto nivel a código intermedio o a código de máquina. El código intermedio puede ser, por ejemplo, un lenguaje ensamblador o alguna otra forma de representación intermedia.**

❒ **Intérprete: No genera código objeto, sino que analiza y ejecuta directamente cada sentencia del código fuente. Como no se genera código de máquina, en cierta forma los programas escritos para ser interpretados son independientes de la máquina.**

❒ **Preprocesador: Reemplaza macros, incluye archivos o extiende el lenguaje.**

En este curso sólo haremos hincapié en los compiladores, y estudiaremos las distintas fases del proceso de compilación, pero no nos detendremos en cuestiones marginales como los sistemas de edición y depuración que forman parte de todo ambiente de compilación moderno.



El análisis léxico es llevado a cabo por el analizador léxico (*lexical scanner* o simplemente *scanner*) y consiste en reconocer los componentes léxicos (símbolos del lenguaje) contenidos en el código fuente del programa a compilar, que ingresa como un flujo de caracteres.

El análisis sintáctico tiene como objetivo revisar si los símbolos detectados durante el análisis léxico aparecen en el orden correcto como para constituir un programa válido. El analizador sintáctico se conoce usualmente como *parser*.

El análisis semántico reconoce si las unidades gramaticales tienen sentido, detectando errores como inconsistencia de tipos u operaciones con objetos no declarados.

Finalmente, la generación de código es llevada a cabo por un módulo que usualmente es reemplazable, con lo cual se pueden obtener códigos objeto para distintas plataformas a partir de un mismo código intermedio. Además, hoy en día es común que compiladores de distintos lenguajes generen el mismo código intermedio, por lo que un programa ejecutable puede obtenerse enlazando módulos de código objeto obtenidos a partir de códigos fuente escritos en lenguajes distintos.

2. Estructura de los lenguajes

Los lenguajes se basan en un *vocabulario*. Sus elementos son comúnmente llamados *palabras*, pero en el estudio de los lenguajes formales se los denomina *símbolos*.

Es característico de los lenguajes que algunas secuencias de palabras sean reconocidas como *frases* correctas y otras no. Lo que determina si una secuencia de palabras es una frase correcta (o no) es la gramática, sintaxis o estructura del lenguaje. De hecho, definimos *sintaxis* como el conjunto de reglas que definen el conjunto de frases formalmente correctas.

Dado que la sintaxis provee a las frases de una estructura que nos sirve para reconocerles el significado, queda claro que la sintaxis y la *semántica* (el significado) están íntimamente conectados. Sin embargo, en un primer momento vamos a dedicarnos exclusivamente al estudio de la sintaxis.

Tomemos la frase "Martín duerme." La palabra "Martín" es el sujeto y la palabra "duerme" es el predicado. Esta frase pertenece a un lenguaje que puede, por ejemplo, estar definido por la siguiente sintaxis:

```
<frase> ::= <sujeto> <predicado>
<sujeto> ::= Martín | Julieta
<predicado> ::= duerme | juega
```

El significado de estas tres líneas es el siguiente:

1. **Una frase está formada por un sujeto seguido de un predicado.**
2. **El sujeto puede ser la palabra "Martín" o la palabra "Julieta"**
3. **El predicado puede ser la palabra "duerme" o la palabra "juega"**

Las frases correctas pueden derivarse a partir del *símbolo inicial* <frase> mediante la aplicación reiterada de *reglas de sustitución*.

La notación utilizada para escribir estas reglas se denomina BNF.

Las palabras *Martín*, *Julieta*, *duerme* y *juega* se denominan *símbolos terminales*. Una secuencia nula de símbolos se representa con ∈.

<frase>, <sujeto> y <predicado> son los *símbolos no terminales*. Las reglas se denominan *producciones*, ya que determinan cómo se pueden generar o *producir* frases correctas.

Los símbolos ::= y | se denominan metasímbolos de la notación BNF, y se pronuncian "puede sustituirse por" y "o", respectivamente. Aunque no forman parte de BNF (ya que son una extensión del mismo), también son consideradas metasímbolos las llaves { y } para encerrar símbolos que se repiten cero o más veces.

A veces, es posible simplificar la notación, utilizando letras minúsculas para los símbolos terminales y letras mayúsculas para los símbolos no terminales, en lugar de distinguirlos con < y >.

Ejemplo 1

S ::= AB
A ::= x|y
B ::= z|w

El lenguaje definido por esta sintaxis (que es equivalente a la sintaxis dada en la página anterior) es el compuesto por las cuatro frases *xz*, *yz*, *xw*, *yw*.

A diferencia del lenguaje anterior, el definido por la siguiente sintaxis está compuesto por un número infinito de frases:

Ejemplo 2

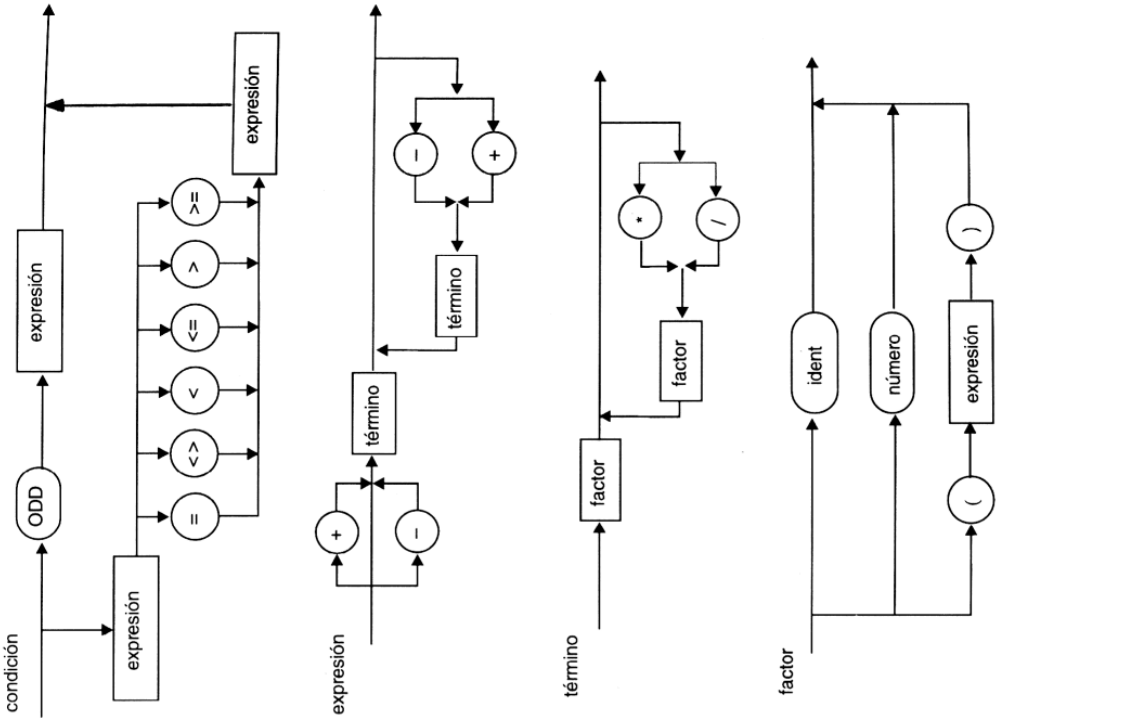
S ::= xA
A ::= z|yA

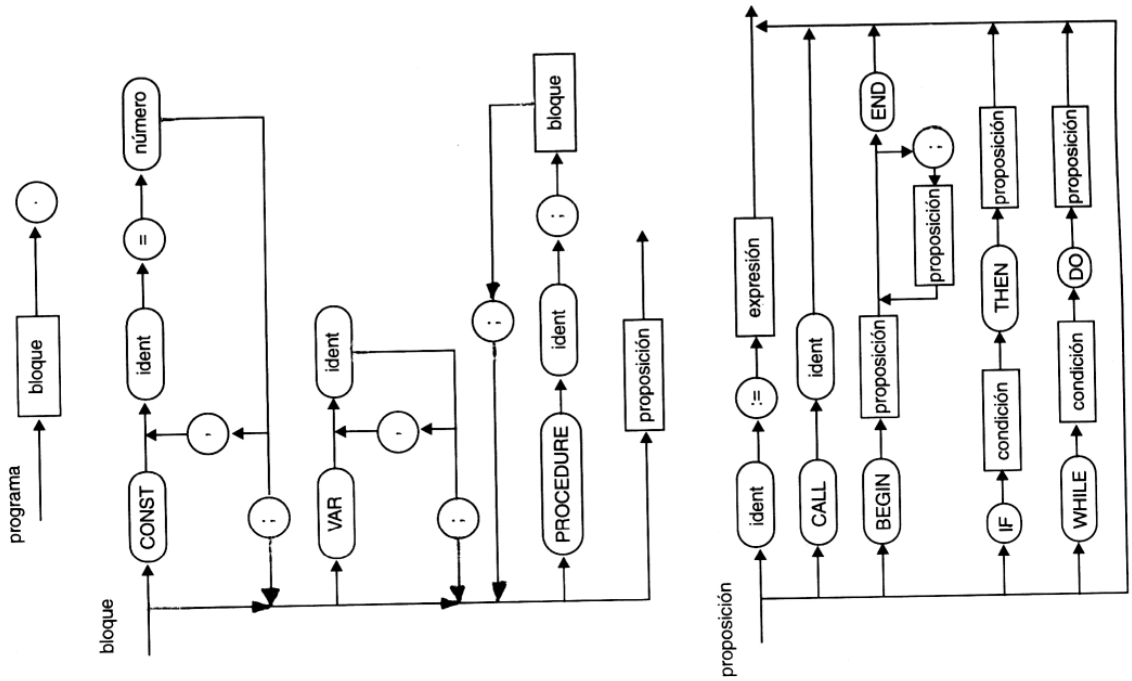
A partir del símbolo inicial *S* pueden generarse las frases *xz*, *xyz*, *xyyz*, *xyyyz*, *xyyyyz*, ...

En síntesis, un lenguaje *L* se caracterizará con referencia a una gramática *G (T, N, P, S)*, donde:

- **❒ *T* es el conjunto de símbolos terminales**
- **❒ *N* es el conjunto de símbolos no terminales**
- **❒ *P* es el conjunto de producciones**
- **❒ *S* es el símbolo inicial (debe ser un símbolo no terminal)**

3. El lenguaje de programación PL/0





Ejemplo de programa escrito en PL/0

```
const M = 7, N = 85;
var X, Y, Z, Q, R;

procedure MULTIPLICAR;
var A, B;
begin
  A := X;
  B := Y;
  Z := 0;
  while B > 0 do
    begin
      if odd B then Z := Z + A;
      A := A * 2;
      B := B / 2
    end
end;

procedure DIVIDIR;
var W;
begin
  R := X;
  Q := 0;
  W := Y;
  while W <= R do W := W * 2;
  while W > Y do
    begin
      Q := Q * 2;
      W := W / 2;
      if W <= R then
        begin
          R := R - W;
          Q := Q + 1
        end
    end
end;

procedure MCD;
var F, G;
begin
  F := X;
  G := Y;
  while F <> G do
    begin
      if F < G then G := G - F;
      if G < F then F := F - G
    end;
  Z := F
end;

begin
  X := M;  Y := N;  call MULTIPLICAR;
  X := 25; Y := 3;  call DIVIDIR;
  X := 84; Y := 36; call MCD
end.
```

Para describir el lenguaje PL/0 se utilizó una alternativa a la notación BNF: los *grafos de sintaxis*. Ambas notaciones son equivalentes, aunque los grafos dan una imagen más clara de la estructura del lenguaje cuya sintaxis describen.

Hay un grafo de sintaxis por cada producción.

Los símbolos no terminales son representados mediante nombres encerrados en rectángulos, con excepción del símbolo inicial, que solamente aparece al comienzo de la primera producción.

Los símbolos terminales son representados mediante nombres encerrados en círculos o rectángulos con bordes redondeados, y pueden ser de dos tipos: si el nombre está en mayúsculas representa una palabra reservada del lenguaje, pero si está en minúsculas se trata del nombre de un grupo de símbolos terminales, y no de un símbolo propiamente dicho, ya que hacer una enumeración completa sería poco práctico (cuando no imposible).

A pesar de su pequeño tamaño, PL/0 es relativamente completo. La asignación es su proposición básica. Los conceptos fundamentales de la programación estructurada (secuencia, condición y repetición) están representados por las proposiciones *begin/end*, *if* y *while*. PL/0 incorpora el concepto de subrutina mediante declaraciones de procedimientos y proposiciones de llamadas a los mismos. Esto ofrece la oportunidad de presentar el concepto de localidad de las constantes, las variables y los procedimientos.

Para reducir su complejidad, PL/0 sólo ofrece un tipo de datos: los enteros (en este curso: enteros de 32 bits con signo). Es posible declarar variables y constantes de este tipo. PL/0 dispone de los operadores aritméticos y relacionales convencionales.

4. Análisis léxico

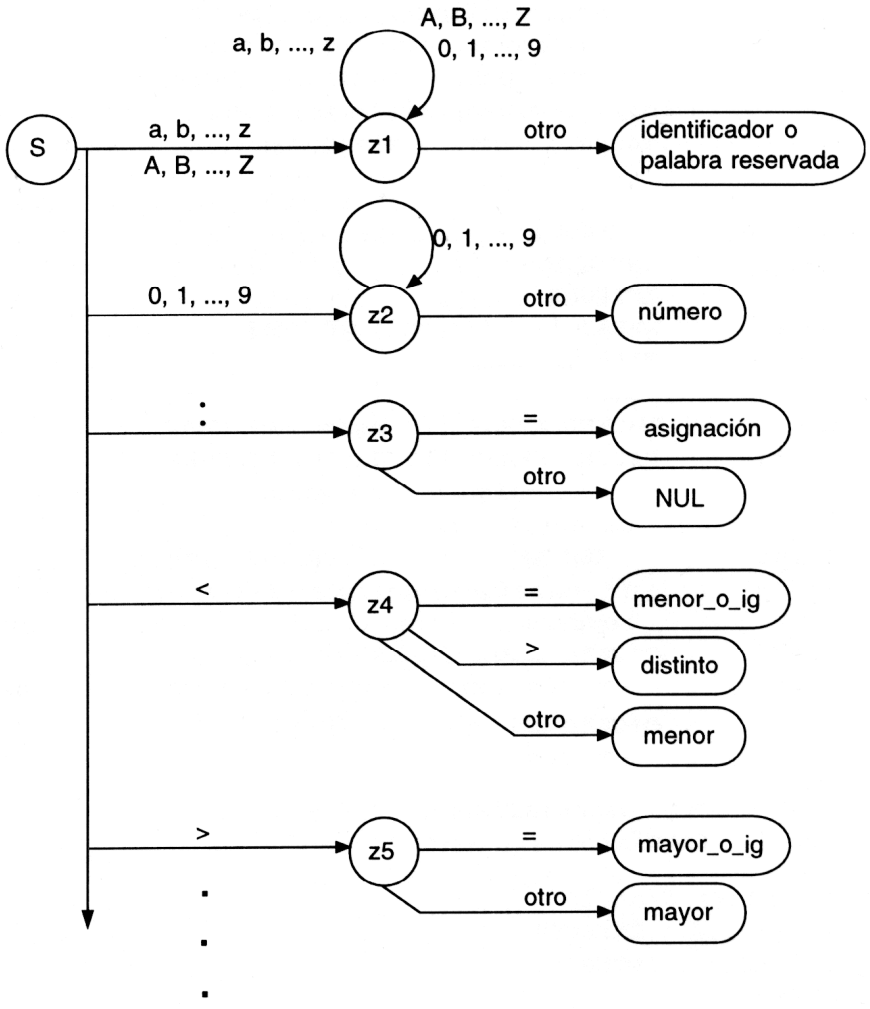
La tarea del analizador léxico consiste en:

a. **Saltear los separadores (blancos, tabulaciones, comentarios).**
b. **Reconocer los símbolos válidos e informar sobre los no válidos.**
c. **Llevar la cuenta de los renglones del programa.**
d. **Copiar los caracteres de entrada a la salida, generando un listado con los renglones numerados.**

El analizador léxico más simple es el de los lenguajes cuyos símbolos están compuestos por un único carácter. Esto no es lo más frecuente en el caso de los lenguajes de programación, donde las palabras reservadas, los identificadores, los números y los operadores pueden estar compuestos por más de un carácter. Para describir estos elementos, lo más conveniente es utilizar gramáticas regulares, o sea, gramáticas *G (T, N, P, S)* en las que cada producción *P* tiene la forma *A ::= aB* o *A ::= a*, y donde *A* y *B* pertenecen a *N* y *a* pertenece a *T*.

Es posible diseñar un autómata finito *F* para cada gramática regular *G*. Este autómata *F* acepta las frases del lenguaje definido por *G*, es decir, *L (G) = L (F)*.

A continuación, se presenta un diagrama de transición rudimentario del autómata que reconoce los símbolos del lenguaje PL/0.



El analizador léxico que se desarrollará para este curso deberá ser un procedimiento que tenga una forma similar a la siguiente (la forma, en definitiva, dependerá del lenguaje utilizado para implementarlo):

```
type terminal = (nulo, _begin, _call, _const, .... , coma, pto, ....);
     archivo = file of char;
     str63 = string [63];

procedure scanner (var Fuente, Listado: archivo; var S: terminal; var
Cad: str63; var Restante: string; var NumLinea: integer);
```

Cada vez que se llame al procedimiento scanner y la cadena Restante esté vacía o sólo contenga separadores, se leerá en Restante un nuevo renglón del archivo Fuente y se lo escribirá en el archivo Listado, anteponiéndole el valor actualizado de la variable NumLinea. Si, en cambio, la variable Restante contuviera caracteres útiles para formar símbolos, se los utilizará (borrándolos de Restante) para formar el próximo símbolo terminal S y la cadena de caracteres Cad correspondiente.

Por ejemplo, si el archivo Fuente contiene en sus dos primeros renglones:

```
CONST A=2;
procedure RAIZ;
```

Estos serán los valores luego de cada llamada:

Llamada	Listado	S	Cad	Restante	NumLinea
1	1: CONST A=2;	_const	'CONST'	' A=2;'	1
2	1: CONST A=2;	identificador	'A'	'=2;'	1
3	1: CONST A=2;	igual	'='	'2;'	1
4	1: CONST A=2;	numero	'2'	';'	1
5	1: CONST A=2;	ptoycoma	';'	''	1
6	1: CONST A=2; 2: procedure RAIZ;	_procedure	'PROCEDURE'	' RAIZ;'	2
7	1: CONST A=2; 2: procedure RAIZ;	identificador	'RAIZ'	';'	2
8	1: CONST A=2; 2: procedure RAIZ;	ptoycoma	';'	''	2

5. Análisis sintáctico

El proceso de determinar si una frase puede ser generada a partir de un conjunto de producciones se denomina *parsing*.

En el ejemplo 2, la frase *xyyz* se obtiene aplicando una vez *S* y tres veces *A* (las dos primeras veces que se aplica A se elige la opción de la derecha y la última vez la opción de la izquierda). Cuál producción se aplica surge inmediatamente al leer la frase de a un símbolo, de izquierda a derecha.

Veamos ahora el siguiente caso:

Ejemplo 3

S ::= A|B

A ::= xA|y

B ::= xB|z

Determinar las producciones aplicadas para generar *xxxxxxz* sólo es posible una vez que se leyó la frase completa, ya que habiendo leído sólo la primera *x* no es posible saber si al aplicar *S* corresponde elegir *A* o *B*.

Otro caso problemático se muestra a continuación:

Ejemplo 4

S ::= Ax

A ::= x|ϵ

Para generar la frase *x*, sólo es posible saber si corresponde aplicar la parte izquierda o la parte derecha de *A* una vez que *A* ya ha sido aplicada (bien o mal) y aparece la *x* final de *S*.

Las gramáticas que no presentan tales dificultades se denominan LL(1). La primera "L" significa que la entrada será leída de izquierda a derecha, y la segunda "L" indica derivaciones por la izquierda. El número "1" significa que alcanza con leer por anticipado un símbolo en cualquier paso del proceso de análisis sintáctico (o sea, solamente se emplea un símbolo de preanálisis). Al sistema de grafos con que se representa una gramática de este tipo se lo conoce como *grafo de sintaxis determinístico*.

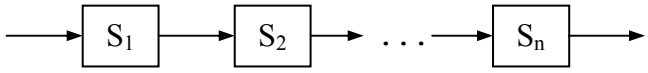
El paso siguiente consiste en construir un reconocedor sintáctico (*parser*) para una sintaxis dada. Este tipo de programa se deriva directamente del grafo de sintaxis determinístico y requiere de un procedimiento que funcione como scanner, salvo que los símbolos del lenguaje consten de un único carácter, en cuyo caso cualquier procedimiento de entrada estándar servirá para el ingreso del siguiente símbolo.

Para escribir un reconocedor sintáctico a partir de un grafo de sintaxis determinístico deberán seguirse las siguientes reglas:

R1. Reducir el sistema de grafos a la menor cantidad de grafos que sea posible, realizando para ello las sustituciones que sean necesarias.

R2. Declarar para cada grafo un procedimiento que contenga las sentencias resultantes de aplicarle al grafo las reglas R3 a R7.

R3. Una secuencia de elementos

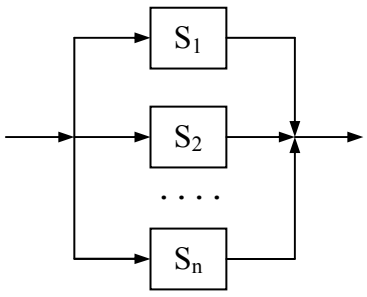


se traduce como una sentencia compuesta:

```
begin
  T(S1); T(S2); ... T(Sn)
end
```

(donde $T(S_i)$ es la sentencia obtenida al traducir el grafo S_i)

R4. Una opción entre elementos

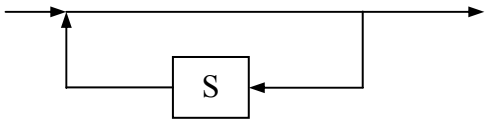


se traduce como una sentencia condicional:

if SIM in L_1 then T(S_1) else
if SIM in L_2 then T(S_2) else
....
if SIM in L_n then T(S_n);

donde SIM es el símbolo devuelto por el analizador léxico y L_i es el conjunto de símbolos iniciales de S_i. Siempre que L_i conste de un único símbolo *a*, "SIM in L_i" podrá expresarse como "SIM = *a*"

R5. Un bucle de la forma



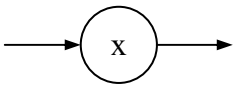
se traduce como la sentencia:

while SIM in L do T(S)

R6. Una referencia a otro grafo A

se traduce como una sentencia de llamada al procedimiento A

R7. Una referencia a un símbolo terminal x



se traduce como la sentencia:

if SIM = x then SCANNER(SIM) else ERROR

donde ERROR es un procedimiento encargado del tratamiento de los errores.

El parser funciona haciendo una llamada al scanner (para tener un símbolo leído de antemano) y una llamada al procedimiento correspondiente

al primero de los grafos. A partir de este procedimiento se irán realizando llamadas a los demás, hasta que aparezca algún error o se termine reconociendo satisfactoriamente el programa leído.

Algunas construcciones redundantes pueden suprimirse al depurar el parser resultante de la estricta aplicación de las reglas R1 a R7.

Veamos ahora el siguiente caso:

Ejemplo 5

A ::= x|(B)

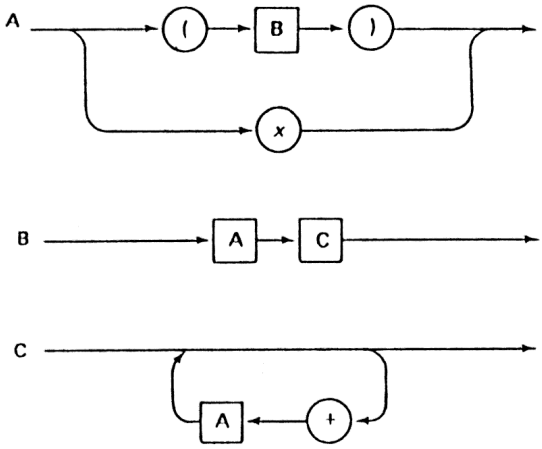
B ::= AC

C ::= {+A}

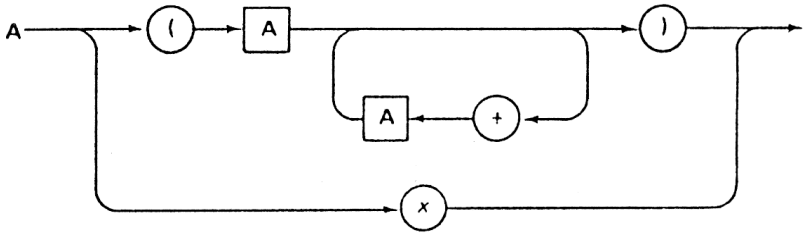
Aquí, los símbolos terminales son "x", "(", ")" y "+". Es posible utilizar el procedimiento *read* para llevar a cabo la función del scanner, ya que todos los símbolos están formados por un único carácter. Algunas de las posibles frases del lenguaje son:

x (x) (x+x) ((x))

Los grafos equivalentes a la gramática expresada en BNF son:



Aplicando la regla R1:



Aplicando las reglas R2 a R7:

```
program PARSER;
var SIM: char;
        procedure A;
        begin
          if SIM = 'x'
            then read (SIM)
            else if SIM = '('
                   then begin
                          read (SIM);
                          A;
                          while SIM = '+' do begin
                                              read (SIM);
                                              A
                                            end;
                          if SIM = ')' then read (SIM)
                                       else ERROR
                        end
                   else ERROR
        end;
begin
  read (SIM);
  A
end.
```

6. Análisis semántico

El análisis sintáctico no garantiza que un programa esté libre de errores. El siguiente programa escrito en PL/0 es sintácticamente correcto, pero contiene un error semántico, ya que un identificador de constante no puede ser llamado mediante la proposición call (que es exclusiva para identificadores de procedimiento).

Ejemplo 6

```
const K = 9;
var V;
procedure P;
  var X;
  begin
    X := K * 2;
    V := X
  end;
call K.
```

Para poder determinar si un programa es semánticamente correcto, el compilador deberá cargar en una tabla cada identificador que se declare. En esa tabla podrán consultarse:

- nombre del identificador
- tipo de identificador
- valor del identificador: Un número de 16 bits con distinto significado, según el tipo de identificador de que se trate:
  - ❒ constante: el valor de la constante
  - ❒ variable: la dirección de memoria a que se refiere la variable (sólo el desplazamiento)
  - ❒ procedimiento: la dirección de memoria donde comienza la proposición a ejecutar en el bloque

Una posible definición del tipo con que se declarará la tabla podría ser la siguiente:

```
type TABLA = array [0..MaxIdent-1] of record
                                        NOM: str63;
                                        TIPO: terminal;
                                        VALOR: longInt
                                      end;
```

El procedimiento BLOQUE recibirá como parámetro (pasaje por valor) la entrada de la tabla a partir de la cual se podrán cargar identificadores. Este parámetro podría llamarse BASE. Cuando se llama a BLOQUE desde el grafo PROGRAMA, se le pasa como parámetro el valor 0, ya que no hay identificadores previamente declarados.

El procedimiento BLOQUE contendrá un variable local llamada DESPLAZAMIENTO, que se irá incrementando con cada identificador que se declare. Cuando se llama a BLOQUE desde el grafo BLOQUE, se le pasa como parámetro el valor BASE+DESPLAZAMIENTO.

De esta forma, cada vez que se declare un identificador, deberá verificarse si éste ya había sido declarado en el mismo ámbito, es decir, entre las posiciones de la tabla delimitadas por BASE y BASE+DESPLAZAMIENTO-1.

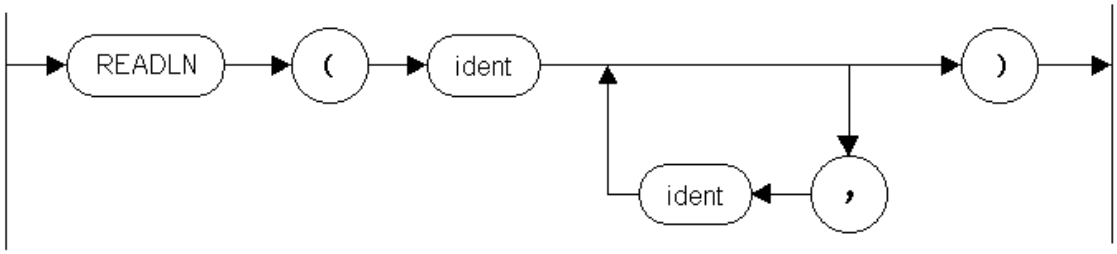
Para el análisis semántico, los identificadores se buscarán en toda la tabla, comenzando en la posición BASE+DESPLAZAMIENTO-1 y retrocediendo hasta la posición 0. De esta forma, siempre se encontrará primero el identificador que haya sido declarado localmente. En caso de no encontrarse el identificador, deberá darse aviso de la falta de declaración.

Una vez hallado el identificador en la tabla, con el campo TIPO podrá verificarse si el identificador es semánticamente correcto en la posición del programa donde fue encontrado.

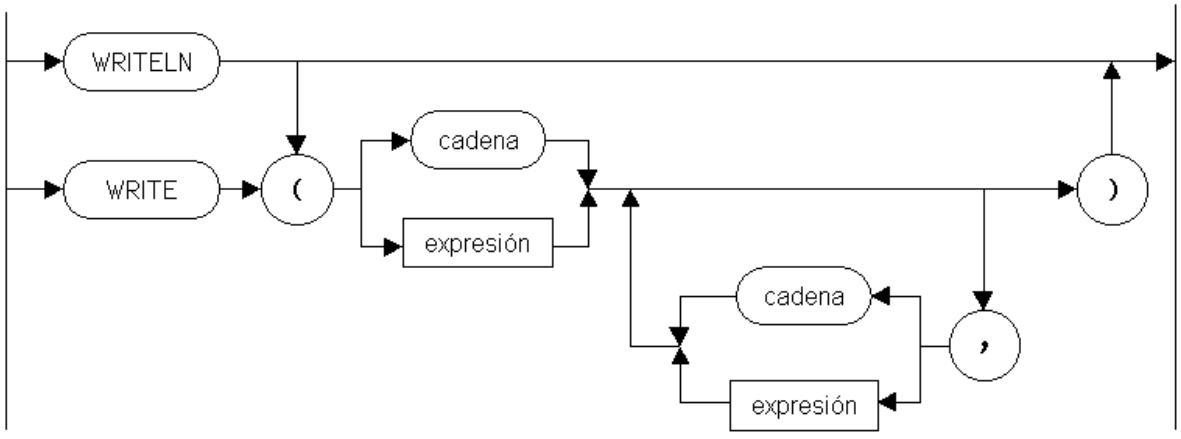
7. Generación de código

Antes de comenzar con el estudio de la generación de código, extenderemos la sintaxis de PL/0 mediante el agregado de tres proposiciones de E/S, ya que de lo contrario no podríamos ingresar valores en tiempo de ejecución ni podríamos ver los resultados de los cálculos realizados por el programa. Adoptaremos los nombres de los procedimientos de E/S de Pascal (*readln*, *write* y *writeln*), y les daremos una funcionalidad similar a la que tienen en este lenguaje cuando se los utiliza para realizar entrada desde el teclado y salida hacia la pantalla. Además, incorporaremos un símbolo terminal nuevo, la *cadena* literal, para permitir mostrar mensajes por pantalla. El analizador léxico considerará que cualquier secuencia de caracteres encerrada entre apóstrofos es una cadena.

La proposición de entrada READLN tendrá la sintaxis:



Las proposiciones de salida WRITELN y WRITE tendrán la sintaxis:



Finalmente, llegamos a la generación de código. Como plataforma de destino de la compilación, en este curso se adoptará una PC con un microprocesador que implemente IA-32 (Intel Architecture, 32-bit) y utilice el sistema operativo Linux.

Utilizaremos (explícitamente) sólo 5 de los registros de 32 bits disponibles: EDI, EAX, EBX, ECX y EDX. Eventualmente, también accederemos a la parte baja de EAX, a través del subregistro AL (el cual permite acceder a los 8 bits menos significativos de EAX).

De los modos de direccionamiento soportados por el microprocesador, solamente vamos a usar los siguientes tres:

	EJEMPLOS	
modo registro	ADD EAX, EBX	(carga EAX con el valor de la suma de EAX más EBX)
modo inmediato	MOV EAX, 00000072	(carga EAX con el valor 00000072)
modo indexado	MOV EAX, [EDI+00000072]	(carga EAX con el contenido de la dirección EDI+00000072)

El archivo ejecutable generado por el compilador será de tipo ELF (*Executable and Linkable Format*). Este tipo de archivo ejecutable es el estándar en las versiones de Linux a partir del kernel 1.2, ya que, hasta entonces, se utilizaba el formato *a.out*.

El código del programa estará compuesto por una parte de longitud fija y una parte de longitud variable.

La parte de longitud fija contendrá:

- **el encabezado ELF, con su número mágico 7F 45 4C 46 (·ELF), formado por campos que contienen las características del archivo y campos que contienen punteros hacia las otras partes del archivo;**
- **la tabla del encabezado de programa (*Program Header Table*), con campos usados para gestionar la carga y la ejecución del programa;**
- **las cadenas (terminadas en cero) de los encabezados de secciones.**
- **la tabla de los encabezados de las secciones (*Section Header Table*)**
- **el comienzo de la sección *text*, formado por instrucciones del x86 mediante las que se implementan las proposiciones de E/S. Estas instrucciones constituyen rutinas desde las cuales se realizarán las llamadas a las funciones del kernel de Linux.**

La parte de longitud fija, inicialmente, deberá tener el siguiente contenido:

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00   .ELF............
00000010   02 00 03 00 01 00 00 00  80 84 04 08 34 00 00 00   ............4...
00000020   65 00 00 00 00 00 00 00  34 00 20 00 01 00 28 00   e.......4. ...(.
00000030   03 00 01 00 01 00 00 00  00 00 00 00 00 80 04 08   ................
00000040   00 80 04 08 97 05 00 00  97 05 00 00 07 00 00 00   ................
00000050   00 10 00 00 00 2e 73 68  73 74 72 74 61 62 00 2e   ......shstrtab..
00000060   74 65 78 74 00 00 00 00  00 00 00 00 00 00 00 00   text............
00000070   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000080   00 00 00 00 00 00 00 00  00 00 00 00 00 01 00 00   ................
00000090   00 03 00 00 00 00 00 00  00 00 00 00 00 54 00 00   .............T..
000000a0   00 11 00 00 00 00 00 00  00 00 00 00 00 01 00 00   ................
000000b0   00 00 00 00 00 0b 00 00  00 01 00 00 00 06 00 00   ................
000000c0   00 e0 80 04 08 e0 00 00  00 b7 04 00 00 00 00 00   .....à...·......
000000d0   00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ................
000000e0   52 51 53 50 b8 04 00 00  00 bb 01 00 00 00 89 e1   RQSP¸....».....á
000000f0   ba 01 00 00 00 cd 80 58  5b 59 5a c3 55 89 e5 81   º....Í.X[YZÃU.å.
00000100   ec 24 00 00 00 52 51 53  b8 36 00 00 00 bb 00 00   ì$...RQS¸6...»..
00000110   00 00 b9 01 54 00 00 8d  55 dc cd 80 81 65 e8 f5   ..¹.T...UÜÍ..eèõ
00000120   ff ff ff b8 36 00 00 00  bb 00 00 00 00 b9 02 54   ÿÿÿ¸6...»....¹.T
00000130   00 00 8d 55 dc cd 80 31  c0 50 b8 03 00 00 00 bb   ...UÜÍ.1ÀP¸....»
00000140   00 00 00 00 89 e1 ba 01  00 00 00 cd 80 81 4d e8   .....áº....Í..Mè
00000150   0a 00 00 00 b8 36 00 00  00 bb 00 00 00 00 b9 02   ....¸6...»....¹.
00000160   54 00 00 8d 55 dc cd 80  58 5b 59 5a 89 ec 5d c3   T...UÜÍ.X[YZ.ì]Ã
00000170   b8 04 00 00 00 bb 01 00  00 00 cd 80 c3 90 90 90   ¸....»....Í.Ã...
00000180   b0 0a e8 59 ff ff ff c3  04 30 e8 51 ff ff ff c3   °.èYÿÿÿÃ.0èQÿÿÿÃ
00000190   3d 00 00 00 80 75 4e b0  2d e8 42 ff ff ff b0 02   =....uN°-èBÿÿÿ°.
000001a0   e8 e3 ff ff ff b0 01 e8  dc ff ff ff b0 04 e8 d5   èãÿÿÿ°.èÜÿÿÿ°.èÕ
000001b0   ff ff ff b0 07 e8 ce ff  ff ff b0 04 e8 c7 ff ff   ÿÿÿ°.èÎÿÿÿ°.èÇÿÿ
000001c0   ff b0 08 e8 c0 ff ff ff  b0 03 e8 b9 ff ff ff b0   ÿ°.èÀÿÿÿ°.è¹ÿÿÿ°
000001d0   06 e8 b2 ff ff ff b0 04  e8 ab ff ff ff b0 08 e8   .è²ÿÿÿ°.è«ÿÿÿ°.è
000001e0   a4 ff ff ff c3 3d 00 00  00 00 7d 0b 50 b0 2d e8   ¤ÿÿÿÃ=....}.P°-è
000001f0   ec fe ff ff 58 f7 d8 3d  0a 00 00 00 0f 8c ef 00   ìþÿÿX÷Ø=......ï.
00000200   00 00 3d 64 00 00 00 0f  8c d1 00 00 00 3d e8 03   ..=d.....Ñ...=è.
00000210   00 00 0f 8c b3 00 00 00  3d 10 27 00 00 0f 8c 95   ....³...=.'.....
00000220   00 00 00 3d a0 86 01 00  7c 7b 3d 40 42 0f 00 7c   ...= ...{=@B..
00000230   61 3d 80 96 98 00 7c 47  3d 00 e1 f5 05 7c 2d 3d   a=....G=.áõ.-=
00000240   00 ca 9a 3b 7c 13 ba 00  00 00 00 bb 00 ca 9a 3b   .Ê.;.º....».Ê.;
00000250   f7 fb 52 e8 30 ff ff ff  58 ba 00 00 00 00 bb 00   ÷ûRè0ÿÿÿXº....».
00000260   e1 f5 05 f7 fb 52 e8 1d  ff ff ff 58 ba 00 00 00   áõ.÷ûRè.ÿÿÿXº...
00000270   00 bb 80 96 98 00 f7 fb  52 e8 0a ff ff ff 58 ba   .»....÷ûRè.ÿÿÿXº
00000280   00 00 00 00 bb 40 42 0f  00 f7 fb 52 e8 f7 fe ff   ....»@B..÷ûRè÷þÿ
00000290   ff 58 ba 00 00 00 00 bb  a0 86 01 00 f7 fb 52 e8   ÿXº....» ...÷ûRè
000002a0   e4 fe ff ff 58 ba 00 00  00 00 bb 10 27 00 00 f7   äþÿÿXº....».'..÷
000002b0   fb 52 e8 d1 fe ff ff 58  ba 00 00 00 00 bb e8 03   ûRèÑþÿÿXº....»è.
000002c0   00 00 f7 fb 52 e8 be fe  ff ff 58 ba 00 00 00 00   ..÷ûRè¾þÿÿXº....
000002d0   bb 64 00 00 00 f7 fb 52  e8 ab fe ff ff 58 ba 00   »d...÷ûRè«þÿÿXº.
000002e0   00 00 00 bb 0a 00 00 00  f7 fb 52 e8 98 fe ff ff   ...»....÷ûRè.þÿÿ
000002f0   58 e8 92 fe ff ff c3 90  90 90 90 90 90 90 90 90   Xè.þÿÿÃ.........
00000300   b8 01 00 00 00 bb 00 00  00 00 cd 80 90 90 90 90   ¸....»....Í.....
00000310   b9 00 00 00 00 b3 03 51  53 e8 de fd ff ff 5b 59   ¹....³.QSèÞýÿÿ[Y
00000320   3c 0a 0f 84 34 01 00 00  3c 7f 0f 84 94 00 00 00   <...4...<.......
00000330   3c 2d 0f 84 09 01 00 00  3c 30 7c db 3c 39 7f d7   <-......<0Û<9.×
00000340   2c 30 80 fb 00 74 d0 80  fb 02 75 0c 81 f9 00 00   ,0.û.tÐ.û.u..ù..
00000350   00 00 75 04 3c 00 74 bf  80 fb 03 75 0a 3c 00 75   ..u.<.t¿.û.u.<.u
```

```
00000360   04 b3 00 eb 02 b3 01 81  f9 cc cc cc 0c 7f a8 81   .³.ë.³..ùÌÌÌ..¨.
00000370   f9 34 33 33 f3 7c a0 88  c7 b8 0a 00 00 00 f7 e9   ù433ó .Ç¸....÷é
00000380   3d 08 00 00 80 74 11 3d  f8 ff ff 7f 75 13 80 ff   =....t.=øÿÿ.u..ÿ
00000390   07 7e 0e e9 7f ff ff ff  80 ff 08 0f 8f 76 ff ff   .~.é.ÿÿÿ.ÿ...vÿÿ
000003a0   ff b9 00 00 00 00 88 f9  80 fb 02 74 04 01 c1 eb   ÿ¹.....ù.û.t..Áë
000003b0   03 29 c8 91 88 f8 51 53  e8 cb fd ff ff 5b 59 e9   .)È..øQSèËýÿÿ[Yé
000003c0   53 ff ff ff 80 fb 03 0f  84 4a ff ff ff 51 53 b0   Sÿÿÿ.û...Jÿÿÿ QS°
000003d0   08 e8 0a fd ff ff b0 20  e8 03 fd ff ff b0 08 e8   .è.ýÿÿ° è.ýÿÿ°.è
000003e0   fc fc ff ff 5b 59 80 fb  00 75 07 b3 03 e9 25 ff   üüÿÿ[Y.û.u.³.é%ÿ
000003f0   ff ff 80 fb 02 75 0f 81  f9 00 00 00 00 75 07 b3   ÿÿ.û.u..ù....u.³
00000400   03 e9 11 ff ff ff 89 c8  b9 0a 00 00 00 ba 00 00   .é.ÿÿÿ.È¹....º..
00000410   00 00 3d 00 00 00 00 7d  08 f7 d8 f7 f9 f7 d8 eb   ..=....}.÷Ø÷ù÷Øë
00000420   02 f7 f9 89 c1 81 f9 00  00 00 00 0f 85 e6 fe ff   .÷ù.Á.ù......æþÿ
00000430   ff 80 fb 02 0f 84 dd fe  ff ff b3 03 e9 d6 fe ff   ÿ.û...Ýþÿÿ³.éÖþÿ
00000440   ff 80 fb 03 0f 85 cd fe  ff ff b0 2d 51 53 e8 8d   ÿ.û...Íþÿÿ°-QSè.
00000450   fc ff ff 5b 59 b3 02 e9  bb fe ff ff 80 fb 03 0f   üÿÿ[Y³.é»þÿÿ.û..
00000460   84 b2 fe ff ff 80 fb 02  75 0c 81 f9 00 00 00 00   .²þÿÿ.û.u..ù....
00000470   0f 84 a1 fe ff ff 51 e8  04 fd ff ff 59 89 c8 c3   ..¡þÿÿQè.ýÿÿY.ÈÃ
```

El significado de los campos de los encabezados es el siguiente:

```
/* ELF HEADER */

memoria[0] = 0x7F; //
memoria[1] = 0x45; // E
memoria[2] = 0x4C; // L
memoria[3] = 0x46; // F

memoria[4] = 0x01; // Flags
memoria[5] = 0x01;
memoria[6] = 0x01;

memoria[7] = 0x00; // Padding zeroes
memoria[8] = 0x00;
memoria[9] = 0x00;
memoria[10] = 0x00;
memoria[11] = 0x00;
memoria[12] = 0x00;
memoria[13] = 0x00;
memoria[14] = 0x00;
memoria[15] = 0x00;

memoria[16] = 0x02; // Type: 2
memoria[17] = 0x00; // (Executable)

memoria[18] = 0x03; // Machine: 3 = i32
memoria[19] = 0x00;

memoria[20] = 0x01; // Version: 1
memoria[21] = 0x00; // (Current)
memoria[22] = 0x00;
memoria[23] = 0x00;

memoria[24] = 0x80; // Entry: absolute
memoria[25] = 0x84; // entry point
memoria[26] = 0x04; // (_start)
memoria[27] = 0x08;

memoria[28] = 0x34; // File offset to PHT
memoria[29] = 0x00;
memoria[30] = 0x00;
memoria[31] = 0x00;

memoria[32] = 0x65; // File offset to SHT
memoria[33] = 0x00;
memoria[34] = 0x00;
memoria[35] = 0x00;

memoria[36] = 0x00; // Flags: 0 for i386
memoria[37] = 0x00;
memoria[38] = 0x00;
memoria[39] = 0x00;

memoria[40] = 0x34; // ELF Header size
memoria[41] = 0x00;

memoria[42] = 0x20; // PHT Entry size
memoria[43] = 0x00;

memoria[44] = 0x01; // Entries in PHT
memoria[45] = 0x00;

memoria[46] = 0x28; // SHT Entry size
memoria[47] = 0x00;

memoria[48] = 0x03; // Entries in SHT
memoria[49] = 0x00;

memoria[50] = 0x01; // Index of .shstrtab
memoria[51] = 0x00;

/* PHT (1 Entry) */

memoria[52] = 0x01; // 1 = load into
memoria[53] = 0x00; // memory
memoria[54] = 0x00;
memoria[55] = 0x00;

memoria[56] = 0x00; // file offset to
memoria[57] = 0x00; // start of segment
memoria[58] = 0x00;
memoria[59] = 0x00;
```

```
memoria[60] = 0x00; // Virtual address
memoria[61] = 0x80; // where loaded
memoria[62] = 0x04;
memoria[63] = 0x08;

memoria[64] = 0x00; // Absolute address
memoria[65] = 0x80; // where loaded
memoria[66] = 0x04;
memoria[67] = 0x08;

memoria[68] = 0x14; // File size
memoria[69] = 0x04;
memoria[70] = 0x00;
memoria[71] = 0x00;

memoria[72] = 0x14; // Memory size
memoria[73] = 0x04;
memoria[74] = 0x00;
memoria[75] = 0x00;

memoria[76] = 0x07; // Permissions (rwx)
memoria[77] = 0x00;
memoria[78] = 0x00;
memoria[79] = 0x00;

memoria[80] = 0x00; // Alignment required
memoria[81] = 0x10;
memoria[82] = 0x00;
memoria[83] = 0x00;

/* SH STRING TABLE (3 Strings) */

memoria[84] = 0x00; // Empty string

memoria[85] = 0x2E; // .shstrtab
memoria[86] = 0x73;
memoria[87] = 0x68;
memoria[88] = 0x73;
memoria[89] = 0x74;
memoria[90] = 0x72;
memoria[91] = 0x74;
memoria[92] = 0x61;
memoria[93] = 0x62;
memoria[94] = 0x00;

memoria[95] = 0x2E; // .text
memoria[96] = 0x74;
memoria[97] = 0x65;
memoria[98] = 0x78;
memoria[99] = 0x74;
memoria[100] = 0x00;

/* SHT (3 Entries) */

// Entry 0 (reserved)

memoria[101] = 0x00; // name
memoria[102] = 0x00;
memoria[103] = 0x00;
memoria[104] = 0x00;

memoria[105] = 0x00; // type
memoria[106] = 0x00;
memoria[107] = 0x00;
memoria[108] = 0x00;

memoria[109] = 0x00; // flags
memoria[110] = 0x00;
memoria[111] = 0x00;
memoria[112] = 0x00;

memoria[113] = 0x00; // addr
memoria[114] = 0x00;
memoria[115] = 0x00;
memoria[116] = 0x00;

memoria[117] = 0x00; // offset
memoria[118] = 0x00;
memoria[119] = 0x00;
memoria[120] = 0x00;

memoria[121] = 0x00; // size
memoria[122] = 0x00;
memoria[123] = 0x00;
memoria[124] = 0x00;

memoria[125] = 0x00; // link
memoria[126] = 0x00;
memoria[127] = 0x00;
memoria[128] = 0x00;

memoria[129] = 0x00; // info
memoria[130] = 0x00;
memoria[131] = 0x00;
memoria[132] = 0x00;

memoria[133] = 0x00; // addrAlign
memoria[134] = 0x00;
memoria[135] = 0x00;
memoria[136] = 0x00;

memoria[137] = 0x00; // entSize
memoria[138] = 0x00;
memoria[139] = 0x00;
memoria[140] = 0x00;

// Entry 1 (.shstrtab)

memoria[141] = 0x01; // name
memoria[142] = 0x00;
memoria[143] = 0x00;
memoria[144] = 0x00;

memoria[145] = 0x03; // type
memoria[146] = 0x00;
memoria[147] = 0x00;
memoria[148] = 0x00;

memoria[149] = 0x00; // flags
memoria[150] = 0x00;
memoria[151] = 0x00;
memoria[152] = 0x00;

memoria[153] = 0x00; // addr
memoria[154] = 0x00;
memoria[155] = 0x00;
memoria[156] = 0x00;

memoria[157] = 0x54; // offset
memoria[158] = 0x00;
memoria[159] = 0x00;
memoria[160] = 0x00;
```

```
memoria[161] = 0x11; // size
memoria[162] = 0x00;
memoria[163] = 0x00;
memoria[164] = 0x00;

memoria[165] = 0x00; // link
memoria[166] = 0x00;
memoria[167] = 0x00;
memoria[168] = 0x00;

memoria[169] = 0x00; // info
memoria[170] = 0x00;
memoria[171] = 0x00;
memoria[172] = 0x00;

memoria[173] = 0x01; // addrAlign
memoria[174] = 0x00;
memoria[175] = 0x00;
memoria[176] = 0x00;

memoria[177] = 0x00; // entSize
memoria[178] = 0x00;
memoria[179] = 0x00;
memoria[180] = 0x00;

// Entry 2 (.text)

memoria[181] = 0x0B; // name
memoria[182] = 0x00;
memoria[183] = 0x00;
memoria[184] = 0x00;

memoria[185] = 0x01; // type
memoria[186] = 0x00;
memoria[187] = 0x00;
memoria[188] = 0x00;

memoria[189] = 0x06; // flags
memoria[190] = 0x00;
memoria[191] = 0x00;
memoria[192] = 0x00;

memoria[193] = 0xE0; // addr
memoria[194] = 0x80;
memoria[195] = 0x04;
memoria[196] = 0x08;

memoria[197] = 0xE0; // offset
memoria[198] = 0x00;
memoria[199] = 0x00;
memoria[200] = 0x00;

memoria[201] = 0xB4; // size
memoria[202] = 0x03;
memoria[203] = 0x00;
memoria[204] = 0x00;

memoria[205] = 0x00; // link
memoria[206] = 0x00;
memoria[207] = 0x00;
memoria[208] = 0x00;

memoria[209] = 0x00; // info
memoria[210] = 0x00;
memoria[211] = 0x00;
memoria[212] = 0x00;

memoria[213] = 0x01; // addrAlign
memoria[214] = 0x00;
memoria[215] = 0x00;
memoria[216] = 0x00;

memoria[217] = 0x00; // entSize
memoria[218] = 0x00;
memoria[219] = 0x00;
memoria[220] = 0x00;
```

Las posiciones 197–200 (00C5–00C8 en hexadecimal) contienen la dirección de inicio de la sección *text*, que es donde estará ubicado el código ejecutable (las rutinas de E/S y la traducción del programa escrito en PL/0). Por lo tanto, las posiciones 221–223 (00DD–00DF en hexadecimal) deberán rellenarse con ceros, para que la sección *text* comience en la posición 224 (00E0 en hexadecimal).

La parte de longitud fija de la sección *text* contiene el código de las rutinas de E/S, de 224 a 1151 (00E0–047F en hexadecimal). Para poder invocar estas rutinas, no es necesario conocer su funcionamiento interno, ya que alcanza con saber en qué posición comienza cada una:

- **368 (0170 en hexadecimal): muestra por consola una cadena alojada a partir de la dirección guardada en ECX, y cuya longitud es EDX.**
- **384 (0180 en hexadecimal): envía un salto de línea a la consola.**
- **400 (0190 en hexadecimal): muestra por consola el número entero contenido en EAX.**
- **768 (0300 en hexadecimal): finaliza el programa**
- **784 (0310 en hexadecimal): lee por consola un número entero y lo deja guardado en EAX.**

La parte de longitud variable de la sección *text* contendrá las instrucciones resultantes de traducir el programa fuente escrito en PL/0. La traducción consiste en escribir instrucciones, byte a byte, en el archivo ejecutable. Estos bytes corresponderán a las siguientes instrucciones del x86 (los valores están en hexadecimal):

MNEMÓNICO	BYTES	SIGNIFICADO
MOV EDI, *abcdefgh*	BF *gh ef cd ab*	COPIA EL SEGUNDO OPERANDO EN EL PRIMERO
MOV EAX, [EDI+*abcdefgh*]	8B 87 *gh ef cd ab*	
MOV [EDI+*abcdefgh*], EAX	89 87 *gh ef cd ab*	
MOV EAX, *abcdefgh*	B8 *gh ef cd ab*	
MOV ECX, *abcdefgh*	B9 *gh ef cd ab*	
MOV EDX, *abcdefgh*	BA *gh ef cd ab*	
XCHG EAX, EBX	93	INTERCAMBIA LOS VALORES DE LOS OPERANDOS
PUSH EAX	50	MANDA EL VALOR DEL OPERANDO A LA PILA
POP EAX	58	EXTRAE EL VALOR DE LA PILA Y LO COLOCA EN EL OPERANDO
POP EBX	5B	

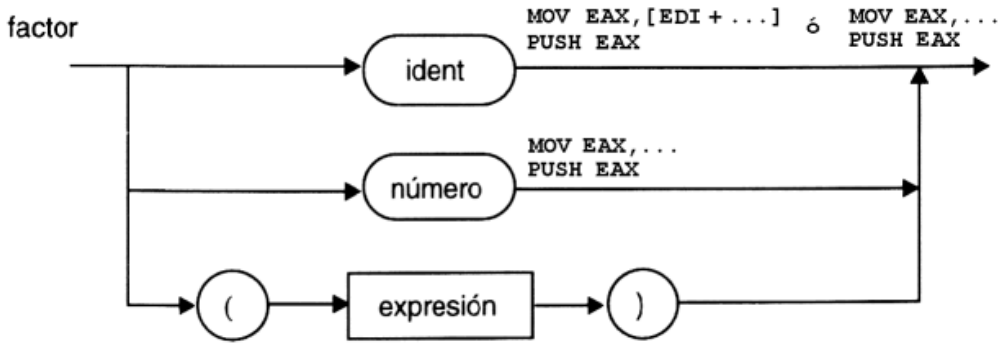
ADD EAX, EBX	**01 D8**	**SUMA AMBOS OPERANDOS Y COLOCA EL RESULTADO EN EL PRIMERO**
SUB EAX, EBX	**29 D8**	**LE RESTA EL SEGUNDO OPERANDO AL PRIMERO Y COLOCA EL RESULTADO EN EL PRIMER OPERANDO**
IMUL EBX	**F7 EB**	**COLOCA EN EDX:EAX EL PRODUCTO DE EAX POR EBX**
IDIV EBX	**F7 FB**	**DIVIDE EDX:EAX POR EL OPERANDO Y COLOCA EL COCIENTE EN EAX Y EL RESTO EN EDX**
CDQ	**99**	**LLENA TODOS LOS BITS DE EDX CON EL VALOR DEL BIT DEL SIGNO DE EAX**
NEG EAX	**F7 D8**	**CAMBIA EL SIGNO DE EAX**
TEST AL, *ab*	**A8 *ab***	**CALCULA EL "Y" ENTRE LOS OPERANDOS Y MODIFICA VARIAS BANDERAS, ENTRE ELLAS PF (PARITY FLAG)**
CMP EBX, EAX	**39 C3**	**COMPARA EL PRIMER OPERANDO CON EL SEGUNDO PARA QUE, SEGÚN EL RESULTADO DE LA COMPARACIÓN,**
JE *dir* JZ *dir*	**74 *ab***	**SEGÚN EL RESULTADO DE UNA COMPARACIÓN, SALTA A LA DIRECCIÓN UBICADA *ab* BYTES ANTES O DESPUÉS DE LA DIRECCIÓN ACTUAL. JE (JUMP IF =) JNE (JUMP IF NOT =) JG (JUMP IF >) JGE (JUMP IF > OR =) JL (JUMP IF <) JLE (JUMP IF < OR =) JPO (JUMP IF PARITY ODD)**
JNE *dir* JNZ *dir*	**75 *ab***	
JG *dir*	**7F *ab***	
JGE *dir*	**7D *ab***	
JL *dir*	**7C *ab***	
JLE *dir*	**7E *ab***	
JPO *dir*	**7B *ab***	
JMP *dir*	**E9 *gh ef cd ab***	**SALTA A LA DIRECCIÓN UBICADA *abcdefgh* BYTES ANTES O DESPUÉS DE LA DIRECCIÓN ACTUAL**
CALL *dir*	**E8 *gh ef cd ab***	**INVOCA LA SUBRUTINA UBICADA *abcdefgh* BYTES ANTES O DESPUÉS DE LA DIRECCIÓN ACTUAL**
RET	**C3**	**RETORNA AL PUNTO DESDE DONDE SE LLAMÓ UNA SUBRUTINA**

Los valores *ab* y *abcdefgh* representan números enteros de 8 y 32 bits, respectivamente. En las instrucciones, *abcdefgh* aparece invertido.

La primera instrucción del código traducido será la inicialización del registro EDI para que apunte a la dirección a partir de la cual estarán alojados los valores de las variables. Como esta dirección aún no se conoce, deberá reservarse el lugar grabando `BF 00 00 00 00`.

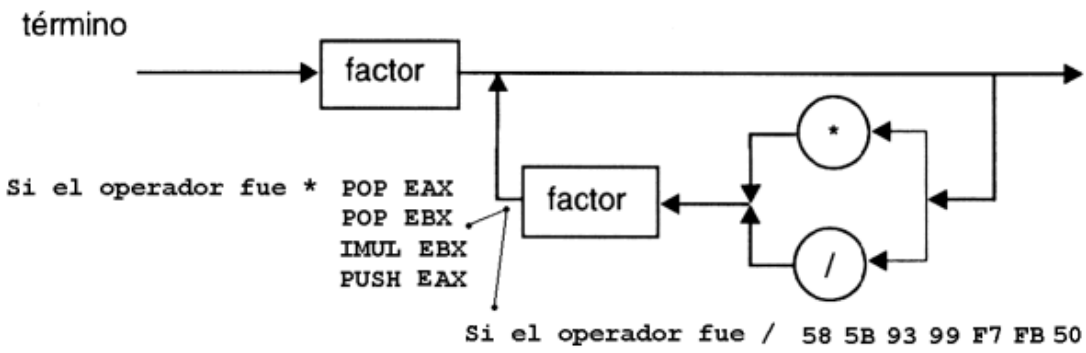
Un programa escrito en PL/0 hará uso intensivo de la pila. La idea general es que los factores se colocarán en la pila (con PUSH) para ser retirados (con POP) siempre que haya que calcular el valor de un término, una expresión o una condición.

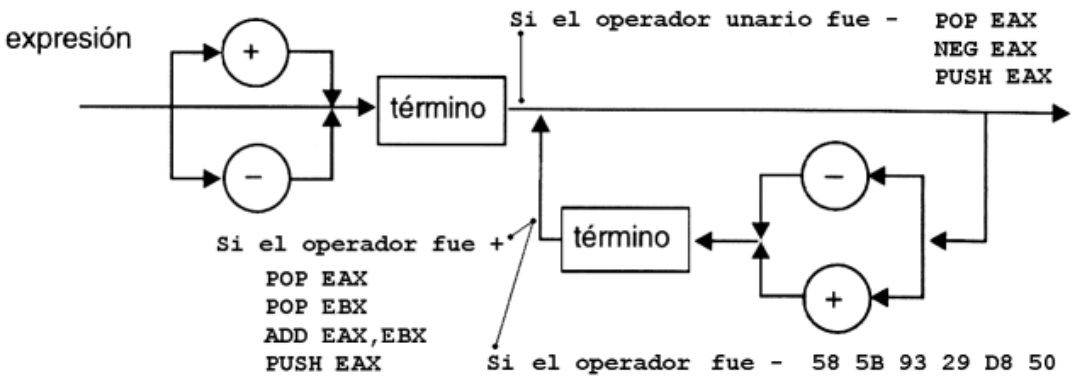
Para ver en detalle qué instrucciones deberán generarse (es decir, qué bytes deberán escribirse en la sección *text* del archivo ejecutable), se analizarán detenidamente los grafos de sintaxis de PL/0 ya vistos en la pág. 6. Comencemos por *factor*:



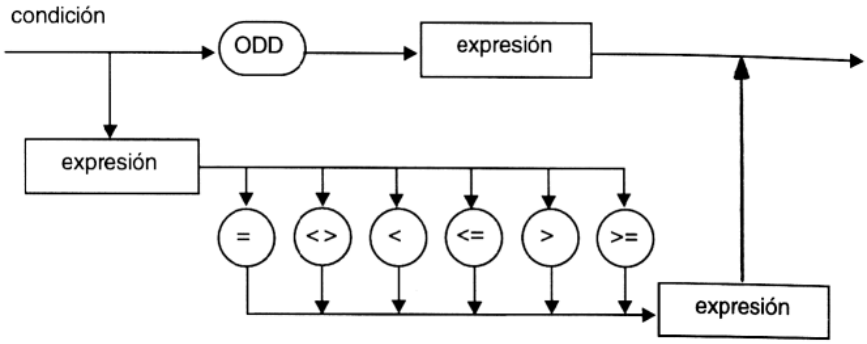
La instrucción MOV debe completarse con los cuatro bytes correspondientes a un valor numérico (si el identificador se refiere a una constante o si en el código fuente aparece directamente un número) o a un desplazamiento en memoria relativo al valor contenido en EDI (si el identificador se refiere a una variable). Como son dos instrucciones diferentes, deben usarse bytes diferentes (B8 y 8B 87, respectivamente).

En *término* y *expresión*, deben grabarse las siguientes instrucciones:





En *condición* deben generarse instrucciones para que, según una expresión (la que aparece luego de ODD) o dos expresiones (las que están relacionadas mediante los operadores lógicos) se ejecuten o se salteen las instrucciones generadas por la *proposición* que siempre viene a continuación (*condición* solamente es llamado desde *if* y desde *while*).



Los bytes generados luego de la *expresión* que sucede a ODD son:
58 A8 01 7B 05 E9 00 00 00 00.

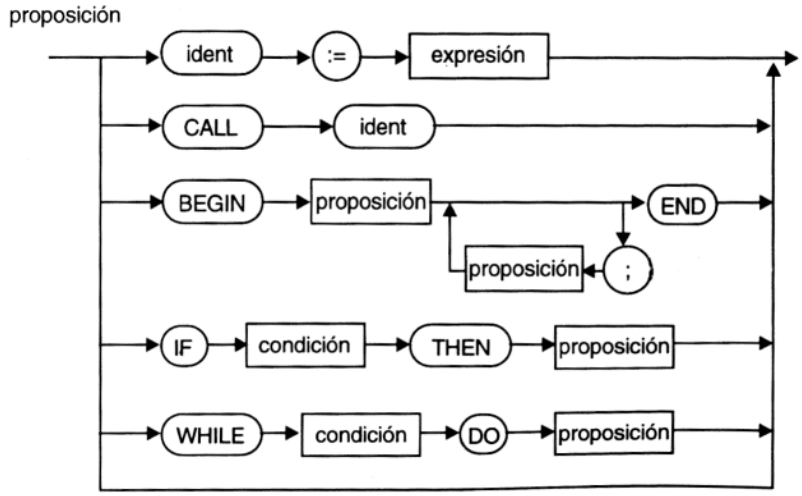
Las instrucciones generadas luego de la segunda *expresión* en la parte inferior del grafo son todas iguales, salvo por un salto condicional (de 2 bytes) que es específico del operador booleano:
58 5B 39 C3

=	<>	<	<=	>	>=
74 05	75 05	7C 05	7E 05	7F 05	7D 05

E9 00 00 00 00

Como no se conoce de antemano la cantidad de bytes que deben saltearse, se genera un salto E9 00 00 00 00 "para reservar el lugar". Luego de generar las instrucciones de la *proposición*, se debe volver atrás para corregir el destino del salto. Esto se conoce como "fix-up".

Veamos ahora los distintos casos de *proposición*:



En la asignación (y también en *READLN*), el valor del registro EAX (traído con POP EAX de la pila donde fue depositado por *expresión* o ingresado desde el teclado mediante una invocación a la rutina de entrada de enteros usando una instrucción CALL) debe copiarse a la posición de memoria correspondiente a la variable representada por el identificador.

En *CALL* debe generarse una instrucción homónima basada en la dirección de memoria del procedimiento que está siendo llamado, por ejemplo: E8 56 FF FF FF. Cabe aclarar que FFFFFF56 indica la cantidad de bytes a saltar (aquí se trata de un salto hacia atrás, por ser FFFFFF56 un número negativo), no la dirección absoluta del procedimiento.

La proposición *IF* no genera instrucciones, simplemente realiza el fix-up del salto generado por *condición*.

La proposición *WHILE* coloca un salto hacia arriba (para volver a evaluar la condición) inmediatamente a continuación de las instrucciones generadas por *proposición*, para luego realizar el fix-up del salto generado por *condición*.

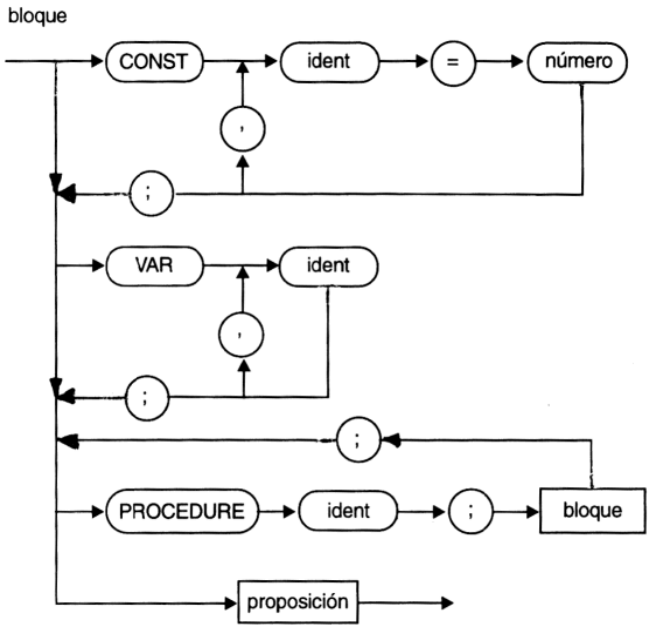
La proposiciones *WRITE* y *WRITELN* se comportan de dos formas diferentes, según se utilicen para imprimir resultados de expresiones o cadenas.

Los resultados de las expresiones se sacan de la pila (con POP EAX) y se muestran llamando a la rutina de salida de números (con CALL).

El código para imprimir cadenas se produce así:

1. **Se genera la inicialización de ECX con la ubicación que tendrá la cadena (se conoce porque los pasos 2, 3 y 4 son de longitud fija), relativa al valor del campo *Address* (posiciones 193-196, o 00C1-00C4 en hexadecimal) de la sección *text* del encabezado del archivo ejecutable;**
2. **Se genera la inicialización de EDX con la longitud de la cadena;**
3. **Se genera la invocación a la rutina de E/S que mostrará la cadena;**
4. **Se genera un salto incondicional E9 00 00 00 00;**
5. **Se generan los bytes de la cadena, seguidos de un cero;**
6. **Se realiza el fix-up del salto colocado en el paso 4.**

Veamos ahora la generación del código en *bloque*:



Al ingresar a *bloque* debe insertarse un salto, que dirige la ejecución hacia la primera instrucción de la primera proposición del bloque, salteando las instrucciones generadas al traducir los procedimientos locales que pudiera haber. El fix-up de este salto debe hacerse justo antes de entrar a *proposición*.

Al salir de *bloque* en *PROCEDURE* debe generarse una instrucción RET (código C3).

La salida del programa se lleva a cabo generando un salto (no una invocación) hacia la rutina de E/S que finaliza el programa.

En este momento de la compilación, el análisis del código fuente ya finalizó, y no quedan instrucciones por grabar.

A continuación, se debe hacer un *fix-up* de la primera instrucción de la parte de longitud variable de la sección *text* (MOV EDI, 00000000), ya que el desplazamiento actual en el archivo ejecutable indica el comienzo del área de almacenamiento de las variables.

Luego, deben grabarse ceros al final del archivo ejecutable, a razón de cuatro bytes por cada variable (a esta altura de la compilación, el número de variables que fueron declaradas ya es conocido).

En este momento, es necesario ajustar los campos *FileSize* (posiciones 68-71, o 0044-0047 en hexadecimal) y *MemorySize* (posiciones 72-75, o 0048-004B en hexadecimal), colocando allí el tamaño final del archivo ejecutable.

Por último, se debe realizar el ajuste del campo *Size* del encabezado de la sección *text* (posiciones 201-204, o 00C9-00CC en hexadecimal), colocando allí el tamaño de la sección *text*.

A modo de ejemplo, consideremos el siguiente programa en PL/0:

CÓDIGO FUENTE	CÓDIGO TRADUCIDO CARGADO EN MEMORIA

var X, Y;
 	
procedure INICIAR;
  const Y = 2;
  procedure ASIGNAR;
  X := Y;
  call ASIGNAR;
 	
begin
  write ('NUM=');
  readln (Y);
  call INICIAR;
  writeln ('NUM*2=',Y*X)
end.
	08048480 BF 02 85 04 08 MOV EDI,08048502
	08048485 E9 17 00 00 00 JMP 080484A1
	0804848A E9 0C 00 00 00 JMP 0804849B
	0804848F B8 02 00 00 00 MOV EAX,00000002
	08048494 89 87 00 00 00 00 MOV [EDI+00000000],EAX
	0804849A C3 RET
	0804849B E8 EF FF FF FF CALL 0804848F
	080484A0 C3 RET
	080484A1 B9 B5 84 04 08 MOV ECX,080484B5
	080484A6 BA 04 00 00 00 MOV EDX,00000004
	080484AB E8 C0 FC FF FF CALL 08048170
	080484B0 E9 04 00 00 00 JMP 080484B9
	080484B5 4E 55 4D 3D ASCII "NUM="
	080484B9 E8 52 FE FF FF CALL 08048310
	080484BE 89 87 04 00 00 00 MOV [EDI+00000004],EAX
	080484C4 E8 C1 FF FF FF CALL 0804848A
	080484C9 B9 DD 84 04 08 MOV ECX,080484DD
	080484CE BA 06 00 00 00 MOV EDX,6
	080484D3 E8 98 FC FF FF CALL 08048170
	080484D8 E9 06 00 00 00 JMP 080484E3
	080484DD 4E 55 4D 2A 32 3D ASCII "NUM*2="
	080484E3 8B 87 04 00 00 00 MOV EAX,[EDI+00000004]
	080484E9 50 PUSH EAX
	080484EA 8B 87 00 00 00 00 MOV EAX,[EDI+00000000]
	080484F0 5B POP EBX
	080484F1 F7 EB IMUL EBX
	080484F3 E8 98 FC FF FF CALL 08048190
	080484F8 E8 83 FC FF FF CALL 08048180
	080484FD E9 FE FD FF FF JMP 08048300

El archivo ejecutable completo (tamaño: 1290 bytes) es el siguiente:

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00   .ELF............
00000010   02 00 03 00 01 00 00 00  80 84 04 08 34 00 00 00   ............4...
00000020   65 00 00 00 00 00 00 00  34 00 20 00 01 00 28 00   e.......4. ...(.
00000030   03 00 01 00 01 00 00 00  00 00 00 00 00 80 04 08   ................
00000040   00 80 04 08 0a 05 00 00  0a 05 00 00 07 00 00 00   ................
00000050   00 10 00 00 00 2e 73 68  73 74 72 74 61 62 00 2e   ......shstrtab..
00000060   74 65 78 74 00 00 00 00  00 00 00 00 00 00 00 00   text............
00000070   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000080   00 00 00 00 00 00 00 00  00 00 00 00 00 01 00 00   ................
00000090   00 03 00 00 00 00 00 00  00 00 00 00 00 54 00 00   .............T..
000000a0   00 11 00 00 00 00 00 00  00 00 00 00 00 01 00 00   ................
000000b0   00 00 00 00 00 0b 00 00  00 01 00 00 00 06 00 00   ................
000000c0   00 e0 80 04 08 e0 00 00  00 2a 04 00 00 00 00 00   .à...à...*......
000000d0   00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ................
000000e0   52 51 53 50 b8 04 00 00  00 bb 01 00 00 00 89 e1   RQSP¸....».....á
000000f0   ba 01 00 00 00 cd 80 58  5b 59 5a c3 55 89 e5 81   º....Í.X[YZÃU.å.
00000100   ec 24 00 00 00 52 51 53  b8 36 00 00 00 bb 00 00   ì$...RQS¸6...»..
00000110   00 00 b9 01 54 00 00 8d  55 dc cd 80 81 65 e8 f5   ..¹.T...UÜÍ..eèõ
00000120   ff ff ff b8 36 00 00 00  bb 00 00 00 00 b9 02 54   ÿÿÿ¸6...»....¹.T
00000130   00 00 8d 55 dc cd 80 31  c0 50 b8 03 00 00 00 bb   ...UÜÍ.1ÀP¸....»
00000140   00 00 00 00 89 e1 ba 01  00 00 00 cd 80 81 4d e8   .....áº....Í..Mè
00000150   0a 00 00 00 b8 36 00 00  00 bb 00 00 00 00 b9 02   ....¸6...»....¹.
00000160   54 00 00 8d 55 dc cd 80  58 5b 59 5a 89 ec 5d c3   T...UÜÍ.X[YZ.ì]Ã
00000170   b8 04 00 00 00 bb 01 00  00 00 cd 80 c3 90 90 90   ¸....»....Í.Ã...
00000180   b0 0a e8 59 ff ff ff c3  04 30 e8 51 ff ff ff c3   °.èYÿÿÿÃ.0èQÿÿÿÃ
00000190   3d 00 00 00 80 75 4e b0  2d e8 42 ff ff ff b0 02   =....uN°-èBÿÿÿ°.
000001a0   e8 e3 ff ff ff b0 01 e8  dc ff ff ff b0 04 e8 d5   èãÿÿÿ°.èÜÿÿÿ°.èÕ
000001b0   ff ff ff b0 07 e8 ce ff  ff ff b0 04 e8 c7 ff ff   ÿÿÿ°.èÎÿÿÿ°.èÇÿÿ
000001c0   ff b0 08 e8 c0 ff ff ff  b0 03 e8 b9 ff ff ff b0   ÿ°.èÀÿÿÿ°.è¹ÿÿÿ°
000001d0   06 e8 b2 ff ff ff b0 04  e8 ab ff ff ff b0 08 e8   .è²ÿÿÿ°.è«ÿÿÿ°.è
000001e0   a4 ff ff ff c3 3d 00 00  00 00 7d 0b 50 b0 2d e8   ¤ÿÿÿÃ=....}.P°-è
000001f0   ec fe ff ff 58 f7 d8 3d  0a 00 00 00 0f 8c ef 00   ìþÿÿX÷Ø=......ï.
00000200   00 00 3d 64 00 00 00 0f  8c d1 00 00 00 3d e8 03   ..=d.....Ñ...=è.
00000210   00 00 0f 8c b3 00 00 00  3d 10 27 00 00 0f 8c 95   ....³...=.'.....
00000220   00 00 00 3d a0 86 01 00  7c 7b 3d 40 42 0f 00 7c   ...= ... {=@B..
00000230   61 3d 80 96 98 00 7c 47  3d 00 e1 f5 05 7c 2d 3d   a=.... G=.áõ. -=
00000240   00 ca 9a 3b 7c 13 ba 00  00 00 00 bb 00 ca 9a 3b   .Ê.; .º....».Ê.;
00000250   f7 fb 52 e8 30 ff ff ff  58 ba 00 00 00 00 bb 00   ÷ûRè0ÿÿÿXº....».
00000260   e1 f5 05 f7 fb 52 e8 1d  ff ff ff 58 ba 00 00 00   áõ.÷ûRè.ÿÿÿXº...
00000270   00 bb 80 96 98 00 f7 fb  52 e8 0a ff ff ff 58 ba   .»....÷ûRè.ÿÿÿXº
00000280   00 00 00 00 bb 40 42 0f  00 f7 fb 52 e8 f7 fe ff   ....»@B..÷ûRè÷þÿ
00000290   ff 58 ba 00 00 00 00 bb  a0 86 01 00 f7 fb 52 e8   ÿXº....» ...÷ûRè
000002a0   e4 fe ff ff 58 ba 00 00  00 00 bb 10 27 00 00 f7   äþÿÿXº....».'..÷
000002b0   fb 52 e8 d1 fe ff ff 58  ba 00 00 00 00 bb e8 03   ûRèÑþÿÿXº....»è.
```

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

000002c0   00 00 f7 fb 52 e8 be fe  ff ff 58 ba 00 00 00 00   ..÷ûRè¾þÿÿXº....
000002d0   bb 64 00 00 00 f7 fb 52  e8 ab fe ff ff 58 ba 00   »d...÷ûRè«þÿÿXº.
000002e0   00 00 00 bb 0a 00 00 00  f7 fb 52 e8 98 fe ff ff   ...»....÷ûRè.þÿÿ
000002f0   58 e8 92 fe ff ff c3 90  90 90 90 90 90 90 90 90   Xè.þÿÿÃ.........
00000300   b8 01 00 00 00 bb 00 00  00 00 cd 80 90 90 90 90   ¸....»....Í.....
00000310   b9 00 00 00 00 b3 03 51  53 e8 de fd ff ff 5b 59   ¹....³.QSèÞýÿÿ[Y
00000320   3c 0a 0f 84 34 01 00 00  3c 7f 0f 84 94 00 00 00   <...4...<.......
00000330   3c 2d 0f 84 09 01 00 00  3c 30 7c db 3c 39 7f d7   <-......<0 Û<9.×
00000340   2c 30 80 fb 00 74 d0 80  fb 02 75 0c 81 f9 00 00   ,0.û.tÐ.û.u..ù..
00000350   00 00 75 04 3c 00 74 bf  80 fb 03 75 0a 3c 00 75   ..u.<.t¿.û.u.<.u
00000360   04 b3 00 eb 02 b3 01 81  f9 cc cc cc 0c 7f a8 81   .³.ë.³..ùÌÌÌ..¨.
00000370   f9 34 33 33 f3 7c a0 88  c7 b8 0a 00 00 00 f7 e9   ù433ó .Ç¸....÷é
00000380   3d 08 00 00 80 74 11 3d  f8 ff ff 7f 75 13 80 ff   =....t.=øÿÿ.u..ÿ
00000390   07 7e 0e e9 7f ff ff ff  80 ff 08 0f 8f 76 ff ff   .~.é.ÿÿÿ.ÿ...vÿÿ
000003a0   ff b9 00 00 00 00 88 f9  80 fb 02 74 04 01 c1 eb   ÿ¹.....ù.û.t..Áë
000003b0   03 29 c8 91 88 f8 51 53  e8 cb fd ff ff 5b 59 e9   .)È..øQSèËýÿÿ[Yé
000003c0   53 ff ff ff 80 fb 03 0f  84 4a ff ff ff 51 53 b0   Sÿÿÿ.û...Jÿÿÿ QS°
000003d0   08 e8 0a fd ff ff b0 20  e8 03 fd ff ff b0 08 e8   .è.ýÿÿ° è.ýÿÿ°.è
000003e0   fc fc ff ff 5b 59 80 fb  00 75 07 b3 03 e9 25 ff   üüÿÿ[Y.û.u.³.é%ÿ
000003f0   ff ff 80 fb 02 75 0f 81  f9 00 00 00 00 75 07 b3   ÿÿ.û.u..ù....u.³
00000400   03 e9 11 ff ff ff 89 c8  b9 0a 00 00 00 ba 00 00   .é.ÿÿÿ.È¹....º..
00000410   00 00 3d 00 00 00 00 7d  08 f7 d8 f7 f9 f7 d8 eb   ..=....}.÷Ø÷ù÷Øë
00000420   02 f7 f9 89 c1 81 f9 00  00 00 00 0f 85 e6 fe ff   .÷ù.Á.ù......æþÿ
00000430   ff 80 fb 02 0f 84 dd fe  ff ff b3 03 e9 d6 fe ff   ÿ.û...Ýþÿÿ³.éÖþÿ
00000440   ff 80 fb 03 0f 85 cd fe  ff ff b0 2d 51 53 e8 8d   ÿ.û...Íþÿÿ°-QSè.
00000450   fc ff ff 5b 59 b3 02 e9  bb fe ff ff 80 fb 03 0f   üÿÿ[Y³.é»þÿÿ.û..
00000460   84 b2 fe ff ff 80 fb 02  75 0c 81 f9 00 00 00 00   .²þÿÿ.û.u..ù....
00000470   0f 84 a1 fe ff ff 51 e8  04 fd ff ff 59 89 c8 c3   ..¡þÿÿQè.ýÿÿY.ÈÃ
00000480   bf 02 85 04 08 e9 17 00  00 00 e9 0c 00 00 00 b8   ¿....é....é....¸
00000490   02 00 00 00 89 87 00 00  00 00 c3 e8 ef ff ff ff   ..........Ãèïÿÿÿ
000004a0   c3 b9 b5 84 04 08 ba 04  00 00 00 e8 c0 fc ff ff   Ã¹µ...º....èÀüÿÿ
000004b0   e9 04 00 00 00 4e 55 4d  3d e8 52 fe ff ff 89 87   é....NUM=èRþÿÿ..
000004c0   04 00 00 00 e8 c1 ff ff  ff b9 dd 84 04 08 ba 06   ....èÁÿÿÿ¹Ý...º.
000004d0   00 00 00 e8 98 fc ff ff  e9 06 00 00 00 4e 55 4d   ...è.üÿÿé....NUM
000004e0   2a 32 3d 8b 87 04 00 00  00 50 8b 87 00 00 00 00   *2=......P......
000004f0   5b f7 eb e8 98 fc ff ff  e8 83 fc ff ff e9 fe fd   [÷ëè.üÿÿè.üÿÿéþý
00000500   ff ff 00 00 00 00 00 00  00 00                     ÿÿ........
```

Como puede observarse comparando ambos listados, una vez que se ha cargado el programa en la memoria, las instrucciones de salto y las llamadas a subrutinas se ajustan automáticamente según los valores de las direcciones donde estén alojadas, a diferencia de los valores que se cargan en ECX y EDI para apuntar a las cadenas o las variables enteras, respectivamente, ya que éstos representan direcciones absolutas.

8. Optimización de código

La generación de código óptimo es un problema NP-completo y, por lo tanto, los llamados compiladores optimizadores por lo general producen código de alta calidad aunque no necesariamente óptimo. Al hablar de técnicas de optimización, hay que señalar que la optimización normalmente aumenta el tiempo de compilación. Por ello, el usuario muchas veces tiene la posibilidad de desactivar la parte de optimización del generador de código durante la fase de desarrollo o depuración de programas.

El código puede optimizarse en función de:

- ❒ reducir el tamaño de un programa, o
- ❒ aumentar la velocidad de ejecución de un programa.

La reducción del tamaño de un programa ya no es tan importante, gracias a la disponibilidad a precios razonables de memorias de alta capacidad, pero la optimización de la velocidad de ejecución sigue siendo de interés vital.

Las técnicas de optimización se basan en un extenso análisis de la estructura del programa y del flujo de datos. Durante la optimización suele subdividirse el programa en regiones de optimización, y las técnicas empleadas pueden categorizarse como independientes de la máquina (técnicas de carácter general) y dependientes de la máquina (técnicas para las cuales debe conocerse el hardware, ya que afectan la asignación de registros o la selección de instrucciones). Entre las técnicas de optimización están las siguientes:

8.a) Cálculo previo de constantes

Cuando aparecen varias constantes en una expresión aritmética, puede ser posible combinarlas, en el momento de la compilación, para formar una sola constante. Por ejemplo:

```
i := mayor - menor + 5
```

donde mayor = 10 y menor = 1, se puede reemplazar por:

```
i := 14
```

El efecto de esta técnica es que el código generado requiere menos memoria (porque sólo hay que almacenar una constante, en lugar de tres) y que aumenta la velocidad de ejecución del programa (porque las operaciones aritméticas se llevan a cabo durante la compilación)

8.b) Reducción de fuerza

Es el proceso por el cual una operación costosa (en términos de tiempo de ejecución) se reemplaza por una más barata. Por ejemplo, para colocar el valor cero en el registro EAX, puede preferirse la instrucción XOR EAX,EAX en lugar de MOV EAX,00000000:

Inst.	Operandos	Bytes
MOV	registro, inmediato	5
XOR	registro, registro	2

```
0000  B8 00 00 00 00   MOV EAX,00000000
0005  31 C0            XOR EAX,EAX
```

La instrucción XOR EAX,EAX no solamente se ejecuta más rápido (por no usar direccionamiento inmediato), sino que además ocupa tres bytes menos que la instrucción MOV EAX,00000000.

En el siguiente ejemplo puede verse cómo una misma instrucción puede ocupar más o menos memoria. Al diseñar el compilador, esto se debe tener en cuenta.

```
0000  A1 78 56 34 12         MOV EAX,[12345678]
0005  8B 05 78 56 34 12      MOV EAX,[12345678]
000B  8B 1D 78 56 34 12      MOV EBX,[12345678]
0011  8B 0D 78 56 34 12      MOV ECX,[12345678]
0007  8B 15 78 56 34 12      MOV EDX,[12345678]
```

Otro ejemplo de reducción de fuerza es el reemplazo de multiplicaciones por potencias de dos (x*2, x*4, x*8, etc.), que pueden expresarse mediante desplazamientos aritméticos (SHR y SHL).

8.c) Reducción de frecuencia

Si un ciclo contiene cálculos que producen el mismo resultado cada vez que se ejecuta el ciclo, es posible hacer los cálculos antes de entrar al ciclo (introduciendo variables temporales, en ciertas circunstancias). Por ejemplo, en:

```
i := 0; x := 3; y := 5;
while i < 1000 do
  begin
    writeln (i+x*y);
    i := i + 1
  end;
```

se realizan 1000 multiplicaciones con el mismo resultado: 15

8.d) Optimización de ciclos

Además de la reducción de frecuencia, pueden optimizarse ciclos combinando una secuencia de ciclos con idénticos intervalos con el objetivo de generar un único ciclo, por ejemplo:

```
i := 0;
while i < 1000 do
  begin
    x := x + 2 * i;
    i := i + 1
  end;
j := 0;
while j < 1000 do
  begin
    y := y + 3 * j;
    j := j + 1
  end;
```

El programa anterior es equivalente al siguiente:

```
i := 0;
while i < 1000 do
  begin
    x := x + 2 * i;
    y := y + 3 * i;
    i := i + 1
  end;
```

8.e) Eliminación de código redundante

Aquellas secciones de código que nunca se ejecutan pueden suprimirse, por ejemplo:

```
i := 0;
while i > 0 do
  begin
    ...
  end;
```

Igualmente pueden suprimirse los términos que valgan cero en una suma, así como también la multiplicación por 1. La multiplicación por cero puede reemplazarse por una instrucción más barata.

8.f) Optimización local

Examinando grupos de instrucciones (el área local), pueden realizarse varias optimizaciones, como por ejemplo:

```
MOV EAX, EBX
MOV EBX, EAX
```

equivale a:

```
MOV EAX, EBX
```

9. Manejo de errores

Un compilador es un sistema que en la mayoría de los casos tiene que manejar una entrada incorrecta. Sobre todo en las primeras etapas de la creación de un programa, es probable que el compilador se utilizará para efectuar las características que debería proporcionar un buen sistema de edición dirigido por la sintaxis, es decir, para determinar si las variables han sido declaradas antes de usarlas, o si faltan corchetes o algo así. Por lo tanto, el manejo de errores es parte importante de un compilador y el escritor del compilador siempre debe tener esto presente durante su diseño.

Hay que señalar que los posibles errores ya deben estar considerados al diseñar un lenguaje de programación. Por ejemplo, considerar si cada proposición del lenguaje de programación comienza con una palabra clave diferente (excepto la proposición de asignación, por supuesto). Sin embargo, es indispensable lo siguiente:

- **el compilador debe ser capaz de detectar errores en la entrada;**
- **el compilador debe recuperarse de los errores sin perder demasiada información;**
- **y sobre todo, el compilador debe producir un mensaje de error que permita al programador encontrar y corregir fácilmente los elementos (sintácticamente) incorrectos de su programa.**

Los mensajes de error de la forma

```
*** Error 111 ***
*** Falta declaración ***
*** Falta delimitador ***
```

no son útiles para el programador y no deben presentarse en un ambiente de compilación amigable y bien diseñado.

Por ejemplo, el mensaje de error

```
*** Falta declaración ***
```

podría reemplazarse por

```
*** No se ha declarado la variable Nombre ***
```

o en el caso del delimitador omitido se puede especificar cuál es el delimitador esperado.

Además de estos mensajes de error informativos, es deseable que el compilador produzca una lista con el código fuente e indique en ese listado dónde han ocurrido los errores.

No obstante, antes de considerar el manejo de errores en el análisis léxico y sintáctico, hay que caracterizar y clasificar los errores posibles. Esta clasificación nos mostrará que un compilador no puede detectar todos los tipos de errores.

9.a) Clasificación de errores

Durante un proceso de resolución de problemas existen varias formas en que pueden surgir errores, las cuales se reflejan en el código fuente del programa. Desde el punto de vista del compilador, los errores se pueden dividir en dos categorías:

- **errores visibles y**
- **errores invisibles.**

Los errores invisibles en un programa son aquellos que no puede detectar el compilador, ya que no son el resultado de un uso incorrecto del lenguaje de programación, sino de decisiones erróneas durante el proceso de especificación o de la mala formulación de algoritmos. Por ejemplo, si se escribe

```
a := b + c;   en lugar de   a := b * c;
```

el error no podrá ser detectado por el compilador ni por el sistema de ejecución. Estos errores lógicos no afectan la validez del programa en cuanto a su corrección sintáctica. Son objeto de técnicas formales de verificación de programas que no se consideran aquí.

Los errores visibles, a diferencia de los errores lógicos, pueden ser detectados por el compilador o al menos por el sistema de ejecución. Estos errores se pueden caracterizar de la siguiente manera:

- **errores de ortografía y**
- **errores que ocurren por omitir requisitos formales del lenguaje de programación.**

Estos errores se presentará porque los programadores no tienen el cuidado suficiente al programar. Los errores del segundo tipo también pueden ocurrir porque el programador no comprende a la perfección el lenguaje que utiliza o porque suele escribir sus programas en otro lenguaje y, por tanto, emplea las construcciones de dicho lenguaje (estos

problemas pueden presentarse al usar a la vez lenguajes de programación como PASCAL y MODULA-2, por ejemplo).

Clasificación de ocurrencias

Por lo regular, los errores visibles o detectables por el compilador se dividen en tres clases, dependiendo de la fase del compilador en la cual se detectan:

- errores léxicos;
- errores sintácticos;
- errores semánticos.

Por ejemplo, un error léxico puede ocasionarse por usar un carácter inválido (uno que no pertenezca al vocabulario del lenguaje de programación) o por tratar de reconocer una constante que produce un desbordamiento.

Un error de sintaxis se detecta cuando el analizador sintáctico espera un símbolo que no corresponde al que se acaba de leer. Los analizadores sintácticos LL y LR tienen la ventaja de que pueden detectar errores sintácticos lo más pronto posible, es decir, se genera un mensaje de error en cuanto el símbolo analizado no sigue la secuencia de los símbolos analizados hasta ese momento.

Los errores semánticos corresponden a la semántica del lenguaje de programación, la cual normalmente no está descrita por la gramática. Los errores semánticos más comunes son la omisión de declaraciones.

Además de estas tres clases de errores, hay otros que serán detectados por el sistema de ejecución porque el compilador ha proporcionado el código generado con ciertas acciones para estos casos. Un error de ejecución típico ocurre cuando el índice de una matriz no es un elemento del subintervalo especificado o por intentar una división por cero. En tales situaciones, se informa del error y se detiene la ejecución del programa.

Clasificación estadística

D. G. Ripley y F. C. Druseikis investigaron los errores que cometen los programadores de PASCAL y analizaron los resultados en relación con las estrategias de recuperación. El resultado principal del estudio fue

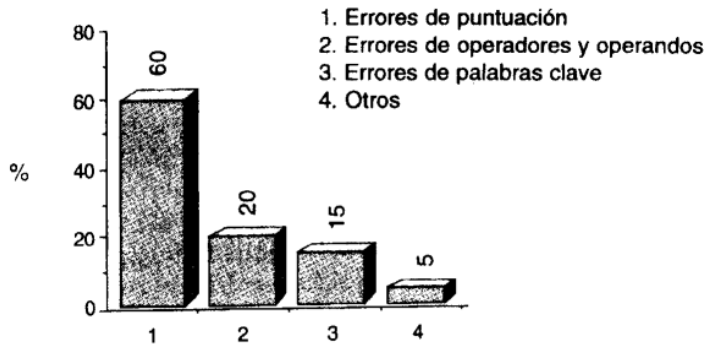
la verificación de que los errores de sintaxis suelen ser muy simples y que, por lo general, sólo ocurre un error por frase. En el resumen siguiente se describen de manera general los resultados del estudio:

- Al menos el 40% de los programas compilados eran sintáctica o semánticamente incorrectos.
- Un 80% de las proposiciones incorrectas sólo tenían un error.
- El 13% de las proposiciones incorrectas tenían dos errores, menos del 3% tenían tres errores y el resto tenían cuatro o más errores por proposición.
- En aproximadamente la mitad de los errores de componentes léxicos olvidados, el elemento que faltaba era ":", mientras que omitir el "END" final ocupaba el segundo lugar, con un 10,5%.
- En un 13% de los errores de componente léxico incorrecto se escribió "," en lugar de ";" y en más del 9% de los casos se escribió ":=" en lugar de "=".

Los errores que ocurren pueden clasificarse en cuatro categorías:

- errores de puntuación,
- errores de operadores y operandos,
- errores de palabras clave, y
- otros tipos de errores.

La distribución estadística de estas cuatro categorías aparece en la siguiente figura:



Distribución estadística de las categorías de errores

9.b) Efectos de los errores

La detección de un error en el código fuente ocasiona ciertas reacciones del compilador.

El comportamiento de un compilador en el caso de que el código fuente contenga un error puede tener varias facetas:

- **El proceso de compilación se detiene al ocurrir el error y el compilador debe informar del error.**
- **El proceso de compilación continúa cuando ocurre el error y se informa de éste en un archivo de listado.**
- **El compilador no reconoce el error y por tanto no advierte al programador.**

La última situación nunca debe presentarse en un buen sistema de compilación; es decir, el compilador debe ser capaz de detectar todos los errores visibles.

La detención del proceso de compilación al detectar el primer error es la forma más simple de satisfacer el requisito de que una compilación siempre debe terminar sin importar cuál sea la entrada. Sin embargo, este comportamiento también es el peor en un ambiente amigable para el usuario, ya que una compilación puede demorar algún tiempo. Por lo tanto, el programador espera que el sistema de compilación detecte todos los errores posibles en el mismo proceso de compilación.

Entonces, en general, el compilador debe recuperarse de un error para poder revisar el código fuente en busca de otros errores. No obstante, hay que observar que cualquier "reparación" efectuada por el compilador tiene el propósito único de continuar la búsqueda de otros errores, no de corregir el código fuente. No hay reglas generales bien definidas acerca de cómo recuperarse de un error, por lo cual el proceso de recuperación debe basarse en hipótesis acerca de los errores. La carencia de tales reglas se debe al hecho de que el proceso de recuperación siempre depende del lenguaje.

9.c) Manejo de errores en el análisis léxico

Los errores léxicos se detectan cuando el analizador léxico intenta reconocer componentes léxicos en el código fuente. Los errores léxicos típicos son:

- **nombres ilegales de identificadores: un nombre contiene caracteres inválidos;**
- **números inválidos: un número contiene caracteres inválidos (por ejemplo, 2,13 en lugar de 2.13), no está formado correctamente (por ejemplo, 0.1.33), o es demasiado grande y por tanto produce un desbordamiento;**
- **cadenas incorrectas de caracteres: una cadena de caracteres es demasiado larga (probablemente por la omisión de comillas que cierran);**
- **errores de ortografía en palabras reservadas: caracteres omitidos, adicionales, incorrectos o mezclados;**
- **fin de archivo: se detecta un fin de archivo a la mitad de un componente léxico.**

La mayoría de los errores léxicos se deben a descuidos del programador. En general, la recuperación de los errores léxicos es relativamente sencilla.

Si un nombre, un número o una etiqueta contiene un carácter inválido, se elimina el carácter y continúa el análisis en el siguiente carácter; en otras palabras, el analizador léxico comienza a reconocer el siguiente componente léxico. El efecto es la generación de un error de sintaxis que será detectado por el analizador sintáctico. Este método también puede aplicarse a números mal formados.

Las secuencias de caracteres como 12AB pueden ocurrir si falta un operador (el caso menos probable) o cuando se han tecleado mal ciertos caracteres. Es imposible que el analizador léxico pueda decidir si esta secuencia es un identificador ilegal o un número ilegal. En tales casos, el analizador léxico puede saltarse la cadena completa o intentar dividir las secuencias ilegales en secuencias legales más cortas. Independientemente de cuál sea la decisión, la consecuencia será un error de sintaxis.

La detección de cadenas demasiado largas no es muy complicada, incluso si faltan las comillas que cierran, porque por lo general no está permitido que las cadenas pasen de una línea a la siguiente. Si faltan las comillas que cierran, puede usarse el carácter de fin de línea como el fin de la cadena y reanudar el análisis léxico en la línea siguiente. Esta reparación quizás produzca errores adicionales. En cualquier caso, el programador debe ser informado por medio de un mensaje de error.

Un caso similar a la falta de comillas que cierran en una cadena, es la falta de un símbolo de terminación de comentario. Como por lo regular está permitido que los comentarios abarquen varias líneas, no podrá detectarse la falta del símbolo que cierra el comentario hasta que el analizador léxico llegue al final del archivo o al símbolo de fin de otro comentario (si no se permiten comentarios anidados).

Si se sabe que el siguiente componente léxico debe ser una palabra reservada, es posible corregir una palabra reservada mal escrita. Esto se hace mediante funciones de corrección de errores, aplicando una función de distancia métrica entre la entrada y el conjunto de palabras reservadas.

Por último, el proceso de compilación puede terminar si se detecta un fin de archivo dentro de un componente léxico.

9.d) Manejo de errores en el análisis sintáctico

El analizador sintáctico detecta un error de sintaxis cuando el analizador léxico proporciona el siguiente símbolo y éste es incompatible con el estado actual del analizador sintáctico. Los errores sintácticos típicos son:

- paréntesis o corchetes omitidos, por ejemplo, `x := y * (1 + z;`
- operadores u operandos omitidos, por ejemplo, `x := y (1 + z);`
- delimitadores omitidos, por ejemplo, `x := y IF a > x THEN y := z;`

No hay estrategias de recuperación de errores cuya validez sea general, y la mayoría de las estrategias conocidas son heurísticas, ya que se basan en suposiciones acerca de cómo pueden ocurrir los errores y lo que probablemente quiso decir el programador con una determinada construcción. Sin embargo, hay algunas estrategias que gozan de amplia aceptación:

- Recuperación de emergencia (o en modo pánico): Al detectar un error, el analizador sintáctico salta todos los símbolos de entrada hasta encontar un símbolo que pertenezca a un conjunto previamente definido de símbolos de sincronización. Estos símbolos de sincronización son el punto y coma, el símbolo *end* o cualquier palabra clave que pueda ser el inicio de una proposición nueva, por ejemplo. Es fácil implantar la recuperación de emergencia, pero sólo reconoce un error por proposición. Esto no necesariamente es una desventaja, ya que no es muy

probable que ocurran varios errores en la misma proposición. Esta suposición es un ejemplo típico del carácter heurístico de esta estrategia.

- Recuperación por inserción, borrado y reemplazo: Éste también es un método fácil de implantar y funciona bien en ciertos casos de error. Usemos como ejemplo una declaración de variable en PASCAL. Cuando una coma va seguida por dos puntos, en lugar de un nombre de variable, es posible eliminar esta coma.
- Recuperación por expansión de gramática: el 60% de los errores en los programas fuente son errores de puntuación, por ejemplo, la escritura de un punto y coma en lugar de una coma, o viceversa. Una forma de recuperarse de estos errores es legalizarlos en ciertos casos, introduciendo lo que llamaremos producciones de error en la gramática del lenguaje de programación. La expansión de la gramática con estas producciones no quiere decir que ciertos errores no serán detectados, ya que pueden incluirse acciones para informar de su detección.

La recuperación de emergencia es la estrategia que se encontrará en la mayoría de los compiladores, pero también la legalización de ciertos errores mediante la definición de una gramática aumentada es una técnica que se emplea con frecuencia. No obstante, hay que expandir la gramática con mucho cuidado para asegurarse de que no cambien el tipo y las características de la gramática.

9.e) Errores semánticos

Los errores que puede detectar el analizador sintáctico son aquellos que violan las reglas de una gramática independiente del contexto. Algunas de las características de un lenguaje de programación no pueden enunciarse con reglas independientes del contexto, ya que dependen de él; por ejemplo, la restricción de que los identificadores deben declararse previamente. Por lo tanto, los principales errores semánticos son:

- identificadores no definidos;
- operadores y operandos incompatibles.

Es mucho más difícil introducir métodos formales para la recuperación de errores semánticos que para la recuperación de errores sintácticos, ya que a menudo la recuperación de errores semánticos es *ad hoc*. No

obstante, puede requerirse que, por lo menos, el error semántico sea informado al programador y que se suprima la generación de código.

Sin embargo, la mayoría de los errores semánticos pueden ser detectados mediante la revisión de la tabla de símbolos. Si se detecta un identificador no definido, es conveniente insertar el identificador en la tabla de símbolos, suponiendo un tipo que se base en el contexto donde ocurra o un tipo universal que permita al identificador ser un operando de cualquier operador del lenguaje. Al hacerlo, evitamos la producción de un mensaje de error cada vez que se use la variable no definida. Si el tipo de un operando no concuerda con los requisitos de tipo del operador, también es conveniente reemplazar el operando con una variable ficticia de tipo universal.

10. Bibliografía

Aho, A. et al.: "Compiladores. Principios, técnicas y herramientas", Pearson, 2da. Edición, 2008

Louden, K.: "Construcción de compiladores. Principios y prácticas", Thomson International, México, 2004

Teufel, B. et al.: "Compiladores. Conceptos Fundamentales", Addison-Wesley Iberoamericana, México, 1995

TIS Committee: "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification – Version 1.2 – May 1995", en: http://refspecs.freestandards.org/elf/elf.pdf

Wirth, N.: "Algorithms + Data Structures = Programs", Prentice-Hall, Englewood Cliffs, 1976

11. Otros recursos sugeridos

***GNU/Linux* incluye las herramientas *readelf*, *hexdump* y *objdump*. Además, pueden ser útiles los siguientes programas:**

***EDB (Evan's Debugger)*: http://www.codef00.com/projects.php#debugger**
***HT Editor*: http://hte.sourceforge.net**
***Editor hexadecimal Okteta*: http://utils.kde.org/projects/okteta**