PRÁCTICA DE COMPILACIÓN

Implementación de un compilador

Víctor Ruiz-Clavijo Jimeno

Imanol Conde González

Aitor González González

Melisa Fernández Rámila

15/05/2022

ÍNDICE

INTRODUCCIÓN

Hemos construido el front-end de un compilador utilizando la técnica de constructor de traductores ascendente, a partir de un esquema de traducción dirigida por la sintaxis o ETDS. El lenguaje de entrada al compilador es un lenguaje de alto nivel Inventado, y el de salida un código de tres direcciones. La implementación del traductor se basa en C++, Flex y Bison.

En nuestra práctica hemos realizado la parte básica y obligatoria de traductor que incluye declaración de procedimientos, while else, if else, sentencias de tipo break if ademas de operaciones de lectura y escritura read y println. Ademas de esto he implementado la mejora de los booleanos con expresiones AND, OR y NOT.

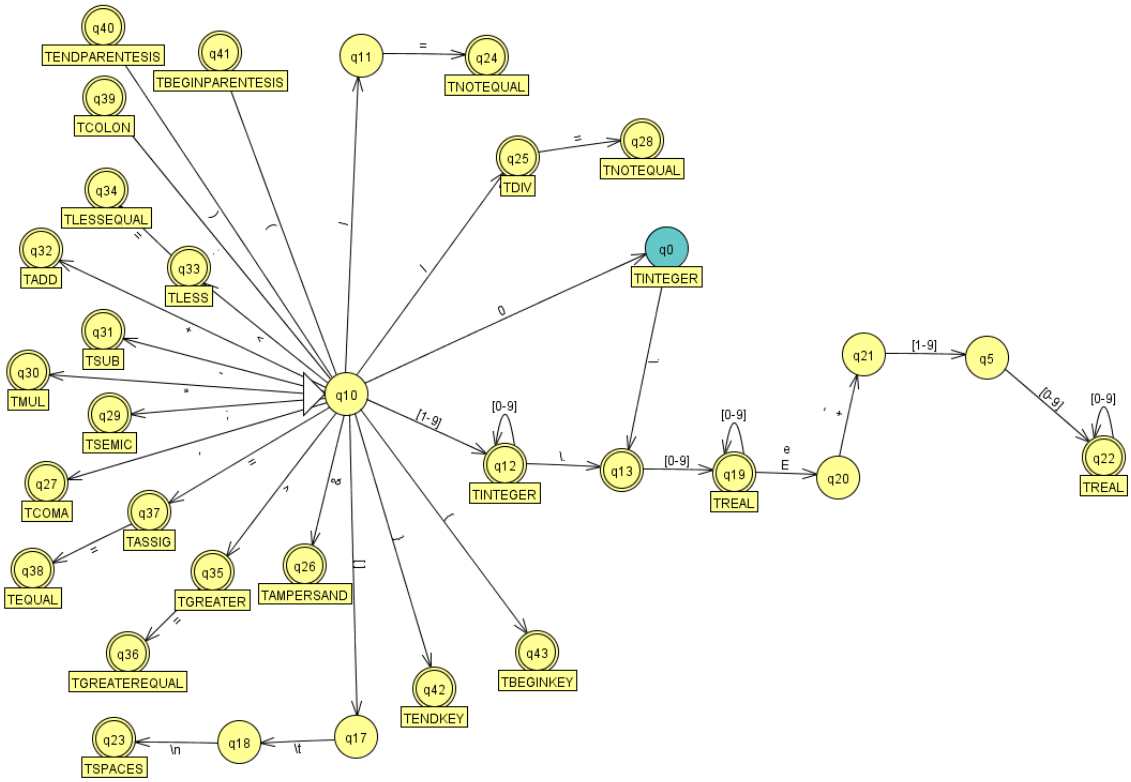
Durante la práctica hemos aprendido mucho sobre el lenguaje de alto nivel c++ y punteros ademas nos ha sido util para manejarnos cada vez mejor con linux y la compilación de ficheros con makefiles. Hemos aprendido en definitiva como funciona un compilador por dentro.

ANÁLISIS LÉXICO

```
digits [1-9][0-9]*
l [a-zA-Z]
d [0-9]
real ({digits}|0)\.[0-9]+
double ({digits}|0)\.[0-9]*
%%
main                    TOKEN(RPROGRAM);
def                     TOKEN(RDEF);
let                     TOKEN(RLET);
integer                 TOKEN(RINTEGER);
float                   TOKEN(RFLOAT);
in                      TOKEN(RIN);
if                      TOKEN(RIF);
else                    TOKEN(RELSE);
while                   TOKEN(RWHILE);
forever                 TOKEN(RFOREVER);
break                   TOKEN(RBREAK);
continue                TOKEN(RCONTINUE);
read                    TOKEN(RREAD);
println                 TOKEN(RPRINTLN);
"{"                     TOKEN(TBEGINKEY) ;
"}"                     TOKEN(TENDKEY) ;
"("                     TOKEN(TBEGINPARENTESIS);
```

")"	TOKEN(TENDPARENTESIS);
":"	TOKEN(TCOLON);
"=="	TOKEN(TEQUAL);
"="	TOKEN(TASSIG);
">"	TOKEN(TGREATER);
"<"	TOKEN(TLESS);
">="	TOKEN(TGREATEREQUAL);
"<="	TOKEN(TLESSEQUAL);
"/="	TOKEN(TNOTEQUAL);
"+"	TOKEN(TADD);
"-"	TOKEN(TSUB);
"*"	TOKEN(TMUL);
"/"	TOKEN(TDIV);
";"	TOKEN(TSEMIC);
","	TOKEN(TCOMA);
"&"	TOKEN(TAMPERSAND);
\/\/.+\n	;
[\t\n]	;
{l}({l}|{d})*([_]({l}|{d})+)*	TOKEN(TIDENTIFIER);
{real}([Ee][+-]?{digits})?	TOKEN(TREAL);
{digits}|{d}	TOKEN(TINTEGER);
'"('|(([^'#]|'[^'#]|"[^'#])*))"'	; //Comentario multilínea
#[^\n#]*\n	; //Comentario de línea
.	printf("Unknown token!\n"); yyterminate();

AUTÓMATA


q40
TENDPARENTESIS
q41
TBEGINPARENTESIS
q39
TCOLON
q34
TLESSEQUAL
q32
TADD
q33
TLESS
q31
TSUB
q30
TMUL
q29
TSEMIC
q27
TCOMA
q37
TASSIG
q38
TEQUAL
q35
TGREATER
q36
TGREATEREQUAL
q26
TAMPERSAND
q23
TSPACES
q18
q17
q42
TENDKEY
q43
TBEGINKEY
q11
q24
TNOTEQUAL
q25
TDIV
q28
TNOTEQUAL
q0
TINTEGER
q10
[1-9]
[0-9]
q12
TINTEGER
q13
[0-9]
q19
TREAL
e
E
q20
q21
[1-9]
q5
[0-9]
[0-9]
q22
TREAL

DEFINICIÓN DE ATRIBUTOS

Símbolo	Nombre	Tipo	L/S	Descripción
id	nombre	string	L	Contiene la cadena de caracteres del id.
num_entero	nombre	string	L	Contiene la cadena de caracteres del integer.
num_real	nombre	string	L	Contiene la cadena de caracteres del float.
tipo	clase	string	S	Contiene el tipo de dato de una variable o argumento.
clase_par	nombre	string	S	Contiene el nombre de la clase de parametro que puede ser "val" si es por valor o "ref" si es por referencia.
expresion	nombre	string	S	Contiene la cadena de carácteres de expresión que puede ser el nombre de una variable de código intermedio o el nombre de una variable de código fuente.
lista_de_ident	lnom	lista de string	S	Lista de strings con los nombres de la lista de ids.
resto_lista_ident	lnom	lista de string	S	Lista de strings con los nombres de la lista de ids locales a este no terminal.
sentencia	breaks	lista de integer	S	Contiene la lista de referencias a los goto generados por los breaks que aun no han sido completados.
sentencia	continues	lista de integer	S	Contiene la lista de referencias a los goto generados por los.continues que aun no han sido completados.
expresion	trues	lista de integer	S	Contiene la referencia a las instrucciones goto del caso verdadero.
expresion	falses	lista de integer	S	Contiene la referencia a las instrucciones goto del caso falso.

variable	nombre	string	S	Contiene la cadena de caracteres de variable.
bloque	breaks	lista de integer	S	Contiene las referencias a los gotos.breakssin comletar de la lista de sentencias que contiene.
bloque	continues	lista de integer	S	Contiene las referencias a los gotos.continues sin comletar de la lista de sentencias que contiene.
M	ref	integer	S	Contiene la dirección de linea a la que hace referencia esta marca.
N	next	lista de integer	S	Contiene la referencia al goto escrito por este no terminal.

ABSTRACCIONES FUNCIONALES

añadirInstruccion: código x instrucción -> código

Descripción: Dada una estructura de código numerada y una inst (String), escribe inst en la siguiente línea de la estructura de código.

Tipo de los argumentos:

código: Estructura numerada de Strings

inst: String

añadirDeclaraciones: codigo x lista x tipo -> código

Descripción: Por cada nombre de la lista de entrada, empezando por el primero y hasta el último añade una instrucción de esta forma: tipo nombre ;. Se verificarán los tipos del lenguaje fuente integer float y se mapean a sus análogos int real de código intermedio.

Tipo de los argumentos:

lista: Estructura numerada de Strings

tipo: String

añadirParametros: código x lista x tipo x clasePar -> código

Descripción: Por cada nombre de la lista de entrada, empezando por el primero y hasta el último añade una instrucción dependiendo del tipo, val o ref, llamando a la función añadir declaraciones.

Tipo de los argumentos:

lista: Estructura numerada de Strings

tipo: String con el tipo del lenguaje fuente

clasePar: String

añadir: lista x nombre -> lista

Descripción: Añade el nombre al comienzo de la lista de entrada y devuelve la nueva lista.

Tipo de los argumentos:

lista: Estructura numerada de Strings

nombre: String

inilista: nombre -> lista

Descripción: Crea una lista con el nombre y devuelve la nueva lista.

Tipo de los argumentos:

nombre: String

lista_vacia: void -> lista

Descripción: Crea una lista vacía de enteros y la devuelve.

completarInstrucciones: código x lista x etiqueta -> codigo

Descripción: Completa con la marca etiqueta las instrucciones referenciadas por la lista.

Tipo de los argumentos:

código: Estructura numerada de strings

lista: Estructura numerada de enteros

etiqueta: String

nuevo_id: void -> identificador

Descripción: devuelve un nuevo identificador de código intermedio: _t1, _t2, _t3…

escribir: código -> fichero

Descripción: dado un array con las instrucciones de código escribe dichas instrucciones en fichero, por defecto la salida estándar.

Tipo de los argumentos:

código: Estructura numerada de strings

unir: lista1 x lista2 -> lista

Descripción: dadas dos listas con referencias a instrucciones las une en una única lista.

Tipo de los argumentos:

lista1: Estructura numerada de enteros

lista2: Estructura numerada de enteros

ETDS

programa → **def main ( ) :**

{añadirInstruccion(“proc main ;”)}

bloque_ppl

{añadirInstruccion(“halt ;”);

escribir();}

bloque_ppl → decl_bl **{** decl_de_subprogs lista_de_sentencias **}**

bloque → { lista_de_sentencias }

{bloque.breaks= lista_de_sentencias.breaks;

bloque.continues = lista_de_sentencias.continues}

decl_bl → let declaraciones in

| ε {}

declaraciones → declaraciones **;** lista_de_ident **:** tipo

{añadirDeclaraciones(lista_ident.lnom, tipo.clase)}

| lista_de_ident **:** tipo

{añadirDeclaraciones(lista_ident.lnom, tipo.clase)}

lista_de_ident → **id** resto_lista_id

{lista_de_ident.lnom := añadir(resto_lista_id.lnom, id.nombre);}

resto_lista_id → **,** **id** resto_lista_id

{resto_lista_id.lnom := añadir(resto_lista_id1.lnom,id.nombre);}

| ε {resto_lista_id.lnom := {};}

tipo → **integer** {tipo.clase := “integer”}

| float {tipo.clase := “float”}

decl_de_subprogs → decl_de_subprograma decl_de_subprogs

| ε {}

decl_de_subprograma → **def** **id**

{añadirInstruccion(“proc” + id.nombre + “ ;”)}

argumentos **:** bloque_ppl

{añadirInstruccion(“endproc” + id.nombre + “ ;”)}

argumentos → **(** lista_de_param **)**

| ε {}

lista_de_param → lista_de_ident **:** clase_par tipo

{añadirParametros(lista_de_ident.lnom,clase_par.nombre, tipo.clase)}

resto_lis_de_param

clase_par → ε {clase_par.nombre = “”}

| & {clase_par.nombre = “&”}

resto_lis_de_param → **;** lista_de_ident **:** clase_par tipo

{añadirParametros(lista_de_ident.lnom,tipo.clase,
clase_par.nombre)}

resto_lis_de_param

| ε {}

lista_de_sentencias → sentencia lista_de_sentencias

{lista_de_sentencias.breaks:=

unir(lista_de_sentencias1.break, sentencia.break);

lista_de_sentencias.continues=

unir(lista_de_sentencias1.continues,

sentencia.continues);}

| ε {lista_de_sentencias.breaks= lista_vacia();

lista_de_sentencias.continues = lista_vacia();}

sentencia → variable = expresion **;**

{añadirInstruccion(variable.nombre || := || expresion.nombre);

sentencia.breaks= lista_vacia();

sentencia.continues = lista_vacia();}

| if expresion **:** M bloque M

{completarInstrucciones(expresion.true,M1.ref);

completarInstrucciones(expresion.false, M2.ref);

sentencia.breaks= bloque.breaks;

sentencia.continues = bloque.continues;}

| **forever :** M bloque M

{completarInstrucciones(bloque.breaks, M2.ref + 1);

añadirInstruccion(goto || M1.ref);

sentencia.breaks= lista_vacia();

sentencia.continues = lista_vacia();}

| **while** M expresion **:** M bloque N **else :** M bloque M

{completarInstrucciones(expresion.true,M2.ref);

completarInstrucciones(expresion.false, M3.ref);

completarInstrucciones(bloque1.continues, M1.ref);

completarInstrucciones(bloque2.continues, M1.ref);

completarInstrucciones(bloque1.breaks, M3.ref);

completarInstrucciones(bloque2.breaks, M4.ref);

completarInstrucciones(N.next, M1.ref);

sentencia.breaks= lista_vacia();

sentencia.continues = lista_vacia();}

|break if expresion **;**

{ completarInstrucciones(expresion.false, obtenRef());

sentencia.breaks := inilista(expresion.trues);

sentencia.continues = lista_vacia();}

|continue ;

{ sentencia.continues := inilista(obtenRef());

añadirInstruccion(“goto”);

sentencia.breaks= lista_vacia();}

| **read (** variable **) ;**

{añadirInstruccion(“read” + variable.nombre + “;”)

sentencia.continues = lista_vacia();

sentencia.breaks= lista_vacia();}

| **println (** expresion **) ;**

{añadirInstruccion(“write” + expresion.nombre + “;”);

añadirInstruccion(“writeln ;”);

sentencia.continues = lista_vacia();

sentencia.breaks= lista_vacia();}

variable → **id** {variable.nombre := id.nombre}

expresion → expresion **==** expresión

{expresion.nombre = “”;

expresion.trues = inilista(obtenref());

expresion.falses = inilista(obtenref() + 1);

añadirInstruccion(if || expresion1.nombre || = ||expresion2 .nombre || goto);

añadirInstruccion(goto)}

| expresion **>** expresion

{expresion.nombre = “”;

expresion.trues = inilista(obtenref());

expresion.falses = inilista(obtenref() + 1);

añadirInstruccion(if || expresion1.nombre || > ||expresion2 .nombre ||goto);

añadirInstruccion(goto)}

| expresion **<** expresion

{expresion.nombre = “”;

expresion.trues = inilista(obtenref());

expresion.falses = inilista(obtenref() + 1);

añadirInstruccion(if || expresion1.nombre || < ||expresion2 .nombre || goto);

añadirInstruccion(goto)}

| expresion **>=** expresion

{expresion.nombre = “”;

expresion.trues = inilista(obtenref());

expresion.falses = inilista(obtenref() + 1);

añadirInstruccion(if || expresion1.nombre || >= ||expresion2 .nombre || goto);

añadirInstruccion(goto)}

| expresion **<=** expresion

{expresion.nombre = “”;

expresion.trues = inilista(obtenref());

expresion.falses = inilista(obtenref() + 1);

añadirInstruccion(if || expresion1.nombre || <= ||expresion2 .nombre || goto);

añadirInstruccion(goto)}

| expresion **/=** expresion

{expresion.nombre = “”;

expresion.trues = inilista(obtenref());

expresion.falses = inilista(obtenref() + 1);

añadirInstruccion(if || expresion1.nombre || /= ||expresion2 .nombre || goto);

añadirInstruccion(goto)}

| expresion **+** expresion

{expresion.trues = lista_vacia();

expresion.falses = lista_vacia();

expresion.nombre = nuevo_id();

añadirInstruccion(expresion.nombre || := || expresion1.nombre || +
expresion2.nombre);}

| expresion **-** expresion

{expresion.trues = lista_vacia();

expresion.falses = lista_vacia();

expresion.nombre = nuevo_id();

añadirInstruccion(expresion.nombre || := || expresion1.nombre || - ||
expresion2.nombre);}

| expresion * expresion

{expresion.trues = lista_vacia();

expresion.falses = lista_vacia();

expresion.nombre = nuevo_id();

añadirInstruccion(expresion.nombre || := || expresion1.nombre || * ||
expresion2.nombre);}

| expresion / expresion

{expresion.trues = lista_vacia();

expresion.falses = lista_vacia();

expresion.nombre = nuevo_id();

añadirInstruccion(expresion.nombre || := || expresion1.nombre || / ||
expresion2.nombre);}

| variable

{expresion.trues = lista_vacia();

expresion.falses = lista_vacia();

expresion.nombre := variable.nombre}

| num_entero

{expresion.trues = lista_vacia();

expresion.falses = lista_vacia();

expresion.nombre := num_entero.nombre}

| num_real

{expresion.trues = lista_vacia();

expresion.falses = lista_vacia();

expresion.nombre := num_real.nombre}

| **(** expresion **)**

{expresion.trues := expresion1.true;

expresion.falses := expresion1.false;

expresion.nombre := expresión1.nombre}

| **not** expresion

{expresion.trues := expresion1.falses;

expresion.falses := expresion1.trues;

expresion.nombre :=expresion1.nombre;}

| expresion **or** M expresion

{completarInstrucciones(expresion1.falses, M.ref);

expresion.nombre := “”;

expresion.trues := unir(expresion1.trues, expresion2.trues);

expresion.falses := expresion2.falses;}

| expresion **and** M expresion

{completarInstrucciones(expresion1.trues, M.ref);

expresion.nombre := “”;

expresion.trues := expresion2.trues;

expresion.falses := unir(expresion1.falses, expresion2.falses);}

M → ε {M.ref := obtenref();}

N → ε {N.next := inilista(obtenref());

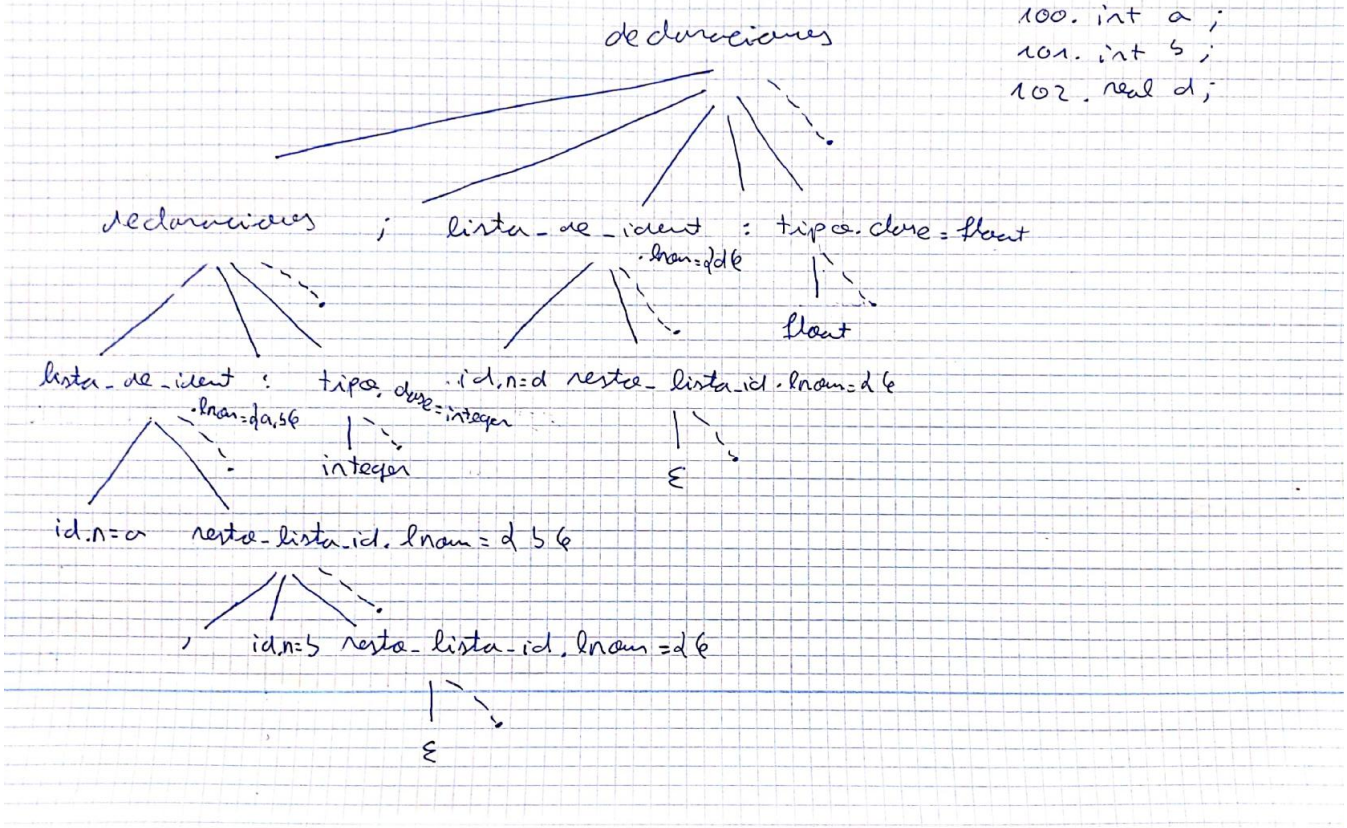
añadirInstruccion(“goto”);}

ÁRBOLES ANÁLISIS

Entrada: a, s : integer ; d : float

Salida:

100. int a ;
101. int s ;
102. real d ;

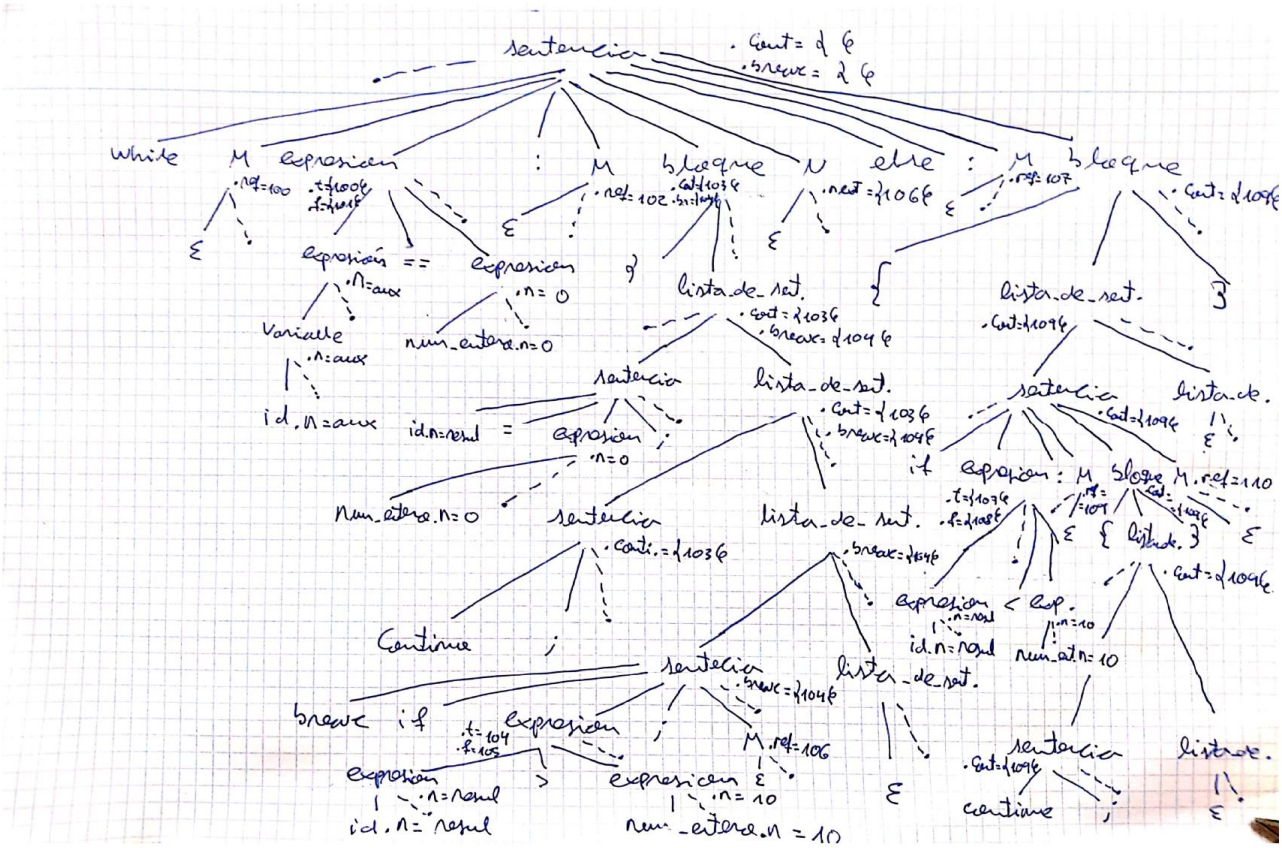


Entrada:

```
while aux == 0 : {
    resul = 0;
    continue;
    break if resul > 10;
} else : {
    if resul < 10 : { continue; }
}
```

Salida:

```
100. if aux == 0 goto 102
101. goto 107
102. resul = 0
103. goto 100
104. if resul > 10 goto 107
105. goto 106
106. goto 100
107. if resul < 10 goto 109
108. goto 110
109. goto 100
110.
112.
113.
114.
115.
```



Enlace arboles de analisis booleanos:

https://drive.google.com/file/d/1O7DQdRjPS3aSihsYF4bHWBS-Cre_eXXS/view?usp=sharing

CASOS DE PRUEBA

Entrada: PruebaBuena1

```
def main ( ):

let a,b, c : integer; d,e: float in
{
''' esto es un '
 comentario multilinea '''

    def sumar (x,y : integer; resul: & integer):
      let aux, vueltas: integer in
    {
            aux = y;
            resul = x;
            if resul < 1000:
        {
                  vueltas = 0;
                  while aux == 0:
            {
                        resul = resul + 1;
                        break if resul > 1000000;
                        aux = aux - 1;
                        vueltas = vueltas + 1;
                  }
            else:
            {
                        if resul < 0: { continue; }
                        println(vueltas);
                  } #fin del while-else
          }  #fin del if
       } # fin de sumar

   read(a); read(b);
   d = 1/b;
   e = 0.1e-1/a;
   ''' sumar(a,b,c); esto solo para aquellos que
                   traten llamadas a procedimientos '''
   c = c*(c*d)+e;
   println(c*c);
} # fin del main
```

Salida:

```
1: proc main;
2: int c;
3: int b;
4: int a;
5: real e;
6: real d;
7: proc sumar;
8: val_int y;
9: val_int x;
10: ref_int resul;
11: int vueltas;
12: int aux;
13: aux=y;
14: resul=x;
15: if resul<1000 goto 17;
16: goto 34;
17: vueltas=0;
18: if aux==0 goto 20;
19: goto 29;
20: _t1:=resul+1;
21: resul=_t1;
22: if resul>1000000 goto 29;
23: goto 24;
24: _t2:=aux-1;
25: aux=_t2;
26: _t3:=vueltas+1;
27: vueltas=_t3;
28: goto 18;
29: if resul<0 goto 31;
30: goto 32;
31: goto  18;
32: write vueltas;
33: writeln ;
34: endproc sumar;
35: read a;
36: read b;
37: _t4:=1/b;
38: d=_t4;
39: _t5:=0.1e-1/a;
40: e=_t5;
41: _t6:=c*d;
42: _t7:=_t6+e;
43: _t8:=c*_t7;
44: c=_t8;
45: _t9:=c*c;
46: write _t9;
47: writeln ;
48: halt;
```

Entrada: PruebaBuena2

```
def main ():

let a,b,c : integer; d,e : float in
{
''' esto es un comentario '''
    def sumar (x,y: integer; resul: & integer):
    let aux, vueltas: integer in
    {
        aux = y; resul = x;
        if resul < 1000:
        {
            vueltas = 0;
            while aux == 0:
            {
                resul = resul + 1;
                break if resul > 100000;
                aux = aux - 1;
                vueltas = vueltas + 1;
            }
            else:
            {
                if resul < 0: { continue; }
                println(vueltas);
            } # acaba while-else
            println(resul);
        } # acaba if
        println(vueltas);
    } # acaba def sumar -let

    read(a); read(b);
    d = 1/b;
    c = c*(c*d)+e;
    println(c);
} # acaba def main -letprintln(c);
```

Salida:

```
1: proc main;
2: int c;
3: int b;
4: int a;
5: real e;
6: real d;
7: proc sumar;
8: val_int y;
9: val_int x;
10: ref_int resul;
11: int vueltas;
12: int aux;
13: aux=y;
14: resul=x;
15: if resul<1000 goto 17;
16: goto 36;
17: vueltas=0;
18: if aux==0 goto 20;
19: goto 29;
20: _t1:=resul+1;
21: resul=_t1;
22: if resul>100000 goto 29;
23: goto 24;
24: _t2:=aux-1;
25: aux=_t2;
26: _t3:=vueltas+1;
27: vueltas=_t3;
28: goto 18;
29: if resul<0 goto 31;
30: goto 32;
31: goto  18;
32: write vueltas;
33: writeln ;
34: write resul;
35: writeln ;
36: write vueltas;
37: writeln ;
38: endproc sumar;
39: read a;
40: read b;
41: _t4:=1/b;
42: d=_t4;
43: _t5:=c*d;
44: _t6:=_t5+e;
45: _t7:=c*_t6;
46: c=_t7;
47: write c;
48: writeln ;
49: halt;
```

Entrada: PruebaBuena3

```
def main ( ):

let aux, y, x : integer in
{
''' esto es un
 comentario multilinea '''

      while y*2 > 10 or 20 > 10 and not 20 > x: {
      aux = 1;
      y = 10;
      } else: {
      x = 1;
      }

} #termina el main
```

Salida:

```
1: proc main ;
2: int aux;
3: int y;
4: int x;
5: _t1 := y * 2;
6: if _t1 > 10 goto 10 ;
7: goto 8 ;
8: if 20 > 10 goto 10 ;
9: goto 15 ;
10: if 20 > x goto 15 ;
11: goto 12 ;
12: aux := 1 ;
13: y := 10 ;
14: goto 5 ;
15: x := 1 ;
16: halt ;
```

Entrada: PruebaMala1

```
def main ():

let a,b,c : integer; d,e : float in
{
```

```
''' esto es un comentario '''
    def sumar (x,y: integer; resul: & integer):
    let aux, vueltas: integer in
    {
        aux = y; resul = x;
        while aux == 0:
        {
            aux = aux - 1;
            vueltas = vueltas + 1;
        }
        else:
        {
            read(def sumar);
            println(vueltas);
        }

        println(aux);
    } # acaba def sumar -let

    read(a); read(b);
    d = 1/b;
    c = c*(c*d)+e;
    println(c);
} # acaba def main -letprintln(c);
```

Salida: line 17: syntax error at 'def'

Entrada: PruebaMala2

```
program ejemplo

{
```

```
   a := a * 0.4756 ;
}
```

Salida: line 17: syntax error at 'de

Entrada: PruebaMala3

```
def main ( ):

let aux, y, x : integer in
{
''' En nuestro lenguaje no permitimos la asociatividad de los operadores
logicos'''

      if not resul < 1000 < 20 or x > 10 and y == 100 : {
      vueltas = 0;
      resul  = 1000;
      }

} # termina el main
```

Salida: line 7: syntax error, unexpected TL, expecting TPLUS or TMINUS or TMULT or TDIV at '<'