

eman ta zabal zazu



Universidad
del País Vasco

Euskal Herriko
Unibertsitatea

COMPILACIÓN

TRIPLE_ESE

Unai Salas Lavesa

USALAS001@IKASLE.EHU.EUS

Mayo 11, 2021

Contents

1	Introducción	2
2	Autoevaluación	2
3	Autómata general	2
4	Análisis léxico	4
4.1	Especificación de los tokens	4
5	Especificación del proceso de traducción	6
5.1	Gramática	6
5.2	Definición de atributos	8
5.3	Descripción de las abstracciones funcionales	9
5.4	ETDS	10
6	Casos de prueba	15
6.1	Pruebas buenas	15
6.1.1	PruebaBuena1.in	15
6.1.2	PruebaBuena2.in	17
6.1.3	PruebaBuena3.in	18
6.1.4	PruebaBuena4.in	19
6.1.5	PruebaBuena5.in	20
6.2	Pruebas malas	21
6.2.1	PruebaMala1	21
6.2.2	PruebaMala2	21
6.2.3	PruebaMala3	22
6.2.4	PruebaMala4	22

1 Introducción

En esta entrega se incluye el front-end de un compilador utilizando la técnica de construcción de traductores ascendente, a partir de un esquema de traducción dirigida por la sintaxis.

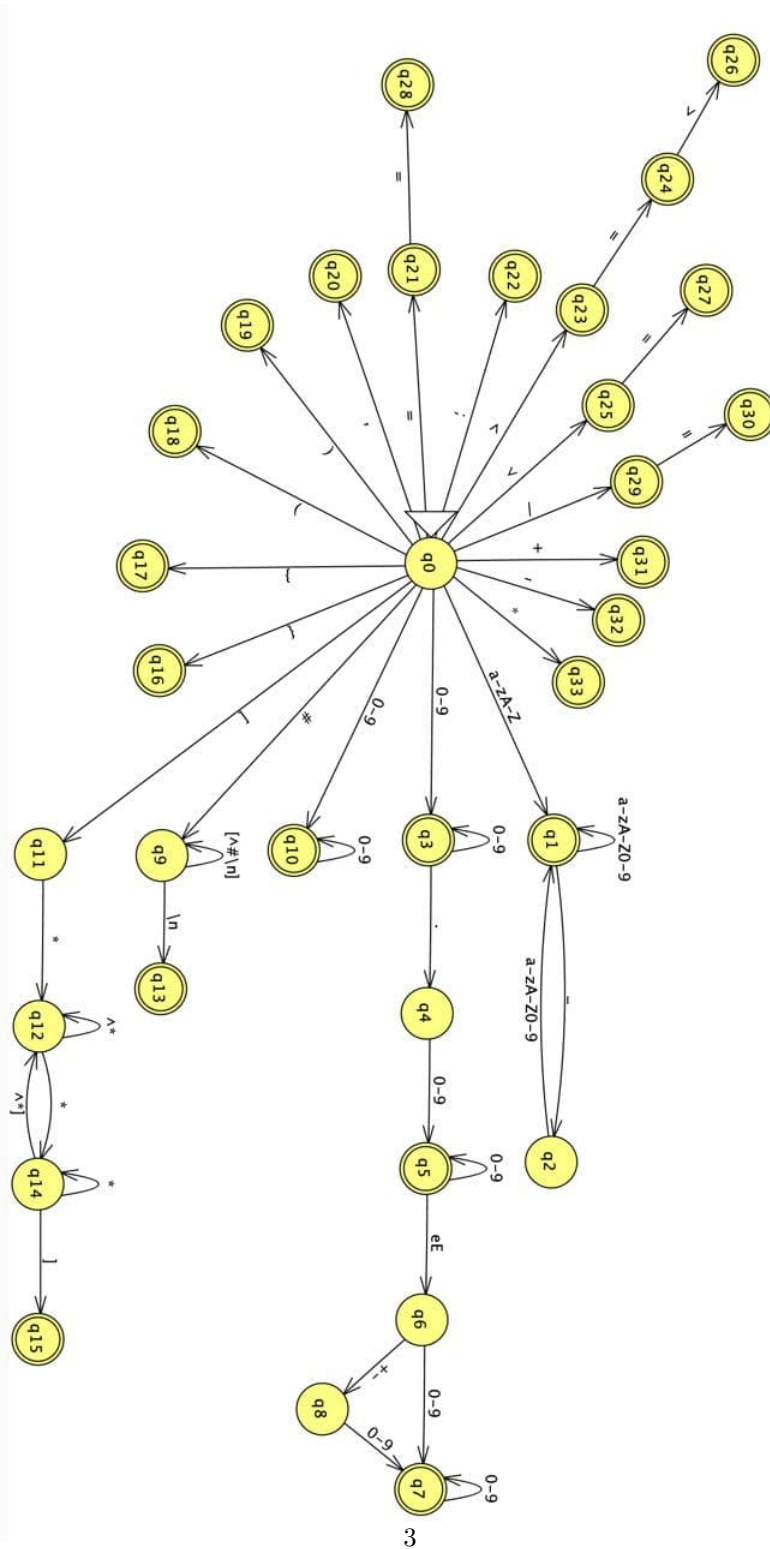
El compilador, cuya gramática ha sido la especificada en el apartado correspondiente, dado como entrada un fichero de código, será capaz de determinar si el código dado es correcto o no, y en caso de serlo, generar la traducción. Además, se han realizado dos objetivos opcionales con el objetivo de conseguir un compilador más completo. Se ha añadido el bucle for como nueva estructura de control y la traducción de las expresiones booleanas and, or y not. Todas ellas pueden ser revisadas más adelante.

2 Autoevaluación

Esta práctica debería, en principio, tener una nota cercana a la nota máxima posible para esta entrega, es decir, cercana a un 7. Se ha implementado todo lo necesario para obtener la nota mínima necesaria para llegar al 5 (diseño, implementación y pruebas del analizador léxico, implementación y pruebas del analizador sintáctico y el desarrollo del ETDS y de las pruebas finales, implementación de un traductor e inclusión de mensajes de errores en determinadas situaciones aplicando comprobaciones estáticas y dinámicas) y además se han implementado dos de las tareas opcionales planteadas. Tras revisar el proyecto en numerosas ocasiones y realizar repetidas pruebas de control sobre los diferentes resultados obtenidos en las diferentes compilaciones realizadas, considero que es un buen trabajo en la cantidad de objetivos que abarca. Habría sido posible ampliar la cantidad de elementos a implementar de haber tenido más tiempo para trabajar en ello, pero desafortunadamente me he detenido en estas dos para poder trabajar otras asignaturas también.

3 Autómata general

El autómata de la próxima página reconoce los tokens incluidos y excluye las palabras que han sido reservadas.



4 Análisis léxico

4.1 Especificación de los tokens

Nombre	Descripción	Expresión regular	Lexemas
TIDENTIFIER	Identificadores	<code>[a-zA-Z](_[a-zA-Z])?[0-9]*</code>	<ul style="list-style-type: none">- variable1- var_as1- var1
TINTEGER	Tipo numérico entero	<code>[0-9]+</code>	<ul style="list-style-type: none">- 9- 1000- 12
TDOUBLE	Tipo numérico real	<code>[0-9](\.[0-9]+)?([eE](\+ -)?[0-9]+)?</code>	<ul style="list-style-type: none">- 0.99- 1.11e+10- 0.50e10
com	Comentario de línea	<code>#\[^\n]*</code>	<ul style="list-style-type: none">- # coment- # com(a)- #C[a,b]
com_multi	Comentario multilínea	<code>(\[^\n]*\ ^\n+)[^\n]*\n\)</code>	<ul style="list-style-type: none">- [* c(a, b) *]- [* ()(-) *]- [****aa**]
RPROGRAM	Palabra reservada program	<code>program</code>	<code>program</code>
RINTEGER	Palabra reservada int	<code>int</code>	<code>int</code>
RFLOAT	Palabra reservada float	<code>float</code>	<code>float</code>
RWHILE	Palabra reservada while	<code>while</code>	<code>while</code>
RUNTIL	Palabra reservada until	<code>until</code>	<code>until</code>
REXIT	Palabra reservada exit	<code>exit</code>	<code>exit</code>
RPROCEDURE	Palabra reservada proc	<code>proc</code>	<code>proc</code>
RIF	Palabra reservada if	<code>if</code>	<code>if</code>
RELSE	Palabra reservada else	<code>else</code>	<code>else</code>
RFOREVER	Palabra reservada forever	<code>forever</code>	<code>forever</code>
RDO	Palabra reservada do	<code>do</code>	<code>do</code>
RSKIP	Palabra reservada skip	<code>skip</code>	<code>skip</code>
RREAD	Palabra reservada read	<code>read</code>	<code>read</code>

RPRINTLN	Palabra reservada println	println	println
RAND	Palabra reservada and	and	and
ROR	Palabra reservada or	or	or
RNOT	Palabra reservada not	not	not
RFOR	Palabra reservada for	for	for
TLBRACE	Token llave izquierda	{	{
TRBRACE	Token llave derecha	}	}
TLPAREN	Token paréntesis izquierda	((
TRPAREN	Token paréntesis derecha))
TCOMMA	Token coma	,	,
TASSIG	Token asignación	=	=
TSEMIC	Token punto y coma	;	;
TCGLE	Token in out	<=>	<=>
TCLT	Token menor	<	<
TCLE	Token menor igual	<=	<=
TCGT	Token mayor	>	>
TCGE	Token mayor igual	>=	>=
TEQUAL	Token igual	==	==
TNEQUAL	Token no igual	/=	/=
TDIV	Token división	/	/
TPLUS	Token suma	+	+
TMINUS	Token resta	-	-
TMUL	Token multiplicación	*	*

5 Especificación del proceso de traducción

5.1 Gramática

```

programa → program id
           declaraciones
           decl_de_subprogs
           { lista_de_sentencias }

declaraciones → tipo lista_de_ident; declaraciones
              | ξ

lista_de_ident → id resto_lista_id

resto_lista_id → , id resto_lista_id
               | ξ

tipo → int
      | float

decl_de_subprogs → decl_de_subprograma decl_de_subprogs
                  | ξ

decl_de_subprograma → proc id
                     argumentos
                     declaraciones
                     decl_de_subprogs
                     { lista_de_sentencias }

argumentos → ( lista_de_param )
            | ξ

lista_de_param → tipo clase_par lista_de_ident resto_lis_de_param

clase_par → =>
           | <=
           | <=>

resto_lis_de_param → ; tipo clase_par lista_de_ident resto_lis_de_param
                   | ξ

lista_de_sentencias → sentencia lista_de_sentencias
                    | ξ

```

```
sentencia → variable = expresion ;  
          | if expresion { lista_sentencias } ;  
          | while forever { lista_sentencias } ;  
          | do { lista_sentencias } until expresion else { lista_sentencias } ;  
          | skip if expresion ;  
          | exit ;  
          | read ( variable ) ;  
          | println ( expresion ) ;  
          | for (tipo variable=expresion; expresion; variable= expresion) {lista_sentencias};
```

```
variable → id
```

```
expresion  expresion == expresion  
          | expresion > expresion  
          | expresion < expresion  
          | expresion >= expresion  
          | expresion <= expresion  
          | expresion /= expresion  
          | expresion + expresion  
          | expresion - expresion  
          | expresion * expresion  
          | expresion / expresion  
          | expresion and expresion  
          | expresion or expresion  
          | not expresion  
          | variable  
          | num_entero  
          | num_real  
          | ( expresion )
```


5.2 Definición de atributos

Atributos: (L: Léxico, S: Sintetizado)

Símbolo	Nombre	Tipo	L/S	Descripción
id	nom	string	L	Contiene la cadena de caracteres del id.
num_entero	nom	string	L	Contiene la cadena de caracteres que representan el valor de un número entero.
num_real	nom	string	L	Contiene la cadena de caracteres que representan el valor de un número real.
tipo	clase	string	S	Contiene el tipo de variable, puede ser ent o real.
clase_par	tipo	string	S	Contiene el tipo, puede ser val o ref.
lista_de_ident	lnom	lista de str	S	Lista de strings con los ids.
resto_lista_id	lnom	lista de str	S	Lista de strings con los ids.
lista_de_sentencias	exits	lista de int	S	Lista numérica con referencias.
lista_de_sentencias	skips	lista de int	S	Lista numérica con referencias.
sentencia	exits	lista de int	S	Lista numérica con referencias.
sentencia	skips	lista de int	S	Lista numérica con referencias.
M	ref	int	S	Contiene la referencia numérica.
variable	nom	string	S	Contiene la cadena de caracteres del nombre de la variable.
expresion	true	lista de int	S	Lista con referencia del goto que saltará en la primera parte del if.
expresion	false	lista de int	S	Lista con referencia del goto que saltará a la siguiente instrucción del if.
expresion	nom	string	S	Contiene la cadena de caracteres de la expresión, o de la variable temporal donde se guardará la expresión.
expresion	tipo	string	S	Guarda la cadena de caracteres del tipo de elemento que es la expresión

5.3 Descripción de las abstracciones funcionales

añadir_inst: código x inst \rightarrow código

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

unir(L1, L2):

Descripción: Dadas dos listas L1 y L2, devuelve la unión de las mismas.

nuevo_id: \rightarrow string

Descripción: no recibe parámetros, genera el siguiente id disponible y lo devuelve.

iniLista: entero \rightarrow lista

Descripción: recibe un cero como parámetro y devuelve una lista vacía de enteros.

iniLista: string \rightarrow lista

Descripción: recibe una cadena vacía como parametro y devuelve una lista vacía de strings.

añadir_declaraciones: lista y tipo \rightarrow código

Descripción: Dada una lista de identificadores y un tipo de clase, crea una línea con la clase, el identificador un ; ,todo ello por cada identificador. *tipo*

añadir: lista y entero \rightarrow lista

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

añadir_params: lista y clase_par \rightarrow código

Descripción: Por cada identificador de la lista crea una línea con la clase_par seguido de un _ , el tipo del identificador y el identificador, y seguido, un ; .

obtenref: código \rightarrow ref.codigo

Descripción: no recibe parámetros y devuelve la dirección (ref) actual.

inilista: arg \rightarrow lista

Descripción: devuelve una lista conteniendo arg, si la entrada es 0, devuelve lista vacía de enteros.

completar: código X lista_ref_código X ref.código \rightarrow código

Descripción: Completa las instrucciones de la lista de referencias (saltos) con la referencia dada.

5.4 ETDS

```

programa → program id {añadir_inst(prog || || id.nom);} (1)
    declaraciones
    decl_de_subprogs
    { lista_de_sentencias }
    {añadir_inst(halt);} (2)

declaraciones → tipo lista_de_ident
    {añadir_declaraciones(lista_de_ident.lnom, tipo.clase)} (3)
    ; declaraciones
    | ξ

lista_de_ident → id resto_lista_id {lista_de_ident.lnom = inilista(id.nom);
    unir(lista_de_ident.lnom, resto_lista_id.lnom);} (4)

resto_lista_id → , id resto_lista_id {resto_lista_id.lnom = inilista(id.nombre);
    resto_lista_id.lnom = unir(resto_lista_id.lnom, resto_lista_id1.lnom)} (5)
    | ξ {resto_lista_id.lnom := inilista(" ");} (6)

tipo → int {tipo.clase = ent;} (7)
    | float {tipo.clase = real;} (8)

decl_de_subprogs → decl_de_subprograma decl_de_subprogs
    | ξ

decl_de_subprograma → proc id {añadir_inst(proc.nom || || id.nom);} (9)
    argumentos
    declaraciones
    decl_de_subprogs
    { lista_de_sentencias }
    {añadir_inst(endproc || );} (10)

argumentos → ( lista_de_param )
    | ξ

lista_de_param → tipo clase_par lista_de_ident {añadir_params(lista_de_ident.lnom,
    clase_par.tipo, tipo.clase)} (11) resto_lis_de_param

```

```

clase_par → => {clase_par.tipo = ref;} (12)
           | <= {clase_par.tipo = val;} (13)
           | <=> {clase_par.tipo = ref;} (14)

resto_lis_de_param → ; tipo clase_par lista_de_ident {añadir_params(lista_de_ident.lnom,
clase_par.tipo, tipo.clase);} (15) resto_lis_de_param
           | ξ

lista_de_sentencias → sentencia lista_de_sentencias
{lista_de_sentencias.exits = unir(sentencia.exits, lista_de_sentencias1.exits);
lista_de_sentencias.skips = unir(sentencia.skips, lista_de_sentencias1.skips);} (16)
           | ξ
           {lista_de_sentencias.exits = inilista(0);
lista_de_sentencias.skips = inilista(0);} (17)

sentencia → variable = expresion ;
           {añadir_inst(variable.nom || = || expresion.nom);
sentencia.exit = inilista(0);
sentencia.skip = inilista(0);} (18)

           | if expresion {M lista_sentencias } M;
           {completar(expresion.true, M1.ref);
completar (expresion.false, M2.ref);
sentencia.exit = lista_sentencia.exit;
sentencia.skip = lista_sentencia.skip;} (19)

           | while forever {M lista_sentencias } M;
           {añadir_inst(goto || M1.ref || );
completar (lista_sentencias.exit, M2.ref+1);
sentencia.exit = inilista(0);
sentencia.skip = inilista(0);} (20)

           | do { M lista_sentencias } until M expresion else {M lista_sentencias } M ;
           {completar(expresion.true, M3.ref);
completar(expresion.false, M1.ref);
completar(lista_sentencias1.skip, M2.ref);
completar(lista_sentencias1.exit, M4.ref);
completar(lista_sentencias2.exit, M4.ref);
sentencia.exit = inilista(0);
sentencia.skip = inilista(0);} (21)

```

```

| skip if expresion M ;
  {completar(expresion.false, M1.ref );
   sentencia.skip := expresion.true;
   sentencia.exit = inilista(0);} (22)

| exit M ;
  {sentencia.skips = inilista(0);
   sentencia.exits = inilista(M1.ref);
   añadir_inst(goto);} (23)

| read ( variable ) ;
  {añadir_inst(read || || variable.nom || );
   sentencia.exits = inilista(0);
   sentencia.skips = inilista(0);} (24)

| println ( expresion ) ;
  {añadir_inst(write || || expresion.nom || );
   añadir_inst(writeln || );
   sentencia.exits = inilista(0);
   sentencia.skips = inilista(0);} (25)

| for (tipo variable=expresion;
  {añadir_declaraciones(inilista(variable.nom), tipo.clase);
   añadir_inst(variable.nom || = || expresion.nom || );} (26)
M expresion M; variable = expresion) {lista_sentencias M};
  añadir_inst(variable.nom || = || expresion.nom || );
  añadir_inst(goto || M1.ref || );
  completar(expresion.true, M2.ref);
  completar(expresion.false, M3.ref +2);
  completar(lista_sentencias.exit, M3.ref +2); (27)

```

variable → **id** {variable.nom = **id**.nom;} (28)

expresion → expresion == expresion

```

  {expresion.nom = iniNom(); expresion.true = inilista(obtenref());
   expresion.false = inilista(obtenref()+1);
   añadir_inst(if || expresion1.nom || == || expresion2.nom || goto);
   añadir_inst(goto);} (29)
| expresion > expresion
  {expresion.nom = iniNom(); expresion.true = inilista(obtenref());
   expresion.false = inilista(obtenref()+1);
   añadir_inst(if || expresion1.nom || > || expresion2.nom || goto);
   añadir_inst(goto);} (30)

```

```

| expresion < expresion
{expresion.nom = iniNom(); expresion.true = inilista(obtenref());
expresion.false = inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || < || expresion2.nom || goto);
añadir_inst(goto);} (31)
| expresion >= expresion
{expresion.nom = iniNom(); expresion.true = inilista(obtenref());
expresion.false = inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || >= || expresion2.nom || goto);
añadir_inst(goto);} (32)
| expresion <= expresion
{expresion.nom = iniNom(); expresion.true = inilista(obtenref());
expresion.false = inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || <= || expresion2.nom || goto);
añadir_inst(goto);} (33)
| expresion /= expresion
{expresion.nom = iniNom(); expresion.true = inilista(obtenref());
expresion.false = inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || /= || expresion2.nom || goto);
añadir_inst(goto);} (34)
| expresion + expresion
{expresion.nom = nuevo_id();
añadir_inst(expresion.nom || = || expresion1.nom || + || expresion2.nom);
expresion.true = inilista(0);
expresion.false = inilista(0);} (35)
| expresion - expresion
{expresion.nom = nuevo_id();
añadir_inst(expresion.nom || = || expresion1.nom || - || expresion2.nom);
expresion.true = inilista(0);
expresion.false = inilista(0);} (36)
| expresion * expresion
{expresion.nom = nuevo_id();
añadir_inst(expresion.nom || = || expresion1.nom || * || expresion2.nom);
expresion.true = inilista(0);
expresion.false = inilista(0);} (37)
| expresion / expresion
{expresion.nom = nuevo_id();
añadir_inst(expresion.nom || = || expresion1.nom || / || expresion2.nom);
expresion.true = inilista(0);
expresion.false = inilista(0);} (38)
| expresion and M expresion
{completar (expresion1.true, M1.ref);
expresion.true = expresion2.true;
expresion.false = unir(expresion1.false, expresion2.false);} (39)

```



```

| expression or M expression
  {completar (expresion1.false, M1.ref);
   expresion.true = unir(expresion1.true, expresion2.true);
   expresion.false = expresion2.false;} (40)
| not expresion
  expresion.true = expresion1.false;
  expresion.false = expresion1.true; (41)
| variable
  {expresion.nom = variable.nom;
   expresion.true = inilista(0);
   expresion.false = inilista(0);} (42)
| num_entero
  {expresion.nom = num_entero.nom;
   expresion.true = inilista(0);
   expresion.false = inilista(0);} (43)
| num_real
  {expresion.nom = num_real.nom;
   expresion.true = inilista(0);
   expresion.false = inilista(0);} (44)
| ( expresion )
  {expresion.nom = expresion1.nom;
   expresion.true = expresion1.true;
   expresion.false = expresion1.false;} (45)

```

M $\rightarrow \xi$ {M.ref:=obtenRef();} (46)

6 Casos de prueba

Se han realizado numerosos casos de prueba para este proyecto, probando así el correcto funcionamiento de las diferentes funcionalidades que ofrece. Se han establecido 5 casos de prueba en los que el compilador no encontrará ningún fallo en los programas y realizará la traducción correctamente. A su vez, también se presentan otras 4 pruebas en las que el compilador encuentra un fallo y no realiza la traducción.

Para todos y cada uno de los programas que deben dar fallos se presenta un comentario que explica el motivo por el que debe dar fallo al ejecutar el compilador. En las próximas páginas se podrá comprobar los resultados tras ejecutar los diferentes casos de prueba y una breve explicación sobre estos. Al ejecutar los casos de prueba se ejecutan unos detrás de otros y en caso de ser un código correcto, lo traducirá. En caso de ser erróneo no lo traducirá, sino que indicará el error y pasará a la siguiente prueba.

Todos los casos de prueba mencionados a continuación pueden encontrarse en el directorio `PRACTICA_LS/Pruebas`.

6.1 Pruebas buenas

6.1.1 PruebaBuena1.in

Este caso de prueba fue facilitado por la profesora de la asignatura, Begoña Losada, para poder realizar pruebas iniciales sobre el proyecto tras realizar las implementaciones en el proyecto de lo trabajado en los laboratorios de la asignatura.

En este caso de prueba se comprueba que funcionan bien las declaraciones, los comentarios de una línea y multilínea, las estructuras de control `if`, `do until else`, `skip if`, `read` y `println`.

En la siguiente página podemos ver el resultado que obtenemos al ejecutar el compilador y traducir dicho caso de prueba.


```
(base) → PRACTICA_LS git:(master) X make
./parser <Pruebas/PruebaBuena1.in
ha comenzado...

1: program ejemplo_con_nombre_muy_largo;
2: ent a;
3: ent b;
4: ent c;
5: real d;
6: real e;
7: proc suma;
8: val_ent x;
9: val_ent y;
10: ref_ent resul;
11: ent aux;
12: ent iteraciones;
13: aux := x;
14: resul := y;
15: if resul < 1000 goto 17;
16: goto 33;
17: iteraciones := 0;
18: _t1 := resul + 1;
19: resul := _t1;
20: if resul > 1000000 goto 26;
21: goto 22;
22: _t2 := aux - 1;
23: aux := _t2;
24: _t3 := iteraciones + 1;
25: iteraciones := _t3;
26: if aux = 0 goto 28;
27: goto 18;
28: if resul < 0 goto 30;
29: goto 31;
30: goto 33;
31: write iteraciones;
32: writeln;
33: endproc;
34: read a;
35: read b;
36: if b=0 goto ERRORDIV0
37: _t4 := 1 / b;
38: d := _t4;
39: if a=0 goto ERRORDIV0
40: _t5 := 0.1e-1 / a;
41: e := _t5;
42: _t6 := c * d;
43: _t7 := c * _t6;
44: _t8 := _t7 + e;
45: c := _t8;
46: _t9 := c * c;
47: write _t9;
48: writeln;
49: halt;
ha finalizado BIEN...
```

Figure 2: Resultado del caso de prueba para el fichero PruebaBuena1

6.1.2 PruebaBuena2.in

En este caso de prueba se comprueban principalmente las características básicas necesarias que debe presentar el compilador y que no han sido revisadas anteriormente, es decir, se comprueba el correcto funcionamiento del `exit` y el `while forever`.

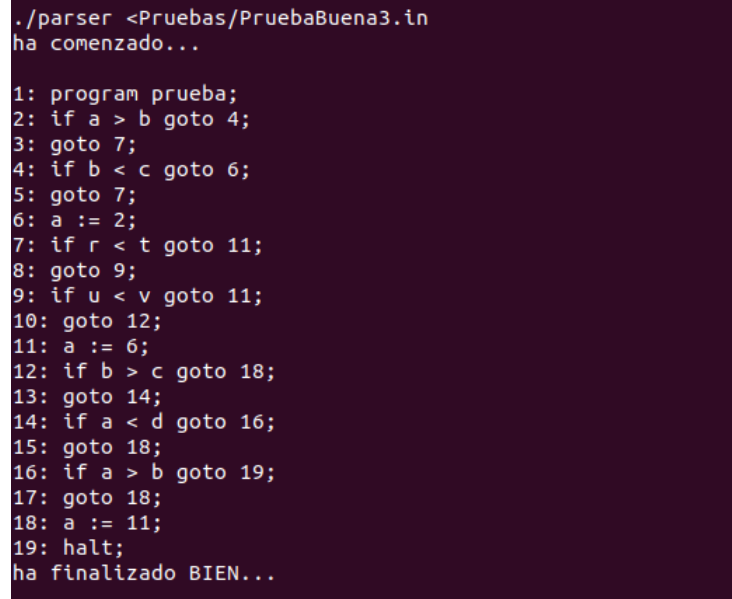
```
./parser <Pruebas/PruebaBuena2.in
ha comenzado...

1: program programa_ejemplo2;
2: ent n1;
3: ent n2;
4: ent n3;
5: ent n_total;
6: proc mult;
7: val_ent n1;
8: val_ent n2;
9: ref_ent n3;
10: if n1 > n2 goto 12;
11: goto 14;
12: _t1 := n1 * n2;
13: n3 := _t1;
14: if n2 <= n1 goto 16;
15: goto 19;
16: _t2 := n1 * n1;
17: _t3 := _t2 + n2;
18: n3 := _t3;
19: read var1;
20: write var1;
21: writeln;
22: if n2 <= n1 goto 29;
23: goto 24;
24: n1 := n2;
25: if n2 >= n1 goto 27;
26: goto 28;
27: goto 29;
28: goto 19;
29: endproc;
30: read n1;
31: read n2;
32: if a=0 goto ERRORDIV0
33: _t4 := 0.1e-1 / a;
34: e := _t4;
35: _t5 := n3 * e;
36: n_total := _t5;
37: write n_total;
38: writeln;
39: halt;
ha finalizado BIEN...
```

Figure 3: Resultado del caso de prueba para el fichero PruebaBuena2

6.1.3 PruebaBuena3.in

En este caso de prueba se ha comprobado el correcto funcionamiento de las expresiones booleanas and, or y not a través de tres sencillos ifs que facilitan su implementación. En la figura nº4 puede verse el resultado tras ejecutar el compilador y obtener la traducción.



```
./parser <Pruebas/PruebaBuena3.in
ha comenzado...

1: program prueba;
2: if a > b goto 4;
3: goto 7;
4: if b < c goto 6;
5: goto 7;
6: a := 2;
7: if r < t goto 11;
8: goto 9;
9: if u < v goto 11;
10: goto 12;
11: a := 6;
12: if b > c goto 18;
13: goto 14;
14: if a < d goto 16;
15: goto 18;
16: if a > b goto 19;
17: goto 18;
18: a := 11;
19: halt;
ha finalizado BIEN...
```

Figure 4: Resultado del caso de prueba para el fichero PruebaBuena3

6.1.4 PruebaBuena4.in

Para este caso de prueba se han probado dos fors anidados, con el objetivo de que se pudiera validar así que la implementación del for ha sido la correcta. Además, se ha comprobado que el exit funciona adecuadamente al ejecutarse en un for y asegurarnos de que cumple correctamente con su función principal en este tipo de bucles también.

```
./parser <Pruebas/PruebaBuena4.in
ha comenzado...

1: program prueba;
2: ent p;
3: ent a;
4: p := 2;
5: a := 0;
6: ent i;
7: i := 0;
8: if i < 5 goto 10;
9: goto 32;
10: _t1 := i + 3;
11: _t2 := a + 1;
12: a := _t2;
13: ent j;
14: j := 0;
15: if j < i goto 17;
16: goto 22;
17: _t3 := j + 1;
18: _t4 := p + 5;
19: p := _t4;
20: j := _t3;
21: goto 15;
22: if p < i goto 24;
23: goto 30;
24: if a > p goto 26;
25: goto 30;
26: if a=0 goto ERRORDIV0
27: _t5 := p / a;
28: p := _t5;
29: goto 32;
30: i := _t1;
31: goto 8;
32: write p;
33: writeln;
34: write a;
35: writeln;
36: halt;
ha finalizado BIEN...
```

Figure 5: Resultado del caso de prueba para el fichero PruebaBuena3

6.1.5 PruebaBuena5.in

En este caso de prueba se hace una prueba general de que todo lo implementado en los objetivos opcionales funciona adecuadamente en un mismo programa. Se tratan tanto las expresiones booleanas como el for, así como el exit y la comprobaciones con los divisores.

```
./parser <Pruebas/PruebaBuena5.in
ha comenzado...

1: program completo;
2: ent a;
3: ent b;
4: ent c;
5: real d;
6: real e;
7: ent aux;
8: ent iteraciones;
9: aux := x;
10: resul := y;
11: if resul < 1000 goto 13;
12: goto 51;
13: iteraciones := 0;
14: if aux > resul goto 16;
15: goto 20;
16: if b < c goto 18;
17: goto 20;
18: _t1 := resul + 1;
19: resul := _t1;
20: if resul > b goto 35;
21: goto 22;
22: _t2 := aux - 1;
23: aux := _t2;
24: if b > c goto 30;
25: goto 26;
26: if a < aux goto 28;
27: goto 30;
28: if a > b goto 33;
29: goto 30;
30: if a=0 goto ERRORDIV0
31: _t3 := b / a;
32: b := _t3;
33: _t4 := iteraciones + 1;
34: iteraciones := _t4;
35: if aux = 0 goto 37;
36: goto 14;
37: if resul < 0 goto 39;
38: goto 40;
39: goto 51;
40: ent i;
41: i := 0;
42: if i < 5 goto 44;
43: goto 49;
```

Figure 6: Primera parte del resultado del caso de prueba para el fichero PruebaBuena5

```
44: _t5 := i + 3;
45: _t6 := a + b;
46: b := _t6;
47: i := _t5;
48: goto 42;
49: write iteraciones;
50: writeln;
51: read a;
52: read b;
53: if b=0 goto ERRORDIV0
54: _t7 := 1 / b;
55: d := _t7;
56: if a=0 goto ERRORDIV0
57: _t8 := 0.1e-1 / a;
58: e := _t8;
59: _t9 := c * d;
60: _t10 := c * _t9;
61: _t11 := _t10 + e;
62: c := _t11;
63: _t12 := c * c;
64: write _t12;
65: writeln;
66: halt;
ha finalizado BIEN...
```

Figure 7: Segunda parte del resultado del caso de prueba para el fichero PruebaBuena5

6.2 Pruebas malas

6.2.1 PruebaMala1

En esta prueba se ha cambiado el tipo de las inicializaciones de "int" a "integer", por lo que al no reconocerlo, devolverá error, indicando primero la línea en la que lo ha encontrado y después el error que ha encontrado. El programa nos comunicará que ha finalizado con un fallo y detendrá la ejecución de ese caso de prueba para continuar con ese.

```
./parser <Pruebas/PruebaMala1.in
ha comenzado...

line 3: syntax error, unexpected TIDENTIFIER, expecting TLBRACE
ha finalizado MAL...
```

Figure 8: Resultado del caso de prueba para el fichero PruebaMala1

6.2.2 PruebaMala2

En esta prueba se ha modificado la palabra "proc" del procedure mult por "procedure", por lo que al no reconocerlo, devolverá error, indicando primero la línea en la que lo ha encontrado y después el error que ha encontrado. El programa nos comunicará que ha finalizado con un fallo y detendrá la ejecución de este caso de prueba para continuar con el siguiente.

```
./parser <Pruebas/PruebaMala2.in  
ha comenzado...  
  
line 9: syntax error, unexpected TIDENTIFIER, expecting TLBRACE  
ha finalizado MAL...
```

Figure 9: Resultado del caso de prueba para el fichero PruebaMala2

6.2.3 PruebaMala3

En este caso de prueba se comprueba que al añadir un exit fuera del bucle, el compilador nos indicará que por medio de un mensaje de error que tenemos un exit fuera del bucle, sin escribirnos el código intermedio. Este es el único caso en el que a pesar de estar mal el código indicará que ha acabado bien, ya que realiza la impresión del error que le hemos indicado que haga.

```
./parser <Pruebas/PruebaMala3.in  
ha comenzado...  
  
Error en la semantica  
line 23: Exit fuera del bucle  
ha finalizado BIEN...
```

Figure 10: Resultado del caso de prueba para el fichero PruebaMala3

6.2.4 PruebaMala4

En este programa se comprueba que al sacar la actualización de la variable que recorre el bucle (la i del primer for), debe devolver error y no debe, por tanto, traducir el caso de prueba.

```
./parser <Pruebas/PruebaMala4.in  
ha comenzado...  
  
line 9: syntax error, unexpected TRPAREN, expecting TSEMIC  
ha finalizado MAL...
```

Figure 11: Resultado del caso de prueba para el fichero PruebaMala4