eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Bleeding Fox

Práctica de Compilación

Edu Vallejo (evallejo009@ikasle.ehu.eus)

Noel Arteche (narteche002@ikasle.ehu.eus)

18 de mayo del 2019

Índice

1. Introducción

El presente informe sirve a modo de documentación de la práctica llevada a cabo por Edu Vallejo y Noel Arteche para la asignatura de Compilación, en el grado de Ingeniería Informática, durante el curso 2018/19 en la Universidad del País Vasco.

La práctica consiste en el diseño e implementación del *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 lenguaje de entrada (fuente) al compilador es un lenguaje de alto nivel, y el de salida un código de tres direcciones. La implementación se ha llevado a cabo en C++ haciendo uso de herramientas y lenguajes adicionales como Flex y Bison.

En la sección 2 de este documento se especifican los objetivos de la práctica, el trabajo realizado y las dedicaciones de cada miembro. En la sección 3 se adjunta la tabla que se corresponde con el análisis léxico del lenguaje de entrada, en la sección 4 se adjunta la gramática empleada de partida, la sección 5 describe el tipo y formato de los atributos y la sección 6 define las abstracciones empleadas en el ETDS. Así las cosas, en el apartado 7 incluimos finalmente el ETDS de los objetivos básicos de la práctica y un caso de pruebas básico. En la sección 8 y sus subsecciones se desarrolla el ETDS modificado, adjuntando en cada paso las modificaciones necesarias para la implementación de cada caso de uso. Asimismo, cada caso viene acompañado de sus pruebas de ejecución.

Edu Vallejo

Noel Arteche

En Donostia, a 18 de mayo de 2019

2. Objetivos y dedicaciones

El objetivo básico de la práctica era implementar el analizador léxico y sintáctico y presentar una versión funcional del traductor básico.

Además de eso, hemos realizado los objetivos adicionales que se detallan a continuación:

1. Se ha especificado una estructura de control adicional, un *switch*.
2. Se han incluido expresiones booleanas.
3. Se han añadido restricciones y comprobaciones semánticas.
4. Se han incluido llamadas a procedimientos.

En base a estos objetivos, nuestra práctica podría optar a una calificación máxima de 11 puntos.

El proyecto ha sido llevado a cabo conjuntamente por ambos miembros del equipo en un período de 20 horas por persona, en las que hemos trabajado de forma presencial los dos a la vez excepto en la etapa final, donde Edu se ocupó personalmente de ultimar los detalles de la implementación, mientras que Noel se dedicó a redactar la documentación.

3. Análisis léxico

La siguiente tabla incluye la especificación léxica del lenguaje de entrada (están incluidos también los tokens que se han añadido posteriormente para los objetivos opcionales). El autómata puede encontrarse en la carpeta adjunta en la que se ha entregado este fichero.

Nombre del token	**Descripción informal**	**Expresión regular**	**Lexemas**
RPROGRAM	<Trivial>	program	<igual que la expresión regular>
RVAR	“	var	“
RPROCEDURE	“	procedure	“
RINT	“	integer	“
RFLOAT	“	float	“
RIN	“	in	“
ROUT	“	out	“
RIF	“	if	“
RTHEN	“	then	“
RELSE	“	else	“
RSKIP	“	skip	“
RDO	“	do	“
RUNTIL	“	until	“
RWHILE	“	while	“
RREAD	“	read	“
RPRINT	“	println	“
RSWITCH	“	switch	“
RCASE	“	case	“
TLBRACE	“	{	“
TRBRACE	“	}	“
TCOMMA	“	,	“
TLPAREN	“	(	“
TRPAREN	“	)	“

TCOLON	“	:	“
TSEMIC	“	;	“
TMUL	“	*	“
TMINUS	“	-	“
TPLUS	“	+	“
TDIV	“	/	“
TASSIG	“	=	“
TEQ	“	==	“
TCNE	“	/=	“
TLT	“	<	“
TLEQ	“	<=	“
TGT	“	>	“
TGEQ	“	>=	“
TAND	“	&&	“
TOR	“	||	“
TNOT	“	!	“
TIDENTIFIER	Un carácter alfabético seguido de caracteres alfanuméricos y barra baja, no puede haber dos barras bajas contiguas ni el identificador puede acabar con barra baja.	[a-zA-Z](_?[a-zA-Z0-9])*	temp, tempPtr, tempPtr2, temp_2
TDOUBLE	Un numero natural (uno o más caracteres numéricos) seguido de un punto y una parte fraccionaria (uno o más caracteres numéricos), y opcionalmente un exponente (“e” o “E” seguido de un numero entero (uno o más caracteres numéricos precedidos	[0-9]+\.[0-9]+([Ee][-+]?[0-9]+)?	7.55, 597.68, 51.5E4, 51.5e+4, 51.5e-4

	opcionalmente de un signo negativo)).		
TINTEGER	Un numero natural (uno o más caracteres numéricos).	[0-9]+	5, 0, 23, 784
<comentario de linea>	Dos barras seguidas de cualquier secuencia de caracteres (de al menos longitud 1, salvo salto de línea), seguidos de un salto de línea	\/\/.+\n	// This is a comment, // @TODO // =), ^_^ , -_-, UwU
<comentario multi-linea>	Una barra seguida de una estrella seguida de cualquier secuencia de caracteres que no contenga una subsecuencia “*/” seguida de la secuencia “*/”.	\/*((*+[^*/])|[^*])*(*)+\/	/* Multi line **comment** nº 5 */

4. Gramática básica

La gramática empleada para los objetivos básicos de la práctica básica es la siguiente:

programa → **program id** bloque

bloque → **{** declaraciones decl_de_subprogs lista_de_sentencias **}**

declaraciones → **var** lista_de_ident **:** tipo **;** declaraciones
| ε

lista_de_ident → **id** resto_lista_id

resto_lista_id → **, id** resto_lista_id
| ε

tipo → **integer**
| **float**

decl_de_subprogs → decl_de_subprograma decl_de_subprogs
| ε

decl_de_subprograma → **procedure id** argumentos bloque **;**

argumentos → **(** lista_de_param **)**
| ε

lista_de_param → lista_de_ident **:** clase_par tipo resto_lis_de_param

clase_par → **in** | **out** | **in out**

resto_lis_de_param → **;** lista_de_ident **:** clase_par tipo
{añadir_declaraciones(lista_de_ident.lnom, clase_par.nombre ||
tipo.nombre);} resto_lis_de_param
| ε

lista_de_sentencias → sentencia lista_de_sentencias
| ε

sentencia → variable = expresion **;**
| **if** expresión **then {** M lista_de_sentencias **};**

| **while** M expresión **{** M lista_de_sentencias **};**

| **do M {**lista_de_sentencias**} until** M expresión **else {** M lista_de_sentencias**};**

| **skip if** expresion **;**

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

| **println (**expresión**);**

M → | ε

variable → **id**

expresion → expresión == expresión

| expresion > expresión

| expresion < expresión

| expresion >= expresión

| expresion <= expresión

| expresion /= expresión

| expresion + expresion

| expresion – expresión

| expresion * expresión

| expresion / expresión

| **id**

| **num_entero**

| **num_real**

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

5. Atributos

Símbolo	Atributos
id	`nombre`: string que almacena el nombre de una variable
lista_de_ident	`lnom`: lista de strings con nombres de identificadores
tipo	`nombre`: string con el nombre del tipo de dato
resto_lista_id	`lnom`: lista de strings con nombres de identificadores
integer	`nombre`: string con el valor **"`integer`"**
float	`nombre`: string con el nombre **"`float`"**
clase_par	`nombre`: string para determinar el tipo de parámetro (**in**, **out**…)
variable	`nombre`: string que almacena el nombre de una variable
M	`ref`: línea de código en la que se encuentra
sentencia	`exit`: lista de direcciones de código intermedio en las que hay instrucciones de salto sin completar.
lista_de_sentencias	`exit`: lista de direcciones de código intermedio en las que hay instrucciones de salto sin completar.
expresion	`nombre`: string que contiene la expresión en cuestión
	`true`: lista de direcciones de código intermedio en las que hay goto sin completar
	`false`: lista de direcciones de código intermedio en las que hay goto sin completar
num_real	`nombre`: string que contiene el número en cuestión
num_entero	`nombre`: string que contiene el número en cuestión

6. Interfaz de abstracciones

Cabecera	Explicación
añadir_inst(String **str**)	Añade una nueva línea con el contenido de **str** en el código compilado. El **str** debe de ser una instrucción válida en el lenguaje intermedio al que se compila. Esta función por sí sola es suficiente para generar el código, sin embargo, se ha optado por extender la interfaz con objeto de abstraer la traducción de ciertas sentencias. Toda función que modifica el código hace uso de esta para ello.
añadir_declaraciones(Lista<String> **lista**, String **tipo**)	Añade instrucciones de declaraciones de las variables con identificadores en **lista** de tipo **tipo.**
añadir(Lista<String> **lista**, String **str**)	Añade la string **str** a la lista de strings **lista** por detrás, no devuelve nada.
añadir_lista(Lista **lista1**, Lista **lista2**)	Añade los elementos de la lista **lista2** a la lista de strings **lista1** por detrás, no devuelve nada.
completar(Lista<int> **E**, int **ref**)	Completa las instrucciones goto en las posiciones indicadas en **E** con el valor **ref**.
nuevo_id()	Genera y devuelve un identificador temporal que se usa para operaciones intermedias.

7. ETDS básico (corregido)

A continuación se adjunta el ETDS básico sobre el que se han implementado los objetivos básicos:

programa → **program id** {añadir_inst("prog" || id.nombre || “;”);} bloque ;{añadir_inst(“halt;”);}

bloque → { declaraciones decl_de_subprogs lista_de_sentencias }

declaraciones → **var** lista_de_ident **:** tipo **;**
{añadir_declaraciones(lista_ident.lnom, tipo.nombre);} declaraciones
| ε

lista_de_ident → **id** resto_lista_id
{añadir(lista_de_ident.lnom, id.nombre);
añadir(lista_de_ident.lnom, resto_lista_id.lnom);}

resto_lista_id → **, id** resto_lista_id
{añadir(resto_lista_id.lnom, id.nombre);
añadir(resto_lista_id.lnom, resto_lista_id1.lnom);}
| ε {resto_lista_id.lnom = new lista();}

tipo → **integer** {tipo.nombre := integer.nombre;}
| **float** {tipo.nombre := float.nombre;}

decl_de_subprogs → decl_de_subprograma decl_de_subprogs
| ε

decl_de_subprograma → **procedure id** { añadir_inst("proc" || id.nombre || “;”);} argumentos bloque {añadir_inst(“endproc;”);} ;

argumentos → **(** lista_de_param **)**
| ε

lista_de_param → lista_de_ident **:** clase_par tipo
{añadir_declaraciones(lista_de_ident.lnom, clase_par.nombre || tipo.nombre);} resto_lis_de_param

clase_par → **in** {clase_par.nombre := “val_”;}
| **out** {clase_par.nombre := “ref_”;}
| **in out** {clase_par.nombre := “ref_”;}

resto_lis_de_param → **;** lista_de_ident **:** clase_par tipo
{añadir_declaraciones(lista_de_ident.lnom, clase_par.nombre || tipo.nombre);} resto_lis_de_param
| ε

lista_de_sentencias → sentencia {añadir_lista(lista_de_sentencias.exit, sentencia.exit);}lista_de_sentencias
{añadir_lista(lista_de_sencias.exit, lista_de_sentencias1.exit);}
| ε {lista_de_sentencias.exit := new list();}

sentencia → variable = expresion **;** {añadir_inst(variable.nombre || := || expresion.nombre || ;);

| **if** expresión **then** **{** M lista_de_sentencias **};**
{completar(expresion.true, M.ref);
completar(expresion.false, obten_ref());
añadir_lista(sentencia.exit,
lista_de_sentencias.exit);}

| **while** M expresión **{** M lista_de_sentencias **};**
{completar(expresion.true, M2.ref);
completar(expresion.false, obten_ref() + 1);
añadir_inst(“goto” || M1.ref);
terminar_skip(lista_de_sentencias.exit, M1.ref);}

| **do M** **{**lista_de_sentencias**}** **until** M expresión **else** **{** M lista_de_sentencias**}**[1]**;**
{completar(expresion.true, M3.ref);
completar(expresion.false, M1.ref);
terminar_skip(lista_de_sentencias.exit, M2.ref);}

| **skip if** expresion **;**
{completar(expresion.true, obten_ref());
completar(expresion.false, obten_ref() + 1);
añadir_instrucción(“goto ”);
sentencia.exit := new list(obten_ref()); }

| **read (**variable**);**
{añadir_inst(“read ” || variable.nombre);}

[1] Semántica: una sentencia skip if dentro del bloque else se considera un skip if para un bucle en el que está anidado el do until.

| **println (**expresión**); {**
añadir_inst(“write ” || variable.nombre || ;);
añadir_inst(“writeln;”);}

M → | ε {M.ref := obten_ref(); }

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

expresion → expresión == expresión
{expresión.nombre = nuevo_id();
expresion.true := new list(obten_ref());
expresion.false := new list(obten_ref() + 1);
añad_inst(if || expreison1.nombre|| == ||
expresion2.nombre || goto);
añadir_inst(goto);}

| expresion > expresión
{expresión.nombre = nuevo_id();
expresion.true := new list(obten_ref());
expresion.false := new list(obten_ref() + 1);
añad_inst(if || expreison1.nombre|| > ||
expresion2.nombre || goto);
añadir_inst(goto);}

| expresion < expresión
{expresión.nombre = nuevo_id();
expresion.true := new list(obten_ref());
expresion.false := new list(obten_ref() + 1);
añad_inst(if || expreison1.nombre|| < ||
expresion2.nombre || goto);
añadir_inst(goto);}

| expresion >= expresión
{expresión.nombre = nuevo_id();
expresion.true := new list(obten_ref());
expresion.false := new list(obten_ref() + 1);
añad_inst(if || expreison1.nombre|| >= ||
expresion2.nombre || goto);
añadir_inst(goto);}

| expresion <= expresión
{expresión.nombre = nuevo_id();
expresion.true := new list(obten_ref());
expresion.false := new list(obten_ref() + 1);
añad_inst(if || expreison1.nombre|| <= ||
expresion2.nombre || goto);
añadir_inst(goto);}

| expresion /= expresión
{expresión.nombre = nuevo_id();
expresion.true := new list(obten_ref());

```
expresion.false := new list(obten_ref() + 1);
añad_inst(if || expreison1.nombre|| /= ||
expresion2.nombre || goto );
añadir_inst(goto);}
```

| expresion + expresion

```
{ expresion.nombre:=nuevo_id();
 añadir_inst(expresion.nombre||:=||
expresion1.nombre||+|| expresion2.nombre);
expresion.true := new list();
expresion.false := new list();
}
```

| expresion – expresión

```
{ expresion.nombre:=nuevo_id();
añadir_inst(expresion.nombre||:=||
expresion1.nombre||-|| expresion2.nombre);
expresion.true := new list();
expresion.false := new list();}
```

| expresion * expresión

```
{ expresion.nombre:=nuevo_id();
añadir_inst(expresion.nombre||:=||
expresion1.nombre||*|| expresion2.nombre);
expresion.true := new list();
expresion.false := new list();}
```

| expresion / expresión

```
{ expresion.nombre:=nuevo_id();
añadir_inst(expresion.nombre||:=||
expresion1.nombre||/|| expresion2.nombre);
expresion.true := new list();
expresion.false := new list();}
```

| **id**

```
{expresion.nombre:= id.nombre;
añadir_inst(expresion.nombre||:=|| id.nombre);
expresion.true := new list();
expresion.false := new list();}
```

| **num_entero**

```
{expresion.nombre:= num_entero.nombre;
añadir_inst(expresion.nombre||:=||
num_entero.nombre);
expresion.true := new list();
expresion.false := new list();}
```

| **num_real**

```
{expresion.nombre:= num_real.nombre;
añadir_inst(expresion.nombre||:=||
num_real.nombre);
expresion.true := new list();
```

```
                    expresion.false := new list();}

                    | ( expresion )
                    {expresion.nombre:= expresion1.nombre;
                    expresion.true := expresion.true;
                    expresion.false := expresion.false}
```

Con el ETDS básico es posible obtener la siguiente traducción de un programa fuente:

```
program ejemplo_basico {

     /* Calcula fibonacci iterativo */
    procedure fibonacci (x: in
    integer ; y: out integer) {

         var aux1 ,aux2, aux3, i :
         integer;

         if (x < 2) then {
             y = 1;
         };
         i = 1;
         aux1 = 1;
         aux2 = 1;
         while (i < x){
             aux3 = aux1 + aux2;
             aux1 = aux2;
             aux2 = aux3;
             i = i + 1;
         };

         y = aux2;

    };
};
```

```
 1  prog ejemplo_basico;
 2  proc fibonacci;
 3  val_integer x;
 4  ref_integer y;
 5  integer i;
 6  integer aux3;
 7  integer aux2;
 8  integer aux1;
 9  if x < 2 goto 11;
10  goto 12;
11  y := 1;
12  i := 1;
13  aux1 := 1;
14  aux2 := 1;
15  if i < x goto 17;
16  goto 24;
17  _t1 := aux1 + aux2;
18  aux3 := _t1;
19  aux1 = aux2;
20  aux2 = aux3;
21  _t2 := i + 1;
22  i := _t2;
23  goto 15;
24  y := aux2;
25  endproc;
26  halt;
```

En el código entregado hay pruebas incorrectas que pueden ejecutarse para ver el funcionamiento del traductor con código incorrecto.

8. Objetivos adicionales

Como se ha comentado en la introducción, se han añadido cuatro objetivos opcionales: nuevas estructura de control (*switch*), expresiones booleanas, validaciones semánticas y llamadas a subprogramas.

8.1 Nueva estructura de control: *switch*

El *switch* es una estructura de control de flujo cuya semántica se sencilla: comprueba si la expresión que se ha pasado como entrada coincide con alguna de las expresiones de los casos disponibles y, en tal caso, ejecuta ese fragmento de código. A diferencia de otros lenguajes de programación, la semántica de nuestro *switch* es ligeramente más restrictiva, en tanto que una vez se encuentra una coincidencia y se ejecuta ese fragmento de código, el *switch* termina.

Para implementarlo se han incluido dos nuevas palabras reservadas, RSWITCH y RCASE, que se corresponden con las expresiones regulares *switch* y *case*, respectivamente.

En la gramática hay símbolos nuevos: *caselist* contiene dos campos, *skips* (una lista de direcciones en las que pueden completarse instrucciones *skip*) y *endSwitch*, donde se guardan las direcciones de instrucciones *goto* a terminar con la dirección de fin del *switch*. El no terminal *caset* tiene los mismos atributos que *caselist*.

Se ha incluido una pila, *switchExpression*, donde se guarda la condición de cabecera del *switch*. Así, si hay varios *switch* anidados, las cabeceras se van superponiendo una sobre otra. Para hacer uso de la pila, contamos con las abstracciones funcionales `empilarSwitchExpression()`, `desmpilarSwitchExpression()` y `cimaSwitchExpression()`:

Cabecera	Explicación
`empilarSwitchExpression(String str)`	Empila el nombre de la variable (temporal o no temporal) que funciona como expresión de cabecera de un bloque *switch*.
`cimaSwitchExpression()`	Devuelve el nombre de la variable (temporal o no temporal) en la cima de la pila de cabeceras de *switchs*.
`desempilarSwitchExpression()`	Desempila la última variable usada como cabecera de un *switch*.

Más adelante se detallan las restricciones semánticas impuestas al *switch*. Este es el ETDS con la gramática ampliada:

sentencia → …
| RSWITCH expresion `{empilarSwitchExpression(expresion);}`
TLBRACE caselist TRBRACE TSEMIC
`{completar(caselist.endSwitch, obtenRef());`
`desempilarSwitchExpression();`
`sentencia.exit = caselist.skips;}`

caselist → caset caselist
`{caselist = caselist1;`
`añadir_lista(caselist.endSwitch, caset.endSwitch);`
`añadir_lista(caselist.skips, caset.skips);}`
| `{caselist.endSwitch = new list();`
`caselist.skips = new list();}`

caset → RCASE expresion RTHEN TLBRACE M
`{añadir_instruccion("if " + cimaSwitchExpresion() + "`
`!= " + expresion.nombre + "goto" );}`
lista_de_sentencias TRBRACE TSEMIC
`{caset.skips = lista_de_sentenicas.exit;`
`caset.endSwitch = new list(obtenRef());`
`añadirInstruccion("goto");`
`completar(list(M.ref), obtenRef());}`

Para mostrar el correcto funcionamiento de la traducción se presenta la salida del siguiente programa:

```
program ejemplo_switch {

   var a, b, c : integer;

   a = 5;
   b = 25;
   c = c - 5;

   switch (a * a) {
             case (a) then {
           println(a);
             };
             case (b) then {
           println(b);
             };
```

```
 1  prog ejemplo_switch;
 2  integer c;
 3  integer b;
 4  integer a;
 5  a := 5;
 6  b := 25;
 7  _t1 := c - 5;
 8  c := _t1;
 9  _t2 := a * a;
10  if _t2 != a goto 14;
11  write a;
12  writeln;
13  goto 22;
14  if _t2 != b goto 18;
15  write b;
```

```
            case (c) then {
          println(c);
            };
    };
};
```

```
16  writeln;
17  goto 22;
18  if _t2 != c goto 22;
19  write b;
20  writeln;
21  goto 22;
22  halt;
```

8.2 Expresiones booleanas

Las expresiones booleanas permiten hacer más complejas las condiciones de los bloques *if* y los *while* mediante la inclusión de los operadores lógicos *not*, *and* y *or*. Siguiendo la sintaxis de C y todos los lenguajes inspirados en él, estos operadores se representan con ! (*not*), && (*and*) y || (*or*), para lo que se han añadido los tokens TNOT, TAND y TOR.

Para este caso se han hecho cambios en la gramática. Se ha eliminado el símbolo *expresion* y se ha sustituido por tres variantes: *arithmetic_expresion*, *logical_expresion* y *relational_expresion*. Con estos nuevos símbolos, en las reglas de producción de sentencia el *if*, *while*, *do while* y *skip if* solo admiten condiciones del tipo *logical_expresion*, mientras que *switch* y *case* reciben *arithmetic_expresions*. En cuanto a atributos, se han añadido las siguientes campos:

Símbolo	Atributos
arithmetic_expresion	`nombre`: string que contiene la expresión en cuestión
relational_expresion	`true`: dirección de código intermedio en la que hay goto sin completar
	`false`: dirección de código intermedio en la que hay goto sin completar
logical_expresion	`trues`: lista de direcciones de código intermedio en las que hay goto sin completar
	`falses`: lista de direcciones de código intermedio en las que hay goto sin completar

El ETDS con la nueva gramática es el siguiente:

arithmetic_expresion → arithmetic_expresion + arithmetic_expresion
```
{arithmetic_expresion.nombre:=nuevo_id();
 añadir_inst(arithmetic_expresion.nombre||:=||
arithmetic_expresion1.nombre||+||
arithmetic_expresion2.nombre);}
```

| arithmetic_expresion - arithmetic_expresion
```
{arithmetic_expresion.nombre:=nuevo_id();
 añadir_inst(arithmetic_expresion.nombre||:=||
arithmetic_expresion1.nombre||-||
arithmetic_expresion2.nombre);}
```

| arithmetic_expresion * arithmetic_expresion
```
{arithmetic_expresion.nombre:=nuevo_id();
 añadir_inst(arithmetic_expresion.nombre||:=||
arithmetic_expresion1.nombre||*||
arithmetic_expresion2.nombre);}
```

| arithmetic_expresion / arithmetic_expresion
{arithmetic_expresion.nombre:=nuevo_id();
añadir_inst(arithmetic_expresion.nombre||:=||
arithmetic_expresion1.nombre||/||
arithmetic_expresion2.nombre);}

| **id**
{arithmetic_expresion.nombre:= id.nombre;}

| **num_entero**
{arithmetic_expresion.nombre:= num_entero.nombre;}

| **num_real**
{arithmetic_expresion.nombre:= num_real.nombre;}

| **(** arithmetic_expresion **)**
{arithmetic_expresion.nombre:=
arithmetic_expresion1.nombre;}

relational_expresion → arithmetic_expresion == arithmetic_expresion
{relational_expresion.true := obten_ref();
relational_expresion.false := obten_ref() + 1;
añad_inst(if || arithmetic_expresion1.nombre|| == ||
arithmetic_expresion2.nombre || goto);
añadir_inst(goto);}

| arithmetic_expresion > arithmetic_expresion
{relational_expresion.true := obten_ref();
relational_expresion.false := obten_ref() + 1;
añad_inst(if || arithmetic_expresion1.nombre|| > ||
arithmetic_expresion2.nombre || goto);
añadir_inst(goto);}

| arithmetic_expresion < arithmetic_expresion
{relational_expresion.true := obten_ref();
relational_expresion.false := obten_ref() + 1;
añad_inst(if || arithmetic_expresion1.nombre|| < ||
arithmetic_expresion2.nombre || goto);
añadir_inst(goto);}

| arithmetic_expresion >= arithmetic_expresion
{relational_expresion.true := obten_ref();
relational_expresion.false := obten_ref() + 1;
añad_inst(if || arithmetic_expresion1.nombre|| >= ||
arithmetic_expresion2.nombre || goto);
añadir_inst(goto);}

| arithmetic_expresion <= arithmetic_expresion
{relational_expresion.true := obten_ref();
relational_expresion.false := obten_ref() + 1;
añad_inst(if || arithmetic_expresion1.nombre|| <= ||
arithmetic_expresion2.nombre || goto);
añadir_inst(goto);}

| arithmetic_expresion /= arithmetic_expresion
{relational_expresion.true := obten_ref();
relational_expresion.false := obten_ref() + 1;
añad_inst(if || arithmetic_expresion1.nombre|| /= ||
arithmetic_expresion2.nombre || goto);
añadir_inst(goto);}

| **(** relational_expresion **)**
{
relational_expresion.true := relational_expresion1.true;
relational_expresion.false := relational_expresion1.false}

logical_expresion → logical_expresion TOR M logical_expreison
{logical_expresion.falses =
logical_expresion2.falses;
añadir_lista(logical_expresion1.trues,
logical_expresion2.trues);
añadir_lista(logical_expresion.trues,
logical_expresion1.trues);
completar(logical_expresion1.falses, M.ref);}

| logical_expresion TAND M logical_expreison
{logical_expresion.trues =
logical_expresion2.trues;
añadir_lista(logical_expresion1.falses,
logical_expresion2.falses);
añadir_lista(logical_expresion.falses,
logical_expresion1.falses);
completar(logical_expresion1.trues, M.ref);}

| TNOT logical_expresion
{logical_expresion.trues =
logical_expresion1.falses;
logical_expresion.falses =
logical_expresion1.trues;}

| (logical_expresion)
{logical_expresion.trues =
logical_expresion1.trues;
logical_expresion.falses =
logical_expresion1.falses;}

```
| relational_expresion
{logical_expresion.trues = new
list(relational_expresion.true);
logical_expresion.falses = new
list(relational_expresion.false);}
```

A continuación mostramos un ejemplo de programa con expresiones lógicas y la correspondiente traducción. Para ver ejemplos de traducciones incorrectas con programas no válidos puede consultarse el entregable:

```
program prueba_booleanas {

       //variables
       var a,b,c,d,e,f : integer;

       /* leer */
       read(a);
       read(b);
       read(c);
       read(d);
       read(e);
       read(f);

       /* ifs*/
       if (a < b) then {
          println(a);
       };

       if (!(a < b)) then {
          println(b);
       };

       if ((a > c) && (b == d)) then {
          println(c);
       };

       if (!(!a > c) && (b == d)) then {
          println(d);
       };

        if (!(((a == c) && (b == d)) ||
        ((c==e)  && (f == d)))) then {
          println(e);
           println(f);
       };
};
```

```
 1  prog prueba_booleanas;
 2  integer f;
 3  integer e;
 4  integer d;
 5  integer c;
 6  integer b;
 7  integer a;
 8  read a;
 9  read b;
10  read c;
11  read d;
12  read e;
13  read f;
14  if a<b goto 16;
15  goto 18;
16  write a;
17  writeln;
18  if a<b goto 22;
19  goto 20;
20  write b;
21  writeln;
22  if a>c goto 24;
23  goto 28;
24  if b==d goto 26;
25  goto 28;
26  write c;
27  writeln;
28  if a>c goto 32;
29  goto 30;
30  if b==d goto 34;
31  goto 32;
32  write d;
33  writeln;
34  if a==c goto 36;
35  goto 38;
36  if b==d goto 46;
```

```
37  goto 38;
38  if c==e goto 40;
39  goto 42;
40  if f==d goto 46;
41  goto 42;
42  write e;
43  writeln;
44  write f;
45  writeln;
46  halt;
```

8.3 Validaciones semánticas

Las validaciones semánticas se comportan como una segunda capa de lectura de la entrada. En este caso, se han incluido las siguientes comprobaciones y restricciones:

1. No es posible asignar un valor entero a una variable real y viceversa.
2. No es posible asignar valores a variables de tipo *in*.
3. No es posible operar valores enteros y reales entre sí.
4. Solo es posible utilizar variables que no hayan sido previamente declaradas y/o no estén en el alcance (*scope*) de ese fragmento del código.
5. Solo es posible llamar a procedimientos que hayan sido previamente declarados (aunque pueden ser llamados a sí mismos dentro del cuerpo de su código; esto es, hay recursividad).
6. Una variable y un procedimiento no pueden tener el mismo identificador.
7. En las llamadas a procedimientos ha de respetarse el número de parámetros, los tipos indicados y que las variables sean de tipo *in* o *out*. Adicionalmente, no puede pasarse una expresión aritmética a evaluar en el hueco de un parámetro de tipo *out*.
8. En un *switch* debe coincidir el tipo de la condición con los casos a evaluar.
9. La semántica del *skip* funciona de forma que solo puede existir dentro de un bloque *while* o *do while* (es decir, dentro de una estructura de control iterativa), y si se cumple la condición del *if* se salta una iteración.
10. A la instrucción *read* no se le puede pasar una variable de tipo *in*.

Cabe destacar que, cada vez que se encuentra una incongruencia (ya sea léxica, sintáctica o semántica) el traductor se detiene para indicar el fallo que se ha producido y la línea en la que ha tenido lugar. Es decir, la compilación funciona como un *debugger* interactivo en el que van corrigiéndose gradualmente los errores que contiene el código fuente, a modo de *Interactive Debugging*.

Para la correcta implementación de todos estos casos se ha utilizado la tabla de símbolos, implementada como una pila de mapas *hash* que guardan los identificadores de cada bloque.

Además, se han realizado algunos cambios en los atributos para hacer posible guardar los tipos de ciertas variables y expresiones:

Símbolo	Atributos
arithmetic_expresion, variable	`nombre`: string que contiene la expresion o el nombre de la variable temporal
	`tipo:` string que guarda el tipo de la variable temporal (val, ref)

	`proteccion:` string que guarda el nivel de accesibilidad de la variable temporal
argumentos, lista_de_param, resto_lis_de_param	`nombres:` vector que guarda los nombres de los parámetros en la declaración de una función
	`tipos:` vector que guarda los tipos de los parámetros en la declaración de una función (val o ref)
	`protecciones:` vector que guarda el nivel de accesibilidad de los parámetros en la declaración de una función

Finalmente, las validaciones semánticas se llevan a cabo a través de las siguientes abstracciones funcionales, entre las que también se encuentran las funciones mediante las que se manipula la pila que implementa la tabla de símbolos:

Cabecera	Explicación
`comienzoBloque()`	Se llama al principio de un bloque para definir el comienzo de un nuevo segmento de alcance (*scope*). Empila en la pila de mapas un mapa vacío.
`finalBloque()`	Se llama al final de un bloque para definir el final de un segmento de alcance (*scope*). Desempila el mapa del bloque que termina..
`obtenerIdentificadorVariable(String id)`	Devuelve una estructura que contiene los datos del identificador id si existe; null en caso de que no exista.
`obtenerIdentificadorFuncion(String f)`	Devuelve una estructura que contiene los datos del identificador de la función f si existe; null en caso de que no exista.
`declararIdentificador(String nombre, String tipo, String proteccion)`	Declara un nuevo identificador en la tabla de símbolos con nombre nombre, de tipo tipo y cuyo nivel de accesibilidad es protección.
`añadirSubprograma(String nombre, List<String> nombres, List<String> tipos, List<String> protecciones)`	Registra el nombre de un nuevo subprograma y la información de sus parámetros, al tiempo que genera código intermedio haciendo visible la declaración. Si ese identificador ya está en uso, se produce un error.
`añadirDeclaraciones(List<String> nombres, String tipo)`	Registra un conjunto de variables del mismo tipo. Genera código intermedio y si algún

	identificador ya está en uso produce un error.
`error()`	Imprimer un error relacionado con el problema encontrado y detiene la traducción.

La única nueva regla de producción es la siguiente:

variable → TIDENTIFIER
```
{if(obtenerIdentificadorVariable(TIDENTIFIER) ==
NULL) error();
variable =
obtenerIdentificadorVaraible(TIDENTIFIER);}
```

En el resto de la gramática y del ETDS se han realizado pequeñas modificaciones haciendo uso de las abstracciones funcionales definidas arriba. A continuación resumimos brevemente algunas, aunque no es el ETDS completo:

bloque → TLBRACE `{comienzoBloque();}` declaraciones
decl_de_subprogs lista_de_sentencias `{finalBloque();}`
TRBRACE

decl_de_subprog → RPROCEDURE TIDENTIFIER `{...}` argumentos
```
{añadirSubprograma(TIDENTIFIER,
argumentos.nombres, argumentos.tipos,
argumentos.protecciones); ... }
```
bloque TSEMIC
```
{...}
```

sentencia → variable TASSIG arithmetic_expresion TSEMIC
```
{if (variable.proteccion == “val”) error();
...}
```

arithmetic_expresion → ...
| variable
```
{arithmetic_expresion = variable;}
```
| ...

Para comprobar el funcionamiento de las validaciones semánticas se ha diseñado una prueba que se adjunta en la siguiente sección, una vez implementadas las llamadas a subprogramas.

8.4 Llamadas a procedimientos

El programa admite llamadas a procedimientos siguiendo las pautas indicadas en el enunciado. Se imprime una línea por cada parámetro recibido (indicando el tipo, *val* o *ref*) y luego imprime una instrucción indicando el nombre al que se llama.

Las restricciones semánticas sobre las llamadas a procedimientos se han discutido en el apartado anterior pero se implementan aquí: se comprueba que el nombre de la función existe, los parámetros son correctos (cantidad, tipo, in/out...) etc.

No ha sido necesario añadir nuevos *tokens* al parser, pero se han añadido nuevos símbolos a la gramática: *funcion*, *argumentos_llamada* y *resto_lista_args*. En lo referente a sus atributos, tenemos:

Símbolo	Atributos
argumentos_llamada, resto_lista_args	`nombres:` vector que guarda los nombres de los parámetros en la declaración de una función
	`tipos:` vector que guarda los tipos de los parámetros en la declaración de una función (val o ref)
	`protecciones:` vector que guarda el nivel de accesibilidad de los parámetros en la declaración de una función
funcion	`nombre:` nombre de la función
	`tipos:` `vector` que guarda los tipos de los parámetros de la función (val o ref)
	`protecciones: vector que guarda el nivel de accesibilidad de los parámetros de la función`

Se han añadido la siguiente abstracción funcional:

Cabecera	Explicación
`hacerLlamada(String b, List<String> nombres, List<String> tipos, List<String> protecciones)`	Comprueba que es posible hacer la llamada a esa función con esos parámetros. Escribe un error en caso de que no sea posible y, si no, genera el código intermedio pertinente.

En la gramática se han añadido las siguientes reglas de producción, que se muestran en el siguiente ETDS:

funcion → TIDENTIFIER {if (obtenerIdentificadorFuncion(TIDENTIFIER) == NULL) then error(); funcion = obtenerIdentificadorFuncion(TIDENTIFIER);}

sentencia → ...
| funcion TLPAREN argumentos_llamada TRPAREN TSEMIC {hacerLlamada(funcion.nombre, argumentos_llamada.nombres, argumentos_llamada.tipos, argumentos_llamada.protecciones);}

El siguiente fragmento de las pruebas muestra el funcionamiento del traductor sobre un programa fuente que contiene llamadas a subprogramas y que sirve también para comprobar las validaciones semánticas. Basta comentar alguna de las líneas o hacer pequeñas modificaciones para ver cómo el traductor reacciona a incongruencias varias:

```
program prueba_semantica {

   var a,b : integer;
   var c,d : float;

   //Descomentar para error de declaracion    repetida
   //var a,c : integer;

   procedure proc1 (x : in integer ; y : out integer){
       var e,f: float;

       //Descomentar para error de proteccion
       //x = e;
       //read(x);
       //proc1(x,x); //Variable in en field out
       //proc1(y,x);

       y = x;
       y = x + y;

       e = f;

       proc1(x,y);
       proc1(y,y);

   };

   //Descomentar para error de Redeclaracion de subprograma
   /*
   procedure proc1(x : in integer ; y : out integer){
```

```
1: prog prueba_semantica;
2: integer b;
3: integer a;
4: float d;
5: float c;
6: proc proc1;
7: integer x;
8: integer y;
9: proc proc1;
10: float f;
11: float e;
12: y:=x;
13: _t1:=x+y;
14: y:=_t1;
15: e:=f;
16: param_val x;
17: param_ref y;
18: call proc1;
19: param_val y;
20: param_ref y;
21: call proc1;
22: endproc;
23: a:=b;
24: _t2:=a+b;
25: b:=_t2;
26: a:=67;
27: c:=d;
28: _t3:=c+d;
```

```
};
*/

//Descomentar para producir error de tipos
//a = b + c;
// d = a + c;
// a = 7.2;
// d = 2;

a = b;
b = a + b;
a = 67;
c = d;
d = c + d;
c = 4.2;

//Descomentar para error de no declaracion de variable
//f = 2.2; //Scope
// g = 15;
//a = g + b;
//read(f);
//proc1 = b;

a = b;
b = a + b;

//Descomentar para error de no declaracion de funcion
//proc2(a,b);
//b(a,c); //Identificador de variable

proc1(a,b);
proc1(a,a);
proc1(b,b);
proc1(b,a);

//Descomentar para error de llamada a funcion
//proc1(a,b,c); //Numero de parametros erroneo
//proc1(a);
//proc1(c,b);      //Tipo de parametros erroneo
//proc1(c,d);
//proc1(a,a+b); //expresion en campo out

proc1(a+b*b, a);

//Descomentar para error de skipif fuera de bucle
//skipif (a > b);

while (a > b && c < d){

    a = a+b/2;
    c = c + 1.0;
```

```
29: d:=_t3;
30: c:=4.2;
31: a:=b;
32: _t4:=a+b;
33: b:=_t4;
34: param_val a;
35: param_ref b;
36: call proc1;
37: param_val a;
38: param_ref a;
39: call proc1;
40: param_val b;
41: param_ref b;
42: call proc1;
43: param_val b;
44: param_ref a;
45: call proc1;
46: _t5:=b*b;
47: _t6:=a+_t5;
48: param_val _t6;
49: param_ref a;
50: call proc1;
51: if a>b goto 53;
52: goto 64;
53: if c<d goto 55;
54: goto 64;
55: _t7:=b/2;
56: _t8:=a+_t7;
57: a:=_t8;
58: _t9:=c+1.0;
59: c:=_t9;
60: _t10:=b/2;
61: if a==_t10 goto 51;
62: goto 63;
63: goto 51;
64: _t11:=a+b;
65: if _t11 != 20 goto
69;
66: write a;
67: writeln;
68: goto  73;
69: if _t11 != b goto 73;
70: write b;
71: writeln;
72: goto  73;
73: halt;
```

```
        skipif (a == (b/2));
    };

    switch (a + b){

        case (20) then{
               println(a);
        };

        case (b) then{
               println(b);
        };

        //Descomentar para error de tipos no compatibles en
switch case

        /*
        case (20.0) then{
               println(b);
        };
        */

        /*
        case (c + d) then{
               println(b);
        };
        */


    };
};
```