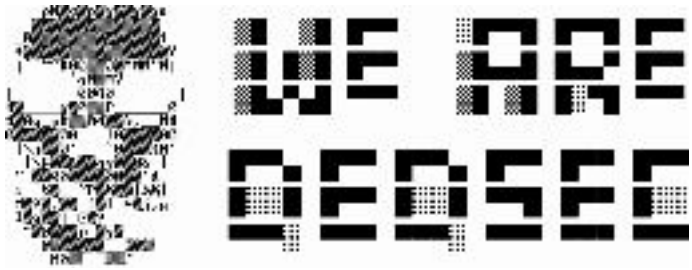
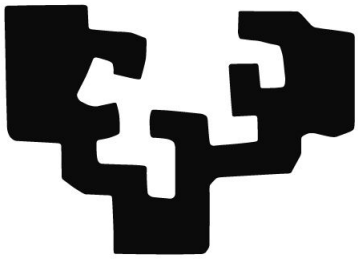


eman ta zabal zazu



Universidad del País Vasco

Euskal Herriko Unibertsitatea

Práctica Compilación

Grupo: DedSec

Andoni Rivera (arivera015@ikasle.ehu.eus)
Iker García (igarcia945@ikasle.ehu.eus)
Eritz Yerga (eyerga001@ikasle.ehu.eus)

Índice:

Introducción

En este trabajo, se ha realizado un traductor de un lenguaje de programación a lenguaje intermedio. Este traductor es capaz de detectar diferentes tokens como son, números enteros, reales, identificadores, operadores aritméticos y booleanos, listas, palabras reservadas, etc. y siguiendo la gramática del lenguaje es capaz de crear un código intermedio que posteriormente serviría para realizar la traducción a código objeto.

Para esta práctica, hemos conseguido realizar los siguientes puntos del alcance de la práctica:

- Implementación del analizador léxico y sintáctico. Desarrollo del ETDS.
- Implementación del traductor.
- Diseño e implementación de la traducción de expresiones booleanas.
- Diseño e implementación de restricciones semánticas.
- Llamadas a procedimientos (uso correcto de parámetros reales).
- Arrays multidimensionales.

Para cada punto indicado, se han realizado sus respectivos ETDS y pruebas que se documentará posteriormente.

Autoevaluación

Consideramos que hemos realizado un trabajo completo, donde la documentación plasma adecuadamente las mejoras progresivas que se han ido añadiendo al traductor.
Además, hemos añadido una variedad de ejemplos relacionados con dichas mejoras, donde se puede observar qué ocurre cuando se escribe correctamente el código y cuando no se hace.
Respecto a la práctica en sí, hemos cumplido tanto los objetivos mínimos como los opcionales.

Esfuerzo invertido

No hemos medido con exactitud el tiempo total invertido pero aproximadamente se han trabajado entre 30 y 40 horas en grupo, estando todos los integrantes del grupo presentes.

Análisis léxico

En este apartado se muestran las expresiones de los distintos tokens que forman parte del traductor:

Token	E. R.	Descripción	Ejemplos
programa	`programa`	Palabra reservada que da inicio a un programa.	**programa** main
procedimiento	`procedimiento`	Palabra reservada que da inicio a un procedimiento.	**procedimiento** ejemplo
variables	`variables`	Palabra reservada que indica que el siguiente o las siguientes identificadores son variables.	**variable** a,b,c : real; **variable** z : real; **variable** x, y : entero;
entero	`entero`	Palabra reservada para indicar que una variable es de tipo entero.	variable y, z : **entero**; procedimiento suma (a,b: in **entero**)
real	`real`	Palabra reservada para indicar que una variable es de tipo real.	variable a,b: **real**;
in	`in`	Palabra reservada que indica variables de entrada en un procedimiento.	procedimiento uno (a: **in** real) procedimiento saludo (b: out real; a, z: **in** entero)
out	`out`	Palabra reservada que indica variables de salida en un procedimiento.	procedimiento hola(a: in entero; b: **out** real)
inout	`“in out”`	Palabra reservada que indica variables de entrada y salida en un procedimiento.	procedimiento tres(a: **inout** real)

si	`si`	Palabra reservada que hace referencia a un “if” en cualquier otro lenguaje de programación.	**si** (a < b) entonces
			{
			…
			}

entonces	`entonces`	Palabra reservada que hace referencia a un “then” en cualquier otro lenguaje de programación.	si (z == x) **entonces**
			{
			…
			}

repetir siempre	`“repetir siempre”`	Palabra reservada que hace repetir siempre un conjunto de instrucciones	**repetir siempre**
			{
			leer(z);
			}

repetir	`repetir`	Palabra reservada que hace repetir un conjunto de instrucciones	**repetir**{ leer(a) } hasta (x<c)

hasta	`hasta`	Palabra reservada que indica el límite de número de repeticiones del token “repetir”	repetir{
			leer(a)
			x = x+1;
			} **hasta** (x<c)

salir si	`“salir si”`	Palabra reservada condicional que indica cuándo salir de un “repetir siempre”	repetir siempre
			{
			a = a+1;
			salir si (a == 50);
			}

leer	`leer`	Palabra reservada que realiza una lectura de la entrada estándar y lo asigna a una variable	**leer**(y);
escribir_linea	`escribir_linea`	Palabra reservada que imprime por salida estándar.	**escribir_linea**(a*2);
comentario1	`\/\/.+\n`	A partir de que se escribe // hasta el salto de línea, el compilador no	**// Esto es un comentario**

		interpreta nada.	
comentario2	`\(\*+([^*]|\*[^)])*\*+\)`	Desde que se escribe (* hasta que se vuelve a escribir *) el compilador no interpreta nada.	**(** comentario **)** **(** comentario * de * varias lineas **)**
salto de línea	`[ \t\n]`	Indica que hay un salto de línea	a = a+b; z = 3;
Identificador	`[a-zA-Z](\_?[a-zA-Z0-9])*`	Token que representa los identificadores para variables, programas y procedimientos.	**abcvar1** **variable_esta** **entero_1**
Doble	`[0-9]+\.[0-9]+([eE][\+\-]?[0-9]+)?`	Token que representa un número real (con o sin parte exponencial)	z = **0.1E10**; h = **0.24522** *2; i = **3.1416e-1**;
Entero	`[0-9]+`	Token que representa un número entero.	o = **45663**; z = **75** * **1767**; q = **126** + **241** / **256**;
Token Desconocido	`.`	Token no reconocidos por el programa (que no cumplen ninguna de las descripciones anteriores)	**agesf,.,.aesfsahgd**
Suma	**“+”**	Token que realiza una suma de sus dos expresiones contiguas.	a = 5 **+** z
Resta	**“-”**	Token que realiza la diferencia de sus dos expresiones contiguas.	b = 5 **-** 2
Multiplicación	**“*”**	Token que realiza el producto de sus dos expresiones contiguas.	c = z * 1
División	**“/”**	Token que realiza el	q = a **/** b

		cociente de la expresión izquierda entre la derecha	
Asignación	“=”	Token que asigna la expresión de la derecha al identificador de la izquierda.	A **=** b + 5 / 7
Mayor	“>”	Token que compara si la expresión de la izquierda es mayor que la de la derecha.	5 -7 **>** a - 70
Menor	“<”	Token que compara si la expresión de la izquierda es menor que la de la derecha.	a **<** z + 3
Igual	“==”	Token que compara si ambas expresiones son iguales	a - 5 **==** s + 3
Mayor o igual	“>=”	Token que compara si la expresión de la izquierda es mayor o igual que la de la derecha.	5 -7 **>=** a - 70
Menor o igual	“<=”	Token que compara si la expresión de la izquierda es menor o igual que la de la derecha.	a **<=** z + 3
Distinto	“/=”	Token que compara si ambas expresiones son distintas.	a - 65 **/=** 7
Negación	`not`	Token que niega el resultado de una expresión booleana	**not** c/=5
O (booleanos)	`or`	Token que une dos expresiones booleanas creando una única donde es	a<b **or** c>d

		true en caso de que alguna de ellas lo sea.	
Y (boolenos)	`and`	Token que une dos expresiones boolenas creando una única donde es true en el único caso de que ambas lo sean.	a<b **and** c>d
Doble punto	`":"`	Token que se utiliza para separar variables de sus tipos.	variables d,e **:** real;
Punto y coma	`";"`	Token que sirve para indicar el fin de instrucción.	variables d,e : real**;**
Coma	`","`	Token que sirve para separar distintas variables a la hora de declararlas.	variables d**,**e : real;

Llaves	`"{" y "}"`	Tokens que limitan funciones, o instrucciones que tienen que realizar iteradores	si b<c entonces
			{
			b=b+1;
			a=a-1;
			}

Paréntesis	`"(" y ")"`	Tokens que rodean parámetros de funciones.	procedimiento sumar **(**x,y: in entero; b: out entero; resul: in out entero**)**
			variables a,aux:entero;
			{
			a=x+y;
			}

Array	`array`	Token que indica el tipo de variable array	variable array2: **array**[1][2] of real;
Indicador de array	`of`	Token que separa las dimensiones de un array y el tipo de	variable array2: array[1][2] **of** real;

		elementos que contiene el array.	
Corchetes	“[” y “]”	Tokens que limitan el valor de una dimensión de un array.	variable array2: array**[**1**][**2**]** of real;

Objetivos obligatorios:

1. Gramática:

prog -> **programa IDENTIFICADOR**
declaraciones decl_de_subprogs
{ lista_de_sentencias **}**

declaraciones -> ε
| **variables** lista_de_ident **:** tipo **;** declaraciones

lista_de_ident -> **IDENTIFICADOR** resto_lista_id

resto_lista_id -> ε
| **,** **IDENTIFICADOR** resto_lista_id

tipo -> **entero**
| **real**

decl_de_subprogs -> ε
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
argumentos declaraciones
{ lista_de_sentencias **}**

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident : clase_par tipo resto_lis_de_param

clase_par -> **in**
| **out**
| **in out**

resto_lis_de_param -> ε
| **;** lista_de_ident **:** clase_par tipo resto_lis_de_param

lista_de_sentencias -> ε
| sentencia lista_de_sentencias

M -> ε

sentencia -> variable **=** expresion **;**
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M

| **repetir** M **{** lista_de_sentencias **}** **hasta** expresion **;** M
| M **repetir siempre {** lista_de_sentencias **}** M
| **salir si** expresion **;** M
| **leer (** variable **) ;**
| **escribir (** expresion **) ;**

variable -> **IDENTIFICADOR**
| **IDENTIFICADOR** array_acceso

expresion -> expresion **==** expresion
| expresion **<** expresion
| expresion **>** expresion
| expresion **<=** expresion
| expresion **>=** expresion
| expresion **/=** expresion
| expresion **+** expresion
| expresion **-** expresion
| expresion ***** expresion
| expresion **/** expresion
| **IDENTIFICADOR**
| **ENTERO**
| **REAL**
| **(** expresion **)**

list_expr -> expresion resto_lista_expr

resto_lista_expr -> ε
| **,** expresion resto_lista_expr

2. Definición de atributos:

A continuación se van a definir los distintos terminales y no terminales, indicando que tipo de información contiene.

Terminal/No Terminal	Atributo que contiene
TIDENTIFIER, TINTEGER, TDOUBLE, TVAR, TSUM, TRES, TMUL, TDIV, TENTERO, TREAL, TDOSP, TSEMIC, TASSIG, TMENOR, TMAYOR, TCOMA, TEQ, TGTH, TLTH, TNEQ, RPROGRAM, TPROC, TKOPEN, TKCLOSE, TPOPEN, TPCLOSE, TIN TOUT, TINOUT, TSI, TENTONCES, TREPSIEMP, TREPETIR, THASTA, TSALSI, TLEER, TESCRIBIR.	nombre (léxico)

sentencia	exit (léxico)
lista_de_ident, resto_lista_id, lista_de_param	lnom (sintetizado)
clase_par	tipo (léxico)
tipo	clase (léxico)
variable	nombre (léxico)
expresion	nombre, true, false (sintetizado)
M	ref (léxico)

- **Nombre:** Almacena el nombre de un identificador (String) o el contenido del token.
- **Clase:** Almacena un string que define el tipo de una variable (“int”, “real”)
- **Tipo:** Almacena un string que indica si una variable es de entrada/salida
- **Exit:** Entero que almacena la referencia a la línea de código intermedio donde termina la estructura actual.
- **Lnom:** Lista de nombre de identificadores (string).
- **Ref:** Entero que almacena la referencia a una línea de código intermedio
- **True:** Lista de referencias a la línea de código intermedio de la comparación verdadera. Por defecto vacías si no se asignan.
- **False:** Lista de referencias a la línea de código intermedio de la comparación falsa. Por defecto vacías si no se asignan.

3. Descripción de las abstracciones funcionales:

- **añadir_inst:** Añade una instrucción a la traducción del programa, además, para que el ETDS quede más limpio se encarga de añadir el “;” al final de la instrucción. Recibe como argumento el String a escribir.
- **añadir_declaraciones:** Recibe como parámetros una lista y un String. Por cada elemento de la lista que recibe, escribe una línea con el tipo de la variable seguido del nombre de la variable (elemento de la lista).
- **añadir:** Añade un identificador a una lista. Recibe como parámetro la lista a la que hay que añadir el identificador y un String.
- **añadir_parametros:** Recibe como parámetros una lista y dos String. Por cada elemento de la lista que recibe, añade un parámetro a la traducción de una declaración de subprograma con su clase y tipos especificados y con los identificadores descritos en la lista que recibe.
- **lista_vacia:** Inicializa una lista.
- **unir:** Une dos listas en una. Recibe como parámetro las dos listas a unir.
- **inilista:** Inicializa una lista y añade el elemento que recibe como parámetro (string).
- **nuevo_id:** Crea una nueva variable intermedia.
- **obten_ref:** Obtiene el número de la siguiente línea a escribir.

- **completar_inst:** Completa los gotos de la línea dada para que vayan a la otra línea especificada, recibe como parámetro dos enteros, uno de ellos indica una línea de código intermedio donde hay un goto y el otro la línea que escribir en el goto.

4. ETDS:

programa -> programa **IDENTIFICADOR** {añadir_inst("prog"+IDENTIFICADOR.nombre);}
declaraciones decl_de_subprogs { lista_de_sentencias }
{añadir_inst("halt");}

declaraciones -> ε
| **var** lista_de_ident **:** tipo **;**
{añadir_declaraciones(tipo.clase,lista_de_ident.lnom);} declaraciones

lista_de_ident -> **IDENTIFICADOR** resto_lista_id {lista_de_ident.lnom:=lista_vacia();
lista_de_ident.lnom:=añadir(resto_lista_id.lnom,IDENTIFICADOR.nombre);}

resto_lista_id -> ε {resto_lista_id.lnom:=lista_vacia();}
| **,** **IDENTIFICADOR** resto_lista_id
{resto_lista_id.lnom:=añadir(resto_lista_id2.lnom,IDENTIFICADOR.nombre);}

tipo -> **entero** {tipo.clase="int";}
| **real** {tipo.clase="real";}

decl_de_subprogs -> ε
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
{añadir_inst("proc"+IDENTIFICADOR.nombre);}
argumentos
declaraciones
{ lista_de_sentencias **}** {añadir_inst("endproc");}

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident **:** clase_par tipo {añadir_parametros(clase_par.tipo,
tipo.clase, lista_de_ident.lnom);}

resto_lis_de_param

clase_par -> **in** {clase_par.tipo="val_";}
| **out** {clase_par.tipo="ref_";}
| **in out** {clase_par.tipo="ref_";}

resto_lis_de_param: **ε**
| **;** lista_de_ident : clase_par tipo { añadir_parametros(clase_par.tipo, tipo.clase, lista_de_ident.lnom); } resto_lis_de_param

lista_de_sentencias -> **ε**
{ lista_de_sentencias.exit = lista_vacia(); }
| sentencia lista_de_sentencias { lista_de_sentencias.exit = unir(sentencia.exit, lista_de_sentencias.exit); }

sentencia -> variable **=** expresion **;** {añadir_ins(id.nombre + ":=" expresion.nombre); sentencia.exit = lista_vacia();}
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M { completar_inst(expresion.true, M1.ref); completar_inst(expresion.false, M2.ref); sentencia.exit = lista_de_sentencias.exit; }
| **repetir** M **{** lista_de_sentencias **}** **hasta** expresion **;** M { completar_inst(expresion.false, M1.ref); completar_inst(expresion.true, M2.ref); completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia(); }
| M **repetir siempre {** lista_de_sentencias **}** { añadir_inst("goto"+M.ref); completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia(); }
| **salir si** expresion M **;** { completar_inst(expresion.false, M.ref); sentencia.exit.true = expresion.true; }
| **leer (** variable **) ;** {añadir_inst("read" + variable.nombre); sentencia.exit = lista_vacia();}
| **escribir_linea (** expresion **) ;** { añadir_inst("write" + expresion.nombre); añadir_inst("writeln"); sentencia.exit = lista_vacia(); }

M -> **ε** { M.ref = obtenref(); }

variable -> **IDENTIFICADOR** {variable.nombre = id.nombre;}

expresion: expresion **==** expresion
{expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);
añadir_inst("if" + expresion1.nombre + "==" + expresion2.nombre+ "goto");
añadir_inst(goto);}
| expresion **<** expresion
{expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);
añadir_inst("if" + expresion1.nombre + "<" + expresion2.nombre+ "goto");
añadir_inst(goto);}

```
        | expresion > expresion
        {expresion.true:= inilista(obtenref());
         expresion.false:=inilista(obtenref()+1);
         añadir_inst("if" + expresion1.nombre + ">" + expresion2.nombre+ "goto");
        añadir_inst(goto);}
        | expresion <= expresion
        {expresion.true:= inilista(obtenref());
         expresion.false:=inilista(obtenref()+1);
         añadir_inst("if" + expresion1.nombre + "<=" + expresion2.nombre+ "goto");
        añadir_inst(goto);}
        | expresion >= expresion
        {expresion.true:= inilista(obtenref());
         expresion.false:=inilista(obtenref()+1);
         añadir_inst("if" + expresion1.nombre + ">=" + expresion2.nombre+ "goto");
        añadir_inst(goto);}
        | expresion /= expresion
        {expresion.true:= inilista(obtenref());
         expresion.false:=inilista(obtenref()+1);
         añadir_inst("if" + expresion1.nombre + "/=" + expresion2.nombre+ "goto");
        añadir_inst(goto);}
        | expresion + expresion { expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"+"+expresion2.nombre); }
        | expresion - expresion { expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"-"+expresion2.nombre); }
        | expresion * expresion { expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"*"+expresion2.nombre); }
        | expresion / expresion { expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"/"+expresion2.nombre); }
        | IDENTIFICADOR { expresion.nombre = id.nombre; }
        | ENTERO { expresion.nombre = ent.nombre; }
        | REAL { expresion.nombre = real.nombre; }
        | ( expresion )  { expresion = expresion1; }
```

Casos de prueba

Programa	Salida
Pruebas que se espera que se compilen correctamente	

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *)
5.
6. procedimiento sumar (x,y: in entero; resul: in out entero)
7. variables aux: entero;
8. {
9. aux=x;
10. resul=y;
11. repetir {
12. aux = 1 - aux;
13. resul = resul+1;
14. } hasta aux == 0 ;
15. }
16.
17. {
18.
19. si a==0 entonces
20. {
21. a = a+1;
22. }
23.
24. }
	1: prog ejemplo;
	2: int a;
	3: int b;
	4: int c;
	5: real d;
	6: real e;
	7: proc sumar;
	8: val_int x;
	9: val_int y;
	10: ref_int resul;
	11: int aux;
	12: aux:=x;
	13: resul:=y;
	14: _t1:=1-aux;
	15: aux:=_t1;
	16: _t2:=resul+1;
	17: resul:=_t2;
	18: if aux==0 goto 20;
	19: goto 14;
	20: endproc;
	21: if a==0 goto 23;
	22: goto 25;
	23: _t3:=a+1;
	24: a:=_t3;
	25: halt;

1. programa ejemplo
2. variables a,b,c : entero;
3. variables e,d : real;
4. (* esto es un comentario *)
5. procedimiento sumar (x,y: in entero; resul: in out entero)
6. variables aux:entero;
7. {
8. aux=x;
9. resul=y;
10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
	1: prog ejemplo;
	2: int a;
	3: int b;
	4: int c;
	5: real e;
	6: real d;
	7: proc sumar;
	8: val_int x;
	9: val_int y;
	10: ref_int resul;
	11: int aux;
	12: aux:=x;
	13: resul:=y;
	14: _t1:=aux-1;

14. } 15. 16. { 17. leer(a); leer(b); 18. d= 1.0/e; 19. e= 1.0/d; 20. b= a*(c*b)+a; 21. escribir_linea(c*c); 22. 23. }	15: aux:=_t1; 16: _t2:=resul+1; 17: resul:=_t2; 18: if aux==0 goto 20; 19: goto 14; 20: endproc; 21: read a; 22: read b; 23: _t3:=1.0/e; 24: if e == 0 goto ERRORDIVNULL; 25: d:=_t3; 26: _t4:=1.0/d; 27: if d == 0 goto ERRORDIVNULL; 28: e:=_t4; 29: _t5:=c*b; 30: _t6:=a*_t5; 31: _t7:=_t6+a; 32: b:=_t7; 33: _t8:=c*c; 34: write _t8; 35: writeln; 36: halt;

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *)
5.
6. procedimiento sumar (x,y: in entero; resul: in out entero)
7. variables aux: entero;
8. {
9. aux=x+y;
10. resul=aux;
11. }
12.
13. {
14. leer(e); leer(a);
15. d= 1.0/e;
16. e= 1.0/(d*d);
17. c= c*((c+1)*a)+b;
18. escribir_linea(c*c);
19. }
	1: prog ejemplo;
	2: int a;
	3: int b;
	4: int c;
	5: real d;
	6: real e;
	7: proc sumar;
	8: val_int x;
	9: val_int y;
	10: ref_int resul;
	11: int aux;
	12: _t1:=x+y;
	13: aux:=_t1;
	14: resul:=aux;
	15: endproc;
	16: read e;
	17: read a;
	18: _t2:=1.0/e;
	19: if e == 0 goto ERRORDIVNULL;
	20: d:=_t2;
	21: _t3:=d*d;
	22: _t4:=1.0/_t3;
	23: if _t3 == 0 goto ERRORDIVNULL;
	24: e:=_t4;
	25: _t5:=c+1;
	26: _t6:=_t5*a;
	27: _t7:=c*_t6;

	28: _t8:=_t7+b; 29: c:=_t8; 30: _t9:=c*c; 31: write _t9; 32: writeln; 33: halt;

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *)
5.
6. {
7.
8. repetir siempre
9. {
10. a=a+1;
11. salir si a==0;
12. }
13.
14.
15. }
	1: prog ejemplo;
	2: int a;
	3: int b;
	4: int c;
	5: real d;
	6: real e;
	7: _t1:=a+1;
	8: a:=_t1;
	9: if a==0 goto 12;
	10: goto 11;
	11: goto 7;
	12: halt;
	ha finalizado...

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *)
5. procedimiento sumar (x,y: in entero;
resul: in out entero)
6. variables aux:entero;
7. {
8. aux=x;
9. resul=y;
10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
14. }
15. procedimiento prueba(a: in real;
resul: out real)
16. variables aux:real;
17. {
18. aux= a*5.0/a*a;
19. resul=aux-a;
20. }
21.
22.
23. {
24. leer(a); leer(b);
25. d= 1.0/e;
26. prueba(d*5.0, d);
	1: prog ejemplo;
	2: int a;
	3: int b;
	4: int c;
	5: real d;
	6: real e;
	7: proc sumar;
	8: val_int x;
	9: val_int y;
	10: ref_int resul;
	11: int aux;
	12: aux:=x;
	13: resul:=y;
	14: _t1:=aux-1;
	15: aux:=_t1;
	16: _t2:=resul+1;
	17: resul:=_t2;
	18: if aux==0 goto 20;
	19: goto 14;
	20: endproc;
	21: proc prueba;
	22: val_real a;
	23: ref_real resul;
	24: real aux;
	25: _t3:=a*5.0;
	26: _t4:=_t3/a;
	27: if a == 0 goto ERRORDIVNULL;
	28: _t5:=_t4*a;

27. e= 1.0/d; 28. sumar(a+5,b,c); (* los que hagan llamadas a procedimientos *) 29. c= c*(c*a)+b; 30. escribir_linea(c*c); 31. }	29: aux:=_t5; 30: _t6:=aux-a; 31: resul:=_t6; 32: endproc; 33: read a; 34: read b; 35: _t7:=1.0/e; 36: if e == 0 goto ERRORDIVNULL; 37: d:=_t7; 38: _t8:=d*5.0; 39: param_val _t8; 40: param_ref d; 41: call prueba; 42: _t9:=1.0/d; 43: if d == 0 goto ERRORDIVNULL; 44: e:=_t9; 45: _t10:=a+5; 46: param_val _t10; 47: param_val b; 48: param_ref c; 49: call sumar; 50: _t11:=c*a; 51: _t12:=c*_t11; 52: _t13:=_t12+b; 53: c:=_t13; 54: _t14:=c*c; 55: write _t14; 56: writeln; 57: halt;

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *)
5. procedimiento sumar (x,y: in entero; resul: in out entero)
6. variables aux:entero;
7. {
8. aux=x;
9. resul=y;
10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
14. }
15. {
16. leer(a); leer(b);
17. d= 1.0/e;
18. e= 1.0/d;
19. c= c*(c*a)+b;
	1: prog ejemplo;
	2: int a;
	3: int b;
	4: int c;
	5: real d;
	6: real e;
	7: proc sumar;
	8: val_int x;
	9: val_int y;
	10: ref_int resul;
	11: int aux;
	12: aux:=x;
	13: resul:=y;
	14: _t1:=aux-1;
	15: aux:=_t1;
	16: _t2:=resul+1;
	17: resul:=_t2;
	18: if aux==0 goto 20;
	19: goto 14;
	20: endproc;

20. escribir_linea(c*c); 21. }	21: read a; 22: read b; 23: _t3:=1.0/e; 24: if e == 0 goto ERRORDIVNULL; 25: d:=_t3; 26: _t4:=1.0/d; 27: if d == 0 goto ERRORDIVNULL; 28: e:=_t4; 29: _t5:=a+5; 30: param_val _t5; 31: param_val b; 32: param_ref c; 33: call sumar; 34: _t6:=c*a; 35: _t7:=c*_t6; 36: _t8:=_t7+b; 37: c:=_t8; 38: _t9:=c*c; 39: write _t9; 40: writeln; 41: halt;
Pruebas que se espera que den error de compilación	

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *) *)
5.
6. procedimiento sumar (x,y: in entero; resul: in out entero)
7. variables aux: entero;
8. {
9. aux=x;
10. resul=y;
11. repetir {
12. aux = 1 - aux;
13. resul = resul+1;
14. } hasta aux == 0 ;
15. }
16.
17. {
18. leer(a); leer(b);
19. d= 1/b;
20. e= 1/a;
21. c= c*(c*d)+e;
22. escribir_linea(c*c);
23. }
	Línea 4: syntax error, unexpected TMUL, expecting TKOPEN en '*'

Objetivo extra 1: Tratamiento de booleanos.

- Gramática

Los cambio respecto al ETDS anterior se marcan en **color verde:**

prog -> **programa IDENTIFICADOR**
declaraciones decl_de_subprogs
{ lista_de_sentencias **}**

declaraciones -> ε
| **variables** lista_de_ident **:** tipo **;** declaraciones

lista_de_ident -> **IDENTIFICADOR** resto_lista_id

resto_lista_id -> ε
| **,** **IDENTIFICADOR** resto_lista_id

tipo -> **entero**
| **real**

decl_de_subprogs -> ε
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
argumentos declaraciones
{ lista_de_sentencias **}**

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident : clase_par tipo resto_lis_de_param

clase_par -> **in**
| **out**
| **in out**

resto_lis_de_param -> ε
| **;** lista_de_ident **:** clase_par tipo resto_lis_de_param

lista_de_sentencias -> ε
| sentencia lista_de_sentencias

M -> ε

sentencia -> variable **=** expresion **;**
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M
| **repetir** M **{** lista_de_sentencias **}** **hasta** expresion **;** M
| M **repetir siempre {** lista_de_sentencias **}** M
| **salir si** expresion **;** M
| **leer (** variable **) ;**
| **escribir (** expresion **) ;**

variable -> **IDENTIFICADOR**
| **IDENTIFICADOR** array_acceso

expresion -> expresion **==** expresion
| expresion **<** expresion
| expresion **>** expresion
| expresion **<=** expresion
| expresion **>=** expresion
| expresion **/=** expresion
| expresion **+** expresion
| expresion **-** expresion
| expresion ***** expresion
| expresion **/** expresion
| expresion **or** M expresion
| expresion **and** M expresion
| **not** expresion
| **IDENTIFICADOR**
| **ENTERO**
| **REAL**
| **(** expresion **)**

list_expr -> expresion resto_lista_expr

resto_lista_expr -> ε
| **,** expresion resto_lista_expr

- ETDS

Los cambio respecto al ETDS anterior se marcan en **color verde:**

programa -> programa **IDENTIFICADOR** {añadir_inst("prog"+IDENTIFICADOR.nombre);}
declaraciones decl_de_subprogs { lista_de_sentencias }
{añadir_inst("halt");}

declaraciones -> **ε**

| **var** lista_de_ident **:** tipo **;**
{añadir_declaraciones(tipo.clase,lista_de_ident.lnom);} declaraciones

lista_de_ident -> **IDENTIFICADOR** resto_lista_id {lista_de_ident.lnom:=lista_vacia();
lista_de_ident.lnom:=añadir(resto_lista_id.lnom,IDENTIFICADOR.nombre);}

resto_lista_id -> ε {resto_lista_id.lnom:=lista_vacia();}
| **,** **IDENTIFICADOR** resto_lista_id
{resto_lista_id.lnom:=añadir(resto_lista_id2.lnom,IDENTIFICADOR.nombre);}

tipo -> **entero** {tipo.clase="int";}
| **real** {tipo.clase="real";}

decl_de_subprogs -> ε
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
{añadir_inst("proc"+IDENTIFICADOR.nombre);}
argumentos
declaraciones
{ lista_de_sentencias **}** {añadir_inst("endproc");}

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident **:** clase_par tipo {añadir_parametros(clase_par.tipo,
tipo.clase, lista_de_ident.lnom);}
resto_lis_de_param

clase_par -> **in** {clase_par.tipo=”val_”;}
| **out** {clase_par.tipo=”ref_”;}
| **in out** {clase_par.tipo=”ref_”;}

resto_lis_de_param: ε
| **;** lista_de_ident : clase_par tipo { añadir_parametros(clase_par.tipo,
tipo.clase, lista_de_ident.lnom); } resto_lis_de_param

lista_de_sentencias -> ε
{ lista_de_sentencias.exit = lista_vacia(); }

| sentencia lista_de_sentencias { lista_de_sentencias.exit = unir(sentencia.exit, lista_de_sentencias.exit); }

sentencia -> variable **=** expresion **;** {añadir_ins(id.nombre + ":=" expresion.nombre); sentencia.exit = lista_vacia();}
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M { completar_inst(expresion.true, M1.ref); completar_inst(expresion.false, M2.ref); sentencia.exit = lista_de_sentencias.exit; }
| **repetir** M **{** lista_de_sentencias **}** **hasta** expresion **;** M { completar_inst(expresion.false, M1.ref); completar_inst(expresion.true, M2.ref); completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia(); }
| M **repetir siempre {** lista_de_sentencias **}** { añadir_inst("goto"+M.ref); completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia(); }
| **salir si** expresion M **;** { completar_inst(expresion.false, M.ref); sentencia.exit.true = expresion.true; }
| **leer (** variable **) ;** {añadir_inst("read" + variable.nombre); sentencia.exit = lista_vacia();}
| **escribir_linea (** expresion **) ;** { añadir_inst("write" + expresion.nombre); añadir_inst("writeln"); sentencia.exit = lista_vacia(); }

M -> ε { M.ref = obtenref(); }

variable -> **IDENTIFICADOR** {variable.nombre = id.nombre;}

expresion: expresion **==** expresion
{expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);
añadir_inst("if" + expresion1.nombre + "==" + expresion2.nombre+ "goto");
añadir_inst(goto);}
| expresion **<** expresion
{expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);
añadir_inst("if" + expresion1.nombre + "<" + expresion2.nombre+ "goto");
añadir_inst(goto);}
| expresion **>** expresion
{expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);
añadir_inst("if" + expresion1.nombre + ">" + expresion2.nombre+ "goto");
añadir_inst(goto);}
| expresion **<=** expresion
{expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);
añadir_inst("if" + expresion1.nombre + "<=" + expresion2.nombre+ "goto");
añadir_inst(goto);}
| expresion **>=** expresion
{expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);

```
        añadir_inst("if" + expresion1.nombre + ">=" + expresion2.nombre+ "goto");
        añadir_inst(goto);}
        | expresion /= expresion
        {expresion.true:= inilista(obtenref());
        expresion.false:=inilista(obtenref()+1);
        añadir_inst("if" + expresion1.nombre + "/=" + expresion2.nombre+ "goto");
        añadir_inst(goto);}
        | expresion + expresion { expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"+"+expresion2.nombre); }
        | expresion - expresion { expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"-"+expresion2.nombre); }
        | expresion * expresion { expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"*"+expresion2.nombre); }
        | expresion / expresion { expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"/"+expresion2.nombre); }
        | expresion or M expresion    {completar (E1.false, M.ref);
                                       E.true = unir(E1.true, E2.true);
                                       E.false = E2.false;}
        | expresion and M expresion   {completar(E1.true,M.ref);
                                       E1.true=E2.true;
                                       E1.false = unir(E1.false,E2.false);}
        | not expresion               {E.true= E1.false;
                                       E.false = E1.true;}
        | IDENTIFICADOR { expresion.nombre = id.nombre; }
        | ENTERO { expresion.nombre = ent.nombre; }
        | REAL { expresion.nombre = real.nombre; }
        | ( expresion )  { expresion = expresion1; }
```

- Casos de prueba

Programa	Salida
Pruebas que se espera que se compilen correctamente	

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *)
5.
6.
7. {
8.
9. si a==1 or b/1+c==2 entonces
10. {
11. d=1.0;
12. }
13.
14. si b*9<1 and c-1==a entonces
15. {
16. e=d;
17. }
18.
19. repetir
20. {
21. a = a+1;
22. } hasta not a<10;
23.
24.
25. }
	1: prog ejemplo;
	2: int a;
	3: int b;
	4: int c;
	5: real d;
	6: real e;
	7: if a==1 goto 13;
	8: goto 9;
	9: _t1:=b/1;
	10: _t2:=_t1+c;
	11: if _t2==2 goto 13;
	12: goto 14;
	13: d:=1.0;
	14: _t3:=b*9;
	15: if _t3<1 goto 17;
	16: goto 21;
	17: _t4:=c-1;
	18: if _t4==a goto 20;
	19: goto 21;
	20: e:=d;
	21: _t5:=a+1;
	22: a:=_t5;
	23: if a<10 goto 21;
	24: goto 25;
	25: halt;

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *)
5.
6.
7. {
8.
9. si not(a==1 or b/1+c==2) entonces
10. {
11. d=1.0;
12. }
13.
14. si not b*9<1 and not c-1==a entonces
	1: prog ejemplo;
	2: int a;
	3: int b;
	4: int c;
	5: real d;
	6: real e;
	7: if a==1 goto 14;
	8: goto 9;
	9: _t1:=b/1;
	10: _t2:=_t1+c;
	11: if _t2==2 goto 14;
	12: goto 13;
	13: d:=1.0;
	14: _t3:=b*9;
	15: if _t3<1 goto 21;

15. {
16. e=d;
17. }
18.
19. repetir
20. {
21. a = a+1;
22. } hasta a<10;
23.
24.
25. }
	16: goto 17;
	17: _t4:=c-1;
	18: if _t4==a goto 21;
	19: goto 20;
	20: e:=d;
	21: _t5:=a+1;
	22: a:=_t5;
	23: if a<10 goto 25;
	24: goto 21;
	25: halt;

Pruebas que se espera que den error de compilación	

1. programa ejemplo
2. variables a,b,c : entero;
3. variables d,e : real;
4. (* esto es un comentario *)
5.
6.
7. {
8.
9. a = a>b;
10.
11.
12.
13.
14. si not a+4 entonces
15. {
16. a=a+1;
17. }
18. }
	**Nota: Estas líneas de error pertenecen al objetivo “Restricciones semánticas”, pero aquí se puede comprobar que efectivamente el programa da error de compilación.
	
	Error en línea 9: Se ha intentado asignar un bool a la variable a de tipo int.
	Error en línea 14: Se esperaba un bool y se ha recibido un int.

Diseño y funcionalidad de la tabla de símbolos:

En la tabla de símbolos almacenamos la siguiente información sobre cada símbolo:

- tipoId: Es un String donde se especifica si un símbolo es una variable o un procedimiento.
- tipoVar: Es un string. Si el símbolo es una variable se almacena su tipo, entero o real.
- parametrosProc: Un vector de clases de parámetros. Si el símbolo es un procedimiento, aquí se almacenan las clases de sus parámetros.
- tipoElemtsArray: Un string. Si el símbolo es un array indica si sus elementos son de tipo entero o real.
- dimensiones: Un vector de enteros. Si el símbolo es un array indica el tamaño de cada una de sus dimensiones. El tamaño de este vector indica las dimensiones del array.

Las funciones definidas para el tratamiento de la tabla de símbolos son las siguientes:

- añadirVariable(string id, string tipo, vector<int> dimensiones): Añade un símbolo de tipo variable y su tipo (entero o real).
- añadirArray(string id, string tipo, vector<int> dimensiones): Añade un símbolo de tipo array, su tipo (entero o real) y sus dimensiones.
- añadirProcedimiento (string id). Añade un símbolo de tipo procedimiento.
- añadirParametro(string id, string clasePar, string tipoVar): Añade un parámetro y su tipo a un procedimiento ya añadido.
- añadirParametro(string id, string ClasePar, string tipoVar, tipoElemtsArray, vector<int> dimensiones): Añade un parámetro de tipo array y si tipo a un procedimiento ya añadido.
- obtenerTipo(string id): Obtiene el tipo de una variable ya ñadida.
- obtenerTipoElemts(string id): Obtiene el tipo de los elementos de un array ya añadido.
- obtenerDimensiones(string id): Obtiene las dimensiones de un array ya añadido.
- obtenerTiposParametro(string id, string numParametro): Devuelve el número de parámetros correspondiente a un procedimiento ya añadido.
- existeId(string id): Dado un Id nos dice si está definido en la tabla de símbolos o no.
- borrarId(string id): Borra un Id de la tabla de símbolos.

La tabla de símbolos tiene varias funcionalidades. Es la herramienta que permite implementar las restricciones semánticas. Para las llamadas a procedimientos nos permite tener almacenados los procedimientos y parámetros para comprobar si la llamada se está realizando correctamente. Y para el tratamiento de arrays nos permite también almacenar las dimensiones del array para poder calcular la dirección a la que tenemos que acceder y comprobar si el acceso al array se está realizando correctamente.

Objetivo extra 2: Restricciones semántica

● Descripción

Se han realizado las siguientes restricciones semánticas:
Comprobaciones estáticas:

- Restricciones de tipos (el lenguaje es fuertemente tipado).
- Comprobación de que no haya variables duplicadas.
- Comprobación de que no haya procedimientos duplicados.
- Comprobación de que no haya parámetros duplicados en un mismo procedimiento.
- Comprobación de operaciones entre expresiones del mismo tipo (lenguaje fuertemente tipado).
- Comprobación de no aplicar operadores aritméticos a expresiones booleanas.
- Comprobación de no aplicar operadores booleanos a expresiones que no sean booleanas.
- Comprobación de no división entre 0 con expresiones constantes.
- No puede haber un “salir si” fuera de un bucle.

Comprobaciones dinámicas:

- Comprobación de no realizar una división entre 0.

● ETDS

Los cambio respecto al ETDS anterior se marcan en **color verde:**
*** Por defecto imprimeError() imprimirá el error correspondiente y causará que no se escriba la traducción.**
*** Por defecto si hay alguna id de variable, procedimiento, … etc, se comprueba que exista y si no existe se hace un imprimeError().**
*** Por defecto si no se especifica:**
· Las expresiones tienen tipo *error* al crearse.
· El flag esConstante es *false* si no se asigna.

programa -> programa **IDENTIFICADOR** {añadir_inst("prog"+IDENTIFICADOR.nombre);}
declaraciones decl_de_subprogs { lista_de_sentencias }
{añadir_inst("halt");}

declaraciones -> ε
| **var** lista_de_ident **:** tipo **;**
{añadir_declaraciones(tipo.clase,lista_de_ident.lnom);
tablaSimbolosActual.añadirVariable(lista_de_ident.lnom, tipo.clase);} declaraciones

lista_de_ident -> **IDENTIFICADOR** resto_lista_id {lista_de_ident.lnom:=lista_vacia();
lista_de_ident.lnom:=añadir(resto_lista_id.lnom,IDENTIFICADOR.nombre);}

resto_lista_id -> ε {resto_lista_id.lnom:=lista_vacia();}
| **,** **IDENTIFICADOR** resto_lista_id
{resto_lista_id.lnom:=añadir(resto_lista_id2.lnom,IDENTIFICADOR.nombre);}

tipo -> **entero** {tipo.clase="int";}
| **real** {tipo.clase="real";}

decl_de_subprogs -> ε
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
{añadir_inst("proc"+IDENTIFICADOR.nombre);
tablaSimbolosActual.añadirProcedimiento(IDENTIFICADOR.nombre); tablaSimbolosAnt =
tablaSimbolosActual; tablaSimbolosActual = nueva tablaSimbolos();}
argumentos
declaraciones
{ lista_de_sentencias **}** {añadir_inst("endproc"); tablaSimbolosActual =
tablaSimbolosAnt;}

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident **:** clase_par tipo {añadir_parametros(clase_par.tipo,
tipo.clase, lista_de_ident.lnom); tablasSimbolos.añadirParametros(lista_de_ident.lnom,
clase_par.tipo, tipo.clase);}
resto_lis_de_param

clase_par -> **in** {clase_par.tipo=”val_”;}
| **out** {clase_par.tipo=”ref_”;}
| **in out** {clase_par.tipo=”ref_”;}

resto_lis_de_param: ε

| **;** lista_de_ident : clase_par tipo { añadir_parametros(clase_par.tipo, tipo.clase, lista_de_ident.lnom); tablasSimbolos.añadirParametros(lista_de_ident.lnom, clase_par.tipo, tipo.clase);} resto_lis_de_param

lista_de_sentencias -> **ε**
{ lista_de_sentencias.exit = lista_vacia(); }
| sentencia lista_de_sentencias { lista_de_sentencias.exit = unir(sentencia.exit, lista_de_sentencias.exit); }

sentencia -> variable **=** expresion **;** { if (expresion.tipo == “bool”) imprimeError();
if (variable.tipo != expresion.tipo) imprimeError();
añadir_ins(id.nombre + “:=” expresion.nombre); sentencia.exit = lista_vacia();}
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M { completar_inst(expresion.true, M1.ref); completar_inst(expresion.false, M2.ref); sentencia.exit = lista_de_sentencias.exit; }
| **repetir** {contWhile++} M **{** lista_de_sentencias **} hasta** expresion **;** M {
if (expresion.tipo != “bool”) imprimeError();
completar_inst(expresion.false, M1.ref); completar_inst(expresion.true, M2.ref);
completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia();contWhile--; }
| M **repetir** {contWhile++} **siempre {** lista_de_sentencias **}** {
añadir_inst(“goto”+M.ref); completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia();contWhile--; }
| **salir si** expresion M **;** {if (contWhile == 0) imprimeError();
if (expresion.tipo != “bool”) imprimeError();
completar_inst(expresion.false, M.ref); sentencia.exit.true = expresion.true;
}
| **leer (** variable **) ;** {añadir_inst(“read” + variable.nombre); sentencia.exit = lista_vacia();}
| **escribir_linea (** expresion **) ;** { if (expresion.tipo == “bool”) imprimeError();
añadir_inst(“write” + expresion.nombre); añadir_inst(“writeln”); sentencia.exit = lista_vacia();
}

M -> **ε** { M.ref = obtenref(); }

variable -> **IDENTIFICADOR** {variable.nombre = id.nombre; variable.tipo = tablaSimbolosActual.buscarTipo(id.nombre);}

expresion: expresion **==** expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);
añadir_inst(“if” + expresion1.nombre + ”==” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion **<** expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();

```
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”<” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion > expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”>” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion <= expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”<=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion >= expresion
{ if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”>=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion /= expresion
{ if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”/=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion + expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”+”+expresion2.nombre); }
| expresion - expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”-”+expresion2.nombre); }
```

| expresion * expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”*”+expresion2.nombre); }
| expresion / expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo;
if (E2.esConstante) { if (valor(E2) == 0) imprimeError(); }
else {añadir_inst(“if ”+E2.nombre+” == 0 goto ERRORDIVNULL”);}
expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”/”+expresion2.nombre); }
| expresion **or** M expresion { if (E1.tipo != “bool” or E2.tipo != “bool”) imprimeError();
completar (E1.false, M.ref);
E.true = unir(E1.true, E2.true);
E.false = E2.false;}
| expresion **and** M expresion { if (E1.tipo != “bool” or E2.tipo != “bool”) imprimeError();
completar(E1.true,M.ref);
E1.true=E2.true;
E1.false = unir(E1.false,E2.false);}
| **not** expresion { if (E.tipo != “bool”) imprimeError();
E.true= E1.false;
E.false = E1.true;}
| **IDENTIFICADOR** { expresion.nombre = id.nombre; expresion.tipo =
tablaSimbolosActual.buscarTipo(id.nombre); expresion.esConstante = false; }
| **ENTERO** { expresion.nombre = ent.nombre; expresion.tipo = “int”;
expresion.esConstante = true; }
| **REAL** { expresion.nombre = real.nombre; expresion.tipo = “real”;
expresion.esConstante = true; }
| **(** expresion **)** { expresion = expresion1; }

- Casos de prueba

Programa	Salida
Pruebas que se espera que se compilen correctamente	
Mismo id declarado en un procedimiento y en el programa principal (“a”): 1. programa ejemplo 2. variables a,b,c : entero; 3. variables e,d : real; 4. (* esto es un comentario *) 5. procedimiento sumar (x,y: in entero; resul: in out entero)	1: prog ejemplo; 2: int a; 3: int b; 4: int c; 5: real e; 6: real d; 7: proc sumar; 8: val_int x;

6. variables a,aux:entero;
7. {
8. a=x;
9. resul=y;
10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
14. }
15.
16. {
17. leer(a); leer(b);
18. d= 1.0/e;
19. e= 1.0/d;
20. b= a*(c*b)+a;
21. escribir_linea(c*c);
22. }
	9: val_int y;
	10: ref_int resul;
	11: int a;
	12: int aux;
	13: a:=x;
	14: resul:=y;
	15: _t1:=aux-1;
	16: aux:=_t1;
	17: _t2:=resul+1;
	18: resul:=_t2;
	19: if aux==0 goto 21;
	20: goto 15;
	21: endproc;
	22: read a;
	23: read b;
	24: _t3:=1.0/e;
	25: if e == 0 goto ERRORDIVNULL;
	26: d:=_t3;
	27: _t4:=1.0/d;
	28: if d == 0 goto ERRORDIVNULL;
	29: e:=_t4;
	30: _t5:=c*b;
	31: _t6:=a*_t5;
	32: _t7:=_t6+a;
	33: b:=_t7;
	34: _t8:=c*c;
	35: write _t8;
	36: writeln;
	37: halt;

Pruebas que se espera que den error de compilación	

Id duplicado (en declaraciones del programa):
1. programa ejemplo
2. variables a,b,c : entero;
3. variables a,d,e : real;
4. (* esto es un comentario *)
5. procedimiento sumar (x,y: in entero; resul: in out entero)
6. variables aux:entero;
7. {
8. aux=x;
9. resul=y;
10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
14. }
15.
	Error en línea 3: La variable a ya ha sido declarada anteriormente.

16. { 17. leer(a); leer(b); 18. d= 1.0/e; 19. e= 1.0/d; 20. b= a*(c*b)+a; 21. escribir_linea(c*c); 22. 23. }	

Id Duplicado (en procedimiento):
1. programa ejemplo
2. variables a,b,c : entero;
3. variables e,d : real;
4. (* esto es un comentario *)
5. procedimiento sumar (aux,y: in entero; resul: in out entero)
6. variables aux:entero;
7. {
8. aux=resul;
9. resul=y;
10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
14. }
15.
16. {
17. leer(a); leer(b);
18. d= 1.0/e;
19. e= 1.0/d;
20. b= a*(c*b)+a;
21. escribir_linea(c*c);
22.
23. }
	Error en línea 6: La variable aux ya ha sido declarada anteriormente.

Id duplicado (parámetro procedimiento): 1. programa ejemplo 2. variables a,b,c : entero; 3. variables e,d : real; 4. (* esto es un comentario *) 5. procedimiento sumar (x,y: in entero; resul,x: in out entero) 6. variables a,aux:entero; 7. { 8. a=x; 9. resul=y;	Error en línea 5: El parámetro x ya ha sido declarado anteriormente en el procedimiento sumar.

10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
14. }
15.
16. {
17. leer(a); leer(b);
18. d= 1.0/e;
19. e= 1.0/d;
20. b= a*(c*b)+a;
21. escribir_linea(c*c);
22.
23. }	

Procedimiento duplicado:
1. programa ejemplo
2. variables a,b,c : entero;
3. variables e,d : real;
4. (* esto es un comentario *)
5. procedimiento sumar (x,y: in entero; resul: in out entero)
6. variables a,aux:entero;
7. {
8. a=x;
9. resul=y;
10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
14. }
15.
16. procedimiento sumar (x,y: in entero; resul: in out entero)
17. variables a,aux:entero;
18. {
19. a=x;
20. resul=y;
21. repetir {
22. aux = aux - 1;
23. resul = resul+1;
24. } hasta aux == 0 ;
25. }
26.
27.
28. {
29. leer(a); leer(b);
30. d= 1.0/e;
	Error en línea 16: El procedimiento sumar ya ha sido declarado anteriormente.

31. e= 1.0/d; 32. b= a*(c*b)+a; 33. escribir_linea(c*c); 34. 35. }	

Errores de tipos variados:
1. programa ejemplo
2. variables a,b,c : entero;
3. variables e,d : real;
4. (* esto es un comentario *)
5.
6. {
7.
8.
9. e = a+b;
10. a = e;
11. a = a<c;
12.
13. e = e+1;
14. e = e+1.0;
15.
16. e = 1*(a<4);
17. e = (a<4) + (a<4);
18. e = a==4;
19.
20. si a<3 or a+4 entonces
21. {
22. a=a+1;
23. }
24.
25. si a+3 entonces
26. {
27. a=a+1;
28. }
29.
30. IdQueNoExiste = 1.0;
31.
32. }
33.
	Error en línea 9: Se ha intentado asignar un int a la variable e de tipo real.
	Error en línea 10: Se ha intentado asignar un real a la variable a de tipo int.
	Error en línea 11: Se ha intentado asignar un bool a la variable a de tipo int.
	Error en línea 13: Se ha intentado operar un real con un int.
	Error en línea 13: Se ha intentado asignar un error a la variable e de tipo real.
	Error en línea 16: Se ha intentado operar un int con un bool.
	Error en línea 16: Se ha intentado asignar un error a la variable e de tipo real.
	Error en línea 17: Se intenta realizar una operación aritmética entre expresiones booleanas.
	Error en línea 17: Se ha intentado asignar un error a la variable e de tipo real.
	Error en línea 18: Se ha intentado asignar un bool a la variable e de tipo real.
	Error en línea 20: Se intenta realizar una operación booleana entre expresiones que no son booleanas.
	Error en línea 28: Se esperaba un bool y se ha recibido un int.
	Error en línea 30: La variable IdQueNoExiste no existe.
	Error en línea 30: Se ha intentado asignar un real a la variable IdQueNoExiste de tipo null.

Salir si fuera de bucle: 1. (* Probando el salir si fuera de repetir *) 2. programa ejemplo 3. variables a,b,c : entero; 4. variables d,e : real; 5. (* esto es un comentario *)	Error en línea 8: Hay un salir si fuera de un bucle.

6.
7. {
8. salir si a==0;
9. repetir siempre
10. {
11. a=a+1;
12.
13. }
14.
15.
16. }	

Objetivo extra 3: Llamadas a procedimientos

● Restricciones semánticas añadidas

- Llamada a procedimiento que no existe.
- Comprobación que el número de parámetros de la llamada y del procedimiento sean los mismos.
- Comprobación de que los tipos de los parámetros de la llama y del procedimiento coinciden (entero o real).
- Los parámetros en la llamada que se refieren a parámetros de tipo “out” e “in_out” deben ser variables (id).

● Gramática

prog -> **programa IDENTIFICADOR**
declaraciones decl_de_subprogs
{ lista_de_sentencias **}**

declaraciones -> ε
| **variables** lista_de_ident **:** tipo **;** declaraciones

lista_de_ident -> **IDENTIFICADOR** resto_lista_id

resto_lista_id -> ε
| **,** **IDENTIFICADOR** resto_lista_id

tipo -> **entero**
| **real**

decl_de_subprogs -> ε
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
argumentos declaraciones
{ lista_de_sentencias **}**

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident : clase_par tipo resto_lis_de_param

clase_par -> **in**

| **out**
| **in out**

resto_lis_de_param -> ε
| **;** lista_de_ident **:** clase_par tipo resto_lis_de_param

lista_de_sentencias -> ε
| sentencia lista_de_sentencias

M -> ε

sentencia -> variable **=** expresion **;**
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M
| **repetir** M **{** lista_de_sentencias **}** **hasta** expresion **;** M
| M **repetir siempre {** lista_de_sentencias **}** M
| **salir si** expresion **;** M
| **leer (** variable **) ;**
| **escribir (** expresion **) ;**
| **IDENTIFICADOR (** list_expr **) ;**

variable -> **IDENTIFICADOR**

expresion -> **not** expresion
| expresion **or** M expresion
| expresion **and** M expresion
| expresion **==** expresion
| expresion **<** expresion
| expresion **>** expresion
| expresion **<=** expresion
| expresion **>=** expresion
| expresion **/=** expresion
| expresion **+** expresion
| expresion **-** expresion
| expresion ***** expresion
| expresion **/** expresion
| **IDENTIFICADOR**
| **ENTERO**
| **REAL**
| **(** expresion **)**

list_expr -> expresion resto_lista_expr

resto_lista_expr -> ε
| **,** expresion resto_lista_expr

- ETDS

Los cambio respecto al ETDS anterior se marcan en **color verde:**
*** Por defecto imprimeError() imprimirá el error correspondiente y causará que no se escriba la traducción.**
*** Por defecto si hay alguna id de variable, procedimiento, … etc, se comprueba que exista y si no existe se hace un imprimeError().**
*** Por defecto si no se especifica:**
· Las expresiones tienen tipo *error* al crearse.
· El flag esConstante es *false* si no se asigna.
· El flag esVar solo es *true* si la expresión es un identificador.

programa -> programa **IDENTIFICADOR** {añadir_inst("prog"+IDENTIFICADOR.nombre);}
declaraciones decl_de_subprogs { lista_de_sentencias }
{añadir_inst("halt");}

declaraciones -> **ε**
| **var** lista_de_ident **:** tipo **;**
{añadir_declaraciones(tipo.clase,lista_de_ident.lnom);
tablaSimbolosActual.añadirVariable(lista_de_ident.lnom, tipo.clase);} declaraciones

lista_de_ident -> **IDENTIFICADOR** resto_lista_id {lista_de_ident.lnom:=lista_vacia();
lista_de_ident.lnom:=añadir(resto_lista_id.lnom,IDENTIFICADOR.nombre);}

resto_lista_id -> **ε** {resto_lista_id.lnom:=lista_vacia();}
| **,** **IDENTIFICADOR** resto_lista_id
{resto_lista_id.lnom:=añadir(resto_lista_id2.lnom,IDENTIFICADOR.nombre);}

tipo -> **entero** {tipo.clase="int";}
| **real** {tipo.clase="real";}

decl_de_subprogs -> **ε**
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
{añadir_inst("proc"+IDENTIFICADOR.nombre);
tablaSimbolosActual.añadirProcedimiento(IDENTIFICADOR.nombre); tablaSimbolosAnt =
tablaSimbolosActual; tablaSimbolosActual = nueva tablaSimbolos();}
argumentos
declaraciones

{ lista_de_sentencias **}** {añadir_inst("endproc"); tablaSimbolosActual = tablaSimbolosAnt;}

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident **:** clase_par tipo {añadir_parametros(clase_par.tipo, tipo.clase, lista_de_ident.lnom); tablasSimbolos.añadirParametros(lista_de_ident.lnom, clase_par.tipo, tipo.clase);}
resto_lis_de_param

clase_par -> **in** {clase_par.tipo="val_";}
| **out** {clase_par.tipo="ref_";}
| **in out** {clase_par.tipo="ref_";}

resto_lis_de_param: ε
| **;** lista_de_ident : clase_par tipo { añadir_parametros(clase_par.tipo, tipo.clase, lista_de_ident.lnom); tablasSimbolos.añadirParametros(lista_de_ident.lnom, clase_par.tipo, tipo.clase);} resto_lis_de_param

lista_de_sentencias -> ε
{ lista_de_sentencias.exit = lista_vacia(); }
| sentencia lista_de_sentencias { lista_de_sentencias.exit = unir(sentencia.exit, lista_de_sentencias.exit); }

sentencia -> variable **=** expresion **;** { if (expresion.tipo == "bool") imprimeError();
if (variable.tipo != expresion.tipo) imprimeError();
añadir_ins(id.nombre + ":=" expresion.nombre); sentencia.exit = lista_vacia();}
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M { completar_inst(expresion.true, M1.ref); completar_inst(expresion.false, M2.ref); sentencia.exit = lista_de_sentencias.exit; }
| **repetir** {contWhile++} M **{** lista_de_sentencias **}** **hasta** expresion **;** M {
if (expresion.tipo != "bool") imprimeError();
completar_inst(expresion.false, M1.ref); completar_inst(expresion.true, M2.ref);
completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia();contWhile--; }
| M **repetir siempre** {contWhile++} **{** lista_de_sentencias **}** {
añadir_inst("goto"+M.ref); completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia();contWhile--; }
| **salir si** expresion M **;** {if (contWhile == 0) imprimeError(); if (expresion.tipo != "bool") imprimeError(); completar_inst(expresion.false, M.ref); sentencia.exit.true = expresion.true;
}
| **leer (** variable **) ;** {añadir_inst("read" + variable.nombre); sentencia.exit = lista_vacia();}
| **escribir_linea (** expresion **) ;** { if (expresion.tipo == "bool") imprimeError();

```
añadir_inst("write" + expresion.nombre); añadir_inst("writeln"); sentencia.exit = lista_vacia();
}
        | IDENTIFICADOR ( list_expr ) ;
        { params = tablaSimbolosActual.obtenerParams(IDENTIFICADOR.nombre;
        if (params.numParams != list_expr.list_e.size()) imprimeError();
        else
        {
                for (elems_in(param) = param_actual and elems_in(list_expr.list_e) = expr)
                {
                        if (param_actual.clase = "ref" and !expr.esVar) imprimeError();
                        else if (param_actual.tipo != expr.tipo) imprimeError();
                        else
                        {
                                añadir_inst("param_"+param_actual.clase+" "+expr.nombre);
                        }
                }
                añadir_inst("call "+IDENTIFICADOR.nombre);
         }

list_expr -> expresion resto_lista_expr {añadir_elemts(list_expr.list_e,
resto_lista_expr.list_e); añadir_elemento(list_expr.list_e, expresion); }

resto_lista_expr -> ε { resto_lista_expr.list_e = inilistaexpr(); }
        | , expresion resto_lista_expr
{añadir_elemts(resto_lista_expr.list_e, resto_lista_expr2.list_e);
añadir_elemento(resto_lista_expr.list_e, expresion); }

M -> ε { M.ref = obtenref(); }

variable -> IDENTIFICADOR {variable.nombre = id.nombre; variable.tipo =
tablaSimbolosActual.buscarTipo(id.nombre);}

expresion: expresion == expresion
        {if (E1.tipo == "bool" or E2.tipo == "bool") imprimeError();
        if (E1.tipo != E2.tipo) { imprimeError(); }
        else E.tipo = "bool";
        expresion.true:= inilista(obtenref());
         expresion.false:=inilista(obtenref()+1);
         añadir_inst("if" + expresion1.nombre + "==" + expresion2.nombre+ "goto");
        añadir_inst(goto);}
        | expresion < expresion
        {if (E1.tipo == "bool" or E2.tipo == "bool") imprimeError();
        if (E1.tipo != E2.tipo) { imprimeError(); }
        else E.tipo = "bool";
        expresion.true:= inilista(obtenref());
         expresion.false:=inilista(obtenref()+1);
```

```
 añadir_inst(“if” + expresion1.nombre + ”<” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion > expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”>” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion <= expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”<=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion >= expresion
{ if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”>=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion /= expresion
{ if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”/=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion + expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”+”+expresion2.nombre); }
| expresion - expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”-”+expresion2.nombre); }
| expresion * expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”*”+expresion2.nombre); }
```

| expresion **/** expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo;
if (E2.esConstante) { if (valor(E2) == 0) imprimeError(); }
else {añadir_inst(“if ”+E2.nombre+” == 0 goto ERRORDIVNULL”);}
expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”/”+expresion2.nombre); }

| expresion **or** M expresion { if (E1.tipo != “bool” or E2.tipo != “bool”) imprimeError();
completar (E1.false, M.ref);
E.true = unir(E1.true, E2.true);
E.false = E2.false;}

| expresion **and** M expresion { if (E1.tipo != “bool” or E2.tipo != “bool”) imprimeError();
completar(E1.true,M.ref);
E1.true=E2.true;
E1.false = unir(E1.false,E2.false);}

| **not** expresion { if (E.tipo != “bool”) imprimeError();
E.true= E1.false;
E.false = E1.true;}

| **IDENTIFICADOR** { expresion.nombre = id.nombre; expresion.tipo =
tablaSimbolosActual.buscarTipo(id.nombre); expresion.esConstante = false; }

| **ENTERO** { expresion.nombre = ent.nombre; expresion.tipo = “int”;
expresion.esConstante = true; }

| **REAL** { expresion.nombre = real.nombre; expresion.tipo = “real”;
expresion.esConstante = true; }

| **(** expresion **)** { expresion = expresion1; }

- Casos de prueba

Programa	Salida
Pruebas que se espera que se compilen correctamente	
1. programa ejemplo 2. variables a,b,c : entero; 3. variables e,d : real; 4. (* esto es un comentario *) 5. procedimiento sumar (x,y: in entero; b: out entero; resul: in out entero) 6. variables a,aux:entero; 7. { 8. a=x+y; 9. } 10. 11. 12. 13. procedimiento restar (x,y: in real; b:	1: prog ejemplo; 2: int a; 3: int b; 4: int c; 5: real e; 6: real d; 7: proc sumar; 8: val_int x; 9: val_int y; 10: ref_int b; 11: ref_int resul; 12: int a; 13: int aux; 14: _t1:=x+y;

out entero; resul: in out entero)
14. variables a,aux:real;
15. {
16. a=x-y;
17. }
18.
19.
20. {
21.
22. sumar(a*1+2,1,c,b);
23. restar(e/1.0,1.0,c,b);
24.
25. }
	15: a:=_t1;
	16: endproc;
	17: proc restar;
	18: val_real x;
	19: val_real y;
	20: ref_int b;
	21: ref_int resul;
	22: real a;
	23: real aux;
	24: _t2:=x-y;
	25: a:=_t2;
	26: endproc;
	27: _t3:=a*1;
	28: _t4:=_t3+2;
	29: param_val _t4;
	30: param_val 1;
	31: param_ref c;
	32: param_ref b;
	33: call sumar;
	34: _t5:=e/1.0;
	35: param_val _t5;
	36: param_val 1.0;
	37: param_ref c;
	38: param_ref b;
	39: call restar;
	40: halt;

Pruebas que se espera que den error de compilación	

Errores variados en las llamadas
1. programa ejemplo
2. variables a,b,c : entero;
3. variables e,d : real;
4. (* esto es un comentario *)
5. procedimiento sumar (x,y: in entero; b: out entero; resul: in out entero)
6. variables a,aux:entero;
7. {
8. a=x;
9. }
10. {
11.
12. sumar(a,1,1,1);
13. sumar(a,1,1,1);
14. sumar(a,b,c,d,e);
15. sumar(1);
16. sumar(a,e,b,b);
17. sumar();
18. ProcQueNoExiste(1,1,1,1,1,1,1,1);
19. sumar(a=a*1+2,1,c,b);
	Error en línea 12: La referencia 1 no es una variable.
	Error en línea 12: La referencia 1 no es una variable.
	Error en línea 13: La referencia 1 no es una variable.
	Error en línea 13: La referencia 1 no es una variable.
	Error en línea 14: El numero de argumentos en la llamada al procedimiento sumar no coincide con los requeridos.
	Error en línea 15: El numero de argumentos en la llamada al procedimiento sumar no coincide con los requeridos.
	Error en línea 16: Se esperaba un int y se ha recibido un real.
	Error en línea 17: El numero de argumentos en la llamada al procedimiento sumar no coincide con los requeridos.
	Error en línea 18: El procedimiento ProcQueNoExiste no existe.

20. 21. }	Línea 19: syntax error, unexpected TASSIG, expecting TPCLOSE en '='

Objetivo extra 4: Arrays multidimensionales

● Restricciones semánticas añadidas

Restricciones estáticas:

- El número de dimensiones indicadas para acceder a un array debe ser igual que el número de dimensiones declaradas.
- Cuando se llama a una función con un array, el array indicado en la llamada y el indicado en los parámetros deben coincidir en tipo y tamaño.
- Comprobación de que el array al que se quiere acceder exista.
- Comprobación de que no se exceda el tamaño del array en el acceso (cuando se accede con constantes, por ejemplo A[1][2]).
- Comprobación de tipos al operar con los elementos del array.

Restricciones dinámicas:

- Si se accede al array con expresiones (por ejemplo, A[1+2][a*8-2]) se comprueba que no excedan los límites del array en tiempo de ejecución.
- Se calcula la dirección de acceso al array en tiempo de ejecución.

● Gramática

prog -> **programa IDENTIFICADOR**
declaraciones decl_de_subprogs
{ lista_de_sentencias **}**

declaraciones -> ε
| **variables** lista_de_ident **:** tipo **;** declaraciones
| **variables** lista_de_ident **: array** dimensiones_array **of** tipo **;** declaraciones

dimensiones_array -> **[ENTERO]**
| **[ENTERO]** dimensiones_array

array_acceso -> **[** expresion **]**
| **[** expresion **]** array_acceso

lista_de_ident -> **IDENTIFICADOR** resto_lista_id

resto_lista_id -> ε
| **,** **IDENTIFICADOR** resto_lista_id

tipo -> **entero**
| **real**

decl_de_subprogs -> ε
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
argumentos declaraciones
{ lista_de_sentencias **}**

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident : clase_par tipo resto_lis_de_param
| lista_de_ident **:** clase_par **array** dimensiones_array **of** tipo resto_lis_de_param

clase_par -> **in**
| **out**
| **in out**

resto_lis_de_param -> ε
| **;** lista_de_ident **:** clase_par tipo resto_lis_de_param
| **;** lista_de_ident **:** clase_par **array** dimensiones_array **of** tipo resto_lis_de_param

lista_de_sentencias -> ε
| sentencia lista_de_sentencias

M -> ε

sentencia -> variable **=** expresion **;**
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M
| **repetir** M **{** lista_de_sentencias **}** **hasta** expresion **;** M
| M **repetir siempre {** lista_de_sentencias **}** M
| **salir si** expresion **;** M
| **leer (** variable **) ;**
| **escribir (** expresion **) ;**
| **IDENTIFICADOR (** list_expr **) ;**

variable -> **IDENTIFICADOR**
| **IDENTIFICADOR** array_acceso

expresion -> **not** expresion
| expresion **or** M expresion
| expresion **and** M expresion
| expresion **==** expresion
| expresion **<** expresion
| expresion **>** expresion
| expresion **<=** expresion
| expresion **>=** expresion

| expresion **/=** expresion
| expresion **+** expresion
| expresion **-** expresion
| expresion ***** expresion
| expresion **/** expresion
| **IDENTIFICADOR**
| **IDENTIFICADOR** array_acceso
| **ENTERO**
| **REAL**
| **(** expresion **)**

list_expr -> expresion resto_lista_expr

resto_lista_expr -> ε
| **,** expresion resto_lista_expr

- ETDS

Los cambio respecto al ETDS anterior se marcan en **color verde:**
*** Por defecto imprimeError() imprimirá el error correspondiente y causará que no se escriba la traducción.**
*** Por defecto si hay alguna id de variable, procedimiento, … etc, se comprueba que exista y si no existe se hace un imprimeError().**
*** En calcular_dir_array() se realizan las comprobaciones de no estar fuera de rango (o se saca el código correspondiente para hacerlas) además de generar el código para el cálculo de la dirección.**
*** Por defecto si no se especifica:**
· Las expresiones tienen tipo *error* al crearse.
· El flag esConstante es *false* si no se asigna.
· El flag esVar solo es *true* si la expresión es un identificador.

programa -> programa **IDENTIFICADOR** {añadir_inst("prog"+IDENTIFICADOR.nombre);}
declaraciones decl_de_subprogs { lista_de_sentencias }
{añadir_inst("halt");}

declaraciones -> **ε**
| **var** lista_de_ident **:** tipo **;**
{añadir_declaraciones(tipo.clase,lista_de_ident.lnom);
tablaSimbolosActual.añadirVariable(lista_de_ident.lnom, tipo.clase);} declaraciones
| **var** lista_de_ident **: array** dimensiones_array **of** tipo **;**
{añadir_declaraciones_array(lista_de_ident.lnom, dimensiones_array, tipo.clase);
tablaSimbolosActual.añadirArray(lista_de_ident.lnom, dimensiones_array, tipo.clase);}
declaraciones

dimensiones_array -> **[ENTERO]** {dimensiones_array = inilista();
añadir_elem(dimensiones_array, entero.nom);}
| **[ENTERO]** dimensiones_array {dimensiones_array = dimensiones_array2;
añadir_elem(dimensiones_array, entero.nom);}

array_acceso -> **[** expresion **]** {array_acceso = inilista(); añadir_elem(array_acceso,
expresion);}
| **[** expresion **]** array_acceso {array_acceso = array_acceso2;
añadir_elem(array_acceso, expresion);}

lista_de_ident -> **IDENTIFICADOR** resto_lista_id {lista_de_ident.lnom:=lista_vacia();
lista_de_ident.lnom:=añadir(resto_lista_id.lnom,IDENTIFICADOR.nombre);}

resto_lista_id -> ε {resto_lista_id.lnom:=lista_vacia();}
| **,** **IDENTIFICADOR** resto_lista_id
{resto_lista_id.lnom:=añadir(resto_lista_id2.lnom,IDENTIFICADOR.nombre);}

tipo -> **entero** {tipo.clase="int";}
| **real** {tipo.clase="real";}

decl_de_subprogs -> ε
| decl_de_subprograma decl_de_subprogs

decl_de_subprograma -> **procedimiento IDENTIFICADOR**
{añadir_inst("proc"+IDENTIFICADOR.nombre);
tablaSimbolosActual.añadirProcedimiento(IDENTIFICADOR.nombre); tablaSimbolosAnt =
tablaSimbolosActual; tablaSimbolosActual = nueva tablaSimbolos();}
argumentos
declaraciones
{ lista_de_sentencias **}** {añadir_inst("endproc"); tablaSimbolosActual =
tablaSimbolosAnt;}

argumentos -> ε
| **(** lista_de_param **)**

lista_de_param -> lista_de_ident **:** clase_par tipo {añadir_parametros(clase_par.tipo,
tipo.clase, lista_de_ident.lnom); tablasSimbolos.añadirParametros(lista_de_ident.lnom,
clase_par.tipo, tipo.clase);}
resto_lis_de_param
| lista_de_ident **:** clase_par **array** dimensiones_array **of** tipo

{tablaSimbolosActual.añadirParametroArray(lista_de_ident.lnom, clase_par.tipo, dimensiones_array, tipo.clase);}
resto_lis_de_param
clase_par -> **in** {clase_par.tipo="val_";}
| **out** {clase_par.tipo="ref_";}
| **in out** {clase_par.tipo="ref_";}

resto_lis_de_param: ε
| **;** lista_de_ident : clase_par tipo { añadir_parametros(clase_par.tipo, tipo.clase, lista_de_ident.lnom); tablasSimbolos.añadirParametros(lista_de_ident.lnom, clase_par.tipo, tipo.clase);} resto_lis_de_param
| **;** lista_de_ident **:** clase_par **array** dimensiones_array **of** tipo
{tablaSimbolosActual.añadirParametroArray(lista_de_ident.lnom, clase_par.tipo, dimensiones_array, tipo.clase);}
resto_lis_de_param

lista_de_sentencias -> ε
{ lista_de_sentencias.exit = lista_vacia(); }
| sentencia lista_de_sentencias { lista_de_sentencias.exit = unir(sentencia.exit, lista_de_sentencias.exit); }

sentencia -> variable **=** expresion **;** { if (expresion.tipo == "bool") imprimeError();
if (variable.tipo != expresion.tipo) imprimeError();
añadir_ins(id.nombre + ":=" expresion.nombre); sentencia.exit = lista_vacia();}
| **si** expresion **entonces** M **{** lista_de_sentencias **}** M { completar_inst(expresion.true, M1.ref); completar_inst(expresion.false, M2.ref); sentencia.exit = lista_de_sentencias.exit; }
| **repetir** {contWhile++} M **{** lista_de_sentencias **}** **hasta** expresion **;** M {
if (expresion.tipo != "bool") imprimeError();
completar_inst(expresion.false, M1.ref); completar_inst(expresion.true, M2.ref);
completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia();contWhile--; }
| M **repetir siempre** {contWhile++} **{** lista_de_sentencias **}** {
añadir_inst("goto"+M.ref); completar(lista_de_sentencias.exit, M2.ref); sentencia.exit = lista_vacia();contWhile--; }
| **salir si** expresion M **;** {if (contWhile == 0) imprimeError(); if (expresion.tipo != "bool") imprimeError(); completar_inst(expresion.false, M.ref); sentencia.exit.true = expresion.true; }
| **leer (** variable **) ;** {añadir_inst("read" + variable.nombre); sentencia.exit = lista_vacia();}
| **escribir_linea (** expresion **) ;** { if (expresion.tipo == "bool") imprimeError();
añadir_inst("write" + expresion.nombre); añadir_inst("writeln"); sentencia.exit = lista_vacia();
}
| **IDENTIFICADOR (** list_expr **) ;**
{ params = tablaSimbolosActual.obtenerParams(IDENTIFICADOR.nombre;
if (params.numParams != list_expr.list_e.size()) imprimeError();
else
{

```
                for (elems_in(param) = param_actual and elems_in(list_expr.list_e) = expr)
                {
                        if (param_actual.clase = “ref” and !expr.esVar) imprimeError();
                        else if (param_actual.tipo != expr.tipo) imprimeError();
                        else
                        {
                                if (param_actual.tipo == “array”)
                                { if (!comprobar_dimensiones(param_actual.dimensiones,
        expr.dimensiones)) imprimeError(); }
                                añadir_inst(“param_”+param_actual.clase+” ”+expr.nombre);
                        }
                }
                añadir_inst(“call ”+IDENTIFICADOR.nombre);
        }
```

list_expr -> expresion resto_lista_expr {añadir_elemts(list_expr.list_e, resto_lista_expr.list_e); añadir_elemento(list_expr.list_e, expresion); }

resto_lista_expr -> ε { resto_lista_expr.list_e = inilistaexpr(); }
| , expresion resto_lista_expr
{añadir_elemts(resto_lista_expr.list_e, resto_lista_expr2.list_e);
añadir_elemento(resto_lista_expr.list_e, expresion); }

M -> ε { M.ref = obtenref(); }

variable -> **IDENTIFICADOR** {variable.nombre = id.nombre; variable.tipo = tablaSimbolosActual.buscarTipo(id.nombre);}
| **IDENTIFICADOR** array_acceso
{ datos_array = tablaSimbolosActual.obtenerDatosArray(IDENTIFICADOR.nombre);
calculodir = calcular_dir_array(datos_array, array_acceso);
variable.nombre = IDENTIFICADOR.nombre+”[”+calculodir+”]”;
variable.tipo = datos_array.tipoElemts;
}

expresion: expresion **==** expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
expresion.false:=inilista(obtenref()+1);
añadir_inst(“if” + expresion1.nombre + ”==” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion **<** expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }

```
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”<” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion > expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”>” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion <= expresion
{if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”<=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion >= expresion
{ if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”>=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion /= expresion
{ if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = “bool”;
expresion.true:= inilista(obtenref());
 expresion.false:=inilista(obtenref()+1);
 añadir_inst(“if” + expresion1.nombre + ”/=” + expresion2.nombre+ “goto”);
añadir_inst(goto);}
| expresion + expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”+”+expresion2.nombre); }
| expresion - expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+”:=”+expresion1.nombre+”-”+expresion2.nombre); }
| expresion * expresion { if (E1.tipo == “bool” or E2.tipo == “bool”) imprimeError();
```

if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo; expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"*"+expresion2.nombre); }
| expresion **/** expresion { if (E1.tipo == "bool" or E2.tipo == "bool") imprimeError();
if (E1.tipo != E2.tipo) { imprimeError(); }
else E.tipo = E1.tipo;
if (E2.esConstante) { if (valor(E2) == 0) imprimeError(); }
else {añadir_inst("if "+E2.nombre+" == 0 goto ERRORDIVNULL");}
expresion.nombre = nuevo_id();
añadir_inst(expresion.nombre+":="+expresion1.nombre+"/"+expresion2.nombre); }
| expresion **or** M expresion { if (E1.tipo != "bool" or E2.tipo != "bool") imprimeError();
completar (E1.false, M.ref);
E.true = unir(E1.true, E2.true);
E.false = E2.false;}
| expresion **and** M expresion { if (E1.tipo != "bool" or E2.tipo != "bool") imprimeError();
completar(E1.true,M.ref);
E1.true=E2.true;
E1.false = unir(E1.false,E2.false);}
| **not** expresion { if (E.tipo != "bool") imprimeError();
E.true= E1.false;
E.false = E1.true;}
| **IDENTIFICADOR** { expresion.nombre = id.nombre; expresion.tipo =
tablaSimbolosActual.buscarTipo(id.nombre); expresion.esConstante = false; }
| **IDENTIFICADOR** array_acceso
{ datos_array = tablaSimbolosActual.obtenerDatosArray(IDENTIFICADOR.nombre);
calculodir = calcular_dir_array(datos_array, array_acceso);
variable.nombre = IDENTIFICADOR.nombre+"["+calculodir+"]";
variable.tipo = datos_array.tipoElemts;
}
| **ENTERO** { expresion.nombre = ent.nombre; expresion.tipo = "int";
expresion.esConstante = true; }
| **REAL** { expresion.nombre = real.nombre; expresion.tipo = "real";
expresion.esConstante = true; }
| **(** expresion **)** { expresion = expresion1; }

- Casos de prueba

Programa	Salida
Pruebas que se espera que se compilen correctamente	
Llamadas a procedimientos con Arrays. 1. programa ejemplo	1: prog ejemplo; 2: int a;

2. variables a,b,c : entero;
3. variables e,d : real;
4. variables array1: array[5][5] of entero;
5. variables array2: array[1][2] of real;
6.
7. (* esto es un comentario *)
8. procedimiento prueba (a: in array[5][5] of entero; b: out array[1][2] of real; resul: in out entero)
9. variables z,aux:entero;
10. variables x: array[5] of real;
11. {
12. a[1][1]=z+aux;
13. x[3] = x[1];
14. }
15.
16.
17.
18.
19. {
20.
21. prueba(array1,array2,b);
22. prueba(array1,array2,array1[0][0]);
23.
24. }
	3: int b;
	4: int c;
	5: real e;
	6: real d;
	7: array_int array1,25;
	8: array_real array2,2;
	9: proc prueba;
	10: val_array_int a,25;
	11: ref_array_real b,2;
	12: ref_int resul;
	13: int z;
	14: int aux;
	15: array_real x,5;
	16: _dir1:=0;
	17: _dir2:=_dir1+1;
	18: _dir3:=_dir2*5;
	19: _dir4:=_dir3+1;
	20: _t1:=z+aux;
	21: a[_dir4]:=_t1;
	22: _dir5:=0;
	23: _dir6:=_dir5+3;
	24: _dir7:=0;
	25: _dir8:=_dir7+1;
	26: x[_dir6]:=x[_dir8];
	27: endproc;
	28: param_val array1;
	29: param_ref array2;
	30: param_ref b;
	31: call prueba;
	32: _dir9:=0;
	33: _dir10:=_dir9+0;
	34: _dir11:=_dir10*5;
	35: _dir12:=_dir11+0;
	36: param_val array1;
	37: param_ref array2;
	38: param_ref array1[_dir12];
	39: call prueba;
	40: halt;

Pruebas variadas de uso de arrays programa ejemplo variables a,b,c : entero; variables e,d : real; variables array1: array[5][5] of entero; variables array2: array[1][2] of real; {	1: prog ejemplo; 2: int a; 3: int b; 4: int c; 5: real e; 6: real d; 7: array_int array1,25; 8: array_real array2,2; 9: _dir1:=0; 10: _dir2:=_dir1+1;

```
array1[1][1]=3;
array2[0][0]=1.0;
a = array1[1][2]+3;

array1[a][b] = 0;
array1[a*b-1*43+1][a/c]  = array1[a][b-1] +
array1[a][b]/2;


si array2[a][b] > 2.0 entonces
{
array1[2][2]=3;
}

}
```

```
11: _dir3:=_dir2*5;
12: _dir4:=_dir3+1;
13: array1[_dir4]:=3;
14: _dir5:=0;
15: _dir6:=_dir5+0;
16: _dir7:=_dir6*2;
17: _dir8:=_dir7+0;
18: array2[_dir8]:=1.0;
19: _dir9:=0;
20: _dir10:=_dir9+1;
21: _dir11:=_dir10*5;
22: _dir12:=_dir11+2;
23: _t1:=array1[_dir12]+3;
24: a:=_t1;
25: _dir13:=0;
26: if 5 <= a goto
ERROR_OUT_OF_BOUNDS;
27: _dir14:=_dir13+a;
28: if 5 <= b goto
ERROR_OUT_OF_BOUNDS;
29: _dir15:=_dir14*5;
30: _dir16:=_dir15+b;
31: array1[_dir16]:=0;
32: _t2:=a*b;
33: _t3:=1*43;
34: _t4:=_t2-_t3;
35: _t5:=_t4+1;
36: _t6:=a/c;
37: if c == 0 goto ERRORDIVNULL;
38: _dir17:=0;
39: if 5 <= _t5 goto
ERROR_OUT_OF_BOUNDS;
40: _dir18:=_dir17+_t5;
41: if 5 <= _t6 goto
ERROR_OUT_OF_BOUNDS;
42: _dir19:=_dir18*5;
43: _dir20:=_dir19+_t6;
44: _t7:=b-1;
45: _dir21:=0;
46: if 5 <= a goto
ERROR_OUT_OF_BOUNDS;
47: _dir22:=_dir21+a;
48: if 5 <= _t7 goto
ERROR_OUT_OF_BOUNDS;
49: _dir23:=_dir22*5;
50: _dir24:=_dir23+_t7;
51: _dir25:=0;
52: if 5 <= a goto
ERROR_OUT_OF_BOUNDS;
53: _dir26:=_dir25+a;
```

	54: if 5 <= b goto ERROR_OUT_OF_BOUNDS; 55: _dir27:=_dir26*5; 56: _dir28:=_dir27+b; 57: _t8:=array1[_dir28]/2; 58: _t9:=array1[_dir24]+_t8; 59: array1[_dir20]:=_t9; 60: _dir29:=0; 61: if 1 <= a goto ERROR_OUT_OF_BOUNDS; 62: _dir30:=_dir29+a; 63: if 2 <= b goto ERROR_OUT_OF_BOUNDS; 64: _dir31:=_dir30*2; 65: _dir32:=_dir31+b; 66: if array2[_dir32]>2.0 goto 68; 67: goto 73; 68: _dir33:=0; 69: _dir34:=_dir33+2; 70: _dir35:=_dir34*5; 71: _dir36:=_dir35+2; 72: array1[_dir36]:=3; 73: halt;

Pruebas variadas de uso de arrays 2
1. programa ejemplo
2. variables a,b,c : array[2][2] of entero;
3. variables d,e : array[5][11][23][242] of real;
4. (* esto es un comentario *)
5. procedimiento sumar (x,y: in array[2][2] of entero; resul: in out entero)
6. variables aux:entero;
7. {
8. aux=x[1][1];
9. resul=y[1][0];
10. repetir {
11. aux = aux - 1;
12. resul = resul+1;
13. } hasta aux == 0 ;
14. }
15. {
16. leer(a[0][0]); leer(e[1][10][20][200]);
17. d[1][1][1][1]= 1.0/e[1][2][3][4];
18. e[2][2][2][100]= 1.0/1.3;
19. sumar(a,b,c[0][0]); (* los que hagan llamadas a procedimientos *)
	1: prog ejemplo;
	2: array_int c,4;
	3: array_int b,4;
	4: array_int a,4;
	5: array_real e,306130;
	6: array_real d,306130;
	7: proc sumar;
	8: val_array_int y,4;
	9: val_array_int x,4;
	10: ref_int resul;
	11: int aux;
	12: _dir1:=0;
	13: _dir2:=_dir1+1;
	14: _dir3:=_dir2*2;
	15: _dir4:=_dir3+1;
	16: aux:=x[_dir4];
	17: _dir5:=0;
	18: _dir6:=_dir5+1;
	19: _dir7:=_dir6*2;
	20: _dir8:=_dir7+0;
	21: resul:=y[_dir8];
	22: _t1:=aux-1;
	23: aux:=_t1;
	24: _t2:=resul+1;
	25: resul:=_t2;

20. 21. si a[0][0]/1 + 4 * 2 > b[1][1] entonces 22. { 23. leer(a[0][0]); 24. } 25. }	26: if aux==0 goto 28; 27: goto 22; 28: endproc; 29: _dir9:=0; 30: _dir10:=_dir9+0; 31: _dir11:=_dir10*2; 32: _dir12:=_dir11+0; 33: read a[_dir12]; 34: _dir13:=0; 35: _dir14:=_dir13+1; 36: _dir15:=_dir14*11; 37: _dir16:=_dir15+10; 38: _dir17:=_dir16*23; 39: _dir18:=_dir17+20; 40: _dir19:=_dir18*242; 41: _dir20:=_dir19+200; 42: read e[_dir20]; 43: _dir21:=0; 44: _dir22:=_dir21+1; 45: _dir23:=_dir22*11; 46: _dir24:=_dir23+1; 47: _dir25:=_dir24*23; 48: _dir26:=_dir25+1; 49: _dir27:=_dir26*242; 50: _dir28:=_dir27+1; 51: _dir29:=0; 52: _dir30:=_dir29+1; 53: _dir31:=_dir30*11; 54: _dir32:=_dir31+2; 55: _dir33:=_dir32*23; 56: _dir34:=_dir33+3; 57: _dir35:=_dir34*242; 58: _dir36:=_dir35+4; 59: _t3:=1.0/e[_dir36]; 60: if e[_dir36] == 0 goto ERRORDIVNULL; 61: d[_dir28]:=_t3; 62: _dir37:=0; 63: _dir38:=_dir37+2; 64: _dir39:=_dir38*11; 65: _dir40:=_dir39+2; 66: _dir41:=_dir40*23; 67: _dir42:=_dir41+2; 68: _dir43:=_dir42*242; 69: _dir44:=_dir43+100; 70: _t4:=1.0/1.3; 71: e[_dir44]:=_t4; 72: _dir45:=0; 73: _dir46:=_dir45+0; 74: _dir47:=_dir46*2; 75: _dir48:=_dir47+0;

	76: param_val a; 77: param_val b; 78: param_ref c[_dir48]; 79: call sumar; 80: _dir49:=0; 81: _dir50:=_dir49+0; 82: _dir51:=_dir50*2; 83: _dir52:=_dir51+0; 84: _t5:=a[_dir52]/1; 85: _t6:=4*2; 86: _t7:=_t5+_t6; 87: _dir53:=0; 88: _dir54:=_dir53+1; 89: _dir55:=_dir54*2; 90: _dir56:=_dir55+1; 91: if _t7>b[_dir56] goto 93; 92: goto 98; 93: _dir57:=0; 94: _dir58:=_dir57+0; 95: _dir59:=_dir58*2; 96: _dir60:=_dir59+0; 97: read a[_dir60]; 98: halt;
Pruebas que se espera que den error de compilación	
Errores variados al tratar arrays: 1. programa ejemplo 2. variables a,b,c : entero; 3. variables e,d : real; 4. variables array1: array[5][5] of entero; 5. variables array2: array[1][2] of real; 6. 7. 8. { 9. 10. array1[1][1]=3.0; 11. array2[0][0]=1; 12. a = array1[1][2]+3; 13. 14. array1[a][b] = array2[a][b]; 15. 16. array1[1]=2; 17. array2[2][a][b][c] = a+2; 18. array2[2.0+1.5][e*2.0] = a+2; 19. }	Error en línea 10: Se ha intentado asignar un real a la variable array1[_dir4] de tipo int. Error en línea 11: Se ha intentado asignar un int a la variable array2[_dir8] de tipo real. Error en línea 14: Se ha intentado asignar un real a la variable array1[_dir16] de tipo int. Error en línea 16: Las dimensiones del array array1 no coinciden con lo que se intenta acceder. Error en línea 16: Se ha intentado asignar un int a la variable de tipo null. Error en línea 17: Las dimensiones del array array2 no coinciden con lo que se intenta acceder. Error en línea 17: Se ha intentado asignar un int a la variable de tipo null. Error en línea 18: Se ha recibido una expresión no entera para la dirección a acceder del array. Error en línea 18: Se ha recibido una expresión no entera para la dirección a acceder del array. Error en línea 18: Se ha intentado asignar

	un int a la variable array2[_dir24] de tipo error.

Mejoras/especificaciones/aspectos a resaltar

- Se podría implementar los casteos de tipos de manera dinámica (nosotros hemos escogido que nuestro lenguaje sea fuertemente tipado y por lo tanto no se realizan).
- Cuando hay ids repetidos en declaraciones de variables o subprogramas se podría indicar en qué línea habían sido declaradas.

Comentarios/valoración de la práctica:

Nos ha parecido una práctica interesante, que nos ha ayudado a comprender mejor el funcionamiento de un compilador.
La práctica ayuda a aprender los conceptos teóricos de clase de manera práctica y es un buen apoyo a la hora de repasar estos mismos.