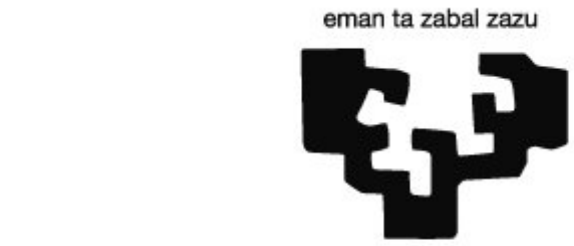
informatika fakultatea — facultad de informática

eman ta zabal zazu — Universidad del País Vasco — Euskal Herriko Unibertsitatea

PRÁCTICA DE COMPILACIÓN



Universidad del País Vasco — Euskal Herriko Unibertsitatea

GRUPO1

OBJETIVOS AÑADIDOS DE IKER PINTADO

Iker Pintado **(ipintado003@ikasle.ehu.eus)**
Iker Hidalgo **(ihidalgo013@ikasle.ehu.eus)**
Álvaro Rodrigo **(arodrigo027@ikasle.ehu.eus)**

grupo1

objetivos: Iker Pintado

ÍNDICE

1. Introducción

En esta práctica se ha construido el *front-end* de un compilador utilizando la técnica de construcción de traductores ascendente, a partir de un Esquema de Traducción Dirigido por la Sintaxis (ETDS). El lenguaje fuente del compilador es uno de alto nivel, y el de salida, un código de tres direcciones. Para la implementación, se ha utilizado el lenguaje de programación C++, junto con las herramientas Flex y Bison.

En el caso particular de esta práctica se ha implementado la parte básica (Analizador léxico y sintáctico, desarrollo del ETDS básico e Implementación del traductor) y ciertos objetivos extra (Expresiones booleanas, sentencia if-elseif-else, sentencia for, sentencia switch-case, llamadas a procedimientos y análisis semántico).

Para evitar la excesiva expansión de esta documentación, en el proceso de traducción, todo lo relacionado con los objetivos extra tendrá asignado un color diferente. Cada objetivo tendrá un color propio. El mapeo es el siguiente:

- Expresiones booleanas: Morado
- Sentencia if-elseif-else: Caqui
- Sentencia for: Verde oscuro
- Sentencia switch-case: Azul oscuro
- Llamadas a procedimientos: Rosa
- Análisis semántico: Rojo

2. Autoevaluación

Teniendo en cuenta los criterios de evaluación establecidos por el enunciado de la práctica, a esta en concreto le correspondría la siguiente puntuación:

- **Mínimo:** Implementación del analizador léxico y sintáctico y desarrollo del ETDS básico. 4 puntos.
- **Aprobado:** Implementación del traductor. 1 punto.
- **Expresiones booleanas:** Especificación de las expresiones booleanas, diseño e implementación de su traducción. 1 punto.
- **Nueva estructura de control 1:** Especificación de un if-else con opción a indefinidos elseif en medio, diseño de la traducción e implementación. 1 punto.
- **Nueva estructura de control 2:** Especificación de un bucle for, diseño de la traducción e implementación. 1 punto.
- **Nueva estructura de control 3:** Especificación de un switch-case, diseño de la traducción e implementación. 1 punto.
- **Llamadas a procedimientos:** Uso correcto de parámetros reales. 1 punto.
- **Análisis semántico:** Diseño e implementación de restricciones semánticas (incluidas las de los objetivos extra). 2 puntos.

Si todos los objetivos abordados son evaluados con la máxima puntuación posible establecida para los mismos, a la práctica le correspondería una puntuación total de 12 puntos.

3. Descripción del esfuerzo

TAREAS		Iker Pintado	Iker Hidalgo	Álvaro Rodrigo
Analizador léxico-sintáctico		8h	7h	6h
ETDS BÁSICO & IMPLEMENTACIÓN	**Tabla de atributos**	—	—	1:15h
	ETDS	2h	2h	2h
	Abstracciones funcionales	—	1:15h	—
	Pruebas	1:30h	—	—
OBJETIVOS EXTRA	**Expresiones booleanas**	1h	—	—
	Sentencia if-elseif-else	1:15h	—	—
	Sentencia for	2h	—	—
	Sentencia switch-case	3:30h	—	—
	Llamada a procedimientos	1h	—	—
	Análisis semántico	5h	—	—
DOCUMENTACIÓN		9h	—	—

4. Análisis léxico

4.1. Especificación de tokens

Nombre del token	Descripción	Patrón (LEX)	Lexemas
RPROGRAM	Palabra reservada "program"	**program**	—
RBEGIN	Palabra reservada "begin"	**begin**	—
RENDPROGRAM	Palabra reservada "fin"	**fin**	—
RDEF	Palabra reservada "def"	**def**	—
RMAIN	Palabra reservada "main"	**main**	—
RLET	Palabra reservada "let"	**let**	—
RIN	Palabra reservada "in"	**in**	—
RIF	Palabra reservada "if"	**if**	—
RELSE	Palabra reservada "else"	**else**	—
RELSEIF	Palabra reservada "elseif"	**elseif**	—

RWHILE	Palabra reservada “while”	**while**	—
RBREAK	Palabra reservada “break”	**break**	—
RFOREVER	Palabra reservada “forever”	**forever**	—
RFOR	Palabra reservada “for”	**for**	—
RSWITCH	Palabra reservada “switch”	**switch**	—
RCASE	Palabra reservada “case”	**case**	—
RDEFAULT	Palabra reservada “default”	**default**	—
RCONTINUE	Palabra reservada “continue”	**continue**	—
RPRINTLN	Palabra reservada “println”	**println**	—
RREAD	Palabra reservada “read”	**read**	—
RINT	Palabra reservada “integer”	**integer**	—
RFLOAT	Palabra reservada “float”	**float**	—
TPTPT	Dos puntos seguidos	**..**	—

TABRIRLLAVE	Carácter de apertura de llave	**{**	—
TCERRARLLAVE	Carácter de cierre de llave	**}**	—
TABRIRPAREN	Carácter de apertura de paréntesis	**(**	—
TCERRARPAREN	Carácter de cierre de paréntesis	**)**	—
TEQ	Carácter de igualdad	**=**	—
TDOSPT	Carácter de dos puntos	**:**	—
TSEMIC	Carácter punto y coma	**;**	—
TAMP	Carácter “ampersand”	**&**	—
TCOMA	Carácter coma	**,**	—
OOR	Operador booleano or	**or**	—
OAND	Operador booleano and	**and**	—
ONOT	Operador booleano not	**not**	—
OMULT	Operador de multiplicación	*****	—

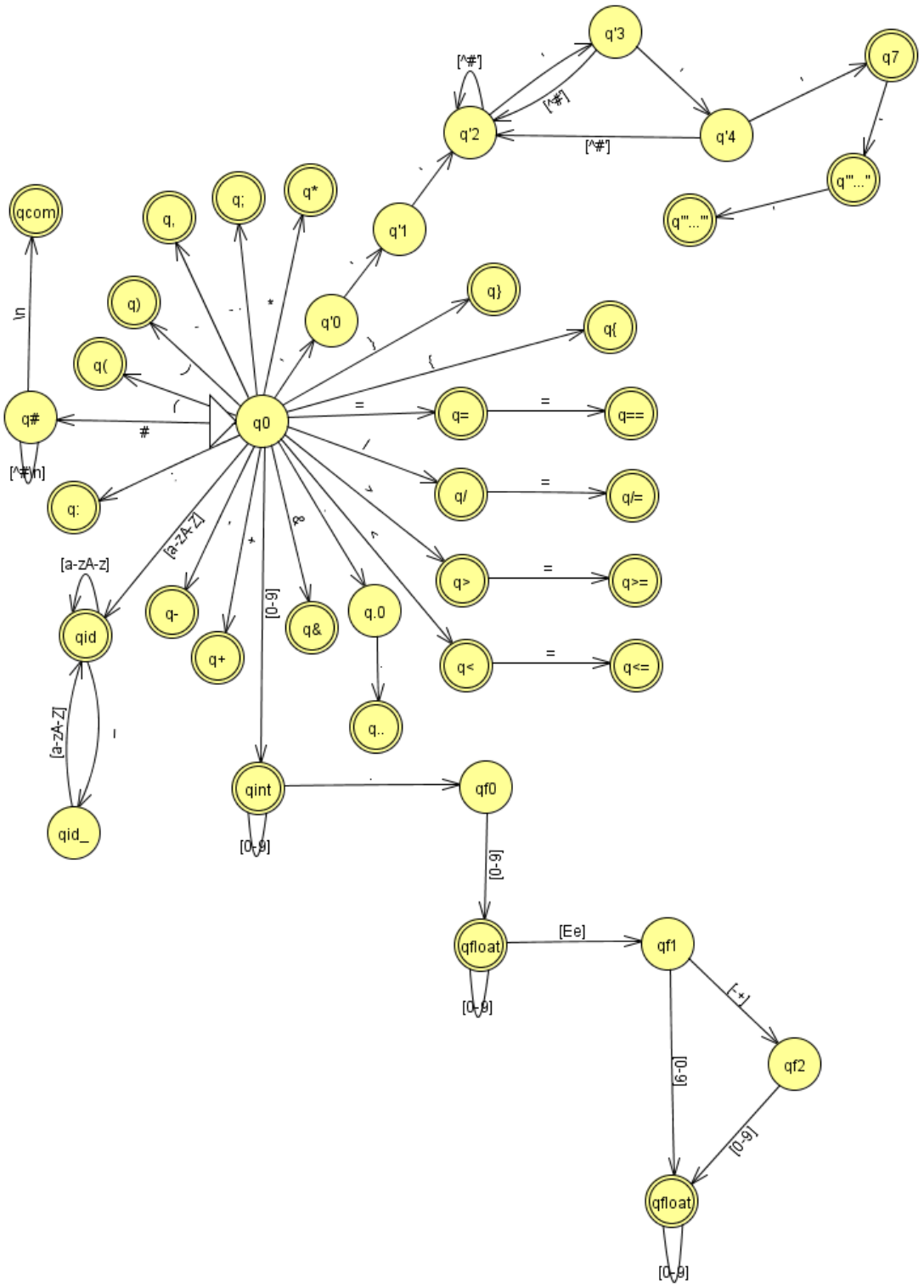
OSUM	Operador de suma	+	—	
OREST	Operador de resta	-	—	
ODIV	Operador de división	/	—	
CEQ	Comparador de igualdad	==	—	
CLT	Comparador de inferioridad	<	—	
CGT	Comparador de superioridad	>	—	
CLE	Comparador inferior o igual	<=	—	
CGE	Comparador superior o igual	>=	—	
CNE	Comparador de desigualdad	/=	—	
TIDENTIFIER	Token identificador de variables	**[a-zA-Z](_?[a-zA-Z0-9])***	id var_a_4 a21e7	_id 23id var__
TDOUBLE	Token de tipo doble	**[0-9]+\.[0-9]+([eE][\-\+]?[0-9]+)?**	2.07 102.3e-12 27.00E7	2.s12 12.34E+-4 -321.0
TINTEGER	Token de tipo integer	**[0-9]+**	432 3	23a2 -2.3

grupo1

objetivos: Iker Pintado

COMENTARIO COMÚN	Comentario de una sola línea, empezado por #	**#[^\n#]*\n**	#hola # #cualquiera	## #ho#la #algo#
COMENTARIO MULTILÍNEA	Comentario de varias líneas, comienza y termina por '''. No permite #	**'''('?'?[^'#])*'?'?'''**	'''eee"oeee""" """""" '''salto de linea"""	'''o'''o''' '''#''' '''no cierra y salta de linea"

4.2. Autómata único



5. Proceso de traducción

5.1. Gramática

M → ξ

N → ξ

programa → **def main ( ) :** bloque_ppl

bloque_ppl → decl_bl {
decl_de_subprogs
lista_de_sentencias
}

bloque → {
lista_de_sentencias
}

decl_bl → **let** declaraciones **in**

| ξ

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

| lista_de_ident **:** tipo

lista_de_ident → **id** resto_lista_id

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

| ξ

tipo → **integer**

| **float**

decl_de_subprogs → decl_de_subprograma decl_de_subprogs

| ξ

decl_de_subprograma →**def id** argumentos : bloque_ppl

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

| ξ

lista_de_param → lista_de_ident : clase_par tipo resto_lis_de_param

clase_par → ξ

| **&**

resto_lis_de_param → ; lista_de_ident : clase_par tipo resto_lis_de_param

| ξ

lista_de_sentencias → sentencia lista_de_sentencias

| ξ

sentencia → variable = expresion **;**

| **if** expresion **:** M bloque M

| **forever :** M bloque M

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

| **break if** expresion M**;**

| **continue;**

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

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

| **if** expresion **:** M bloque N M parte_else M

| **for** variable **in** expresion **..** expresion **..** expresion **:** M bloque M

| M **break;**

| **switch** expresion **: {** cases M **}**

| **id (** lista_expresiones **) ;**

lista_expresiones → expresion resto_lista_expr

| ξ

resto_lista_expr → **,** expresion resto_lista_expr

| ξ

cte → **num_entero**
| **num_real**

cases → **case** cte **:** M lista_de_sentencias M **;** r_cases

| **default :** lista_de_sentencias **;**

r_cases → **default :** M lista_de_sentencias **;**

|**case** cte **:** M lista_de_sentencias M **;** r_cases

| ξ

parte_else → **elseif** expresion **:** M bloque N M parte_else M

| **else :** bloque

| **elseif** expresion **:** M bloque M

variable → **id**

expresion → expresion == expresion

| expresion > expresion

| expresion < expresion

| expresion >= expresion

| expresion <= expresion

| expresion /= expresion

| expresion **or** M expresion

| expresion **and** M expresion

| **not** expression

| expresion + expresion

| expresion - expresion

| expresion * expresion

| expresion / expresion

| variable

| **num_entero**

| **num_real**

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

5.2. Definición de atributos

(L: Léxico, S: Sintetizado)

Símbolo	Nombre	Tipo	L/S	Descripción
id	nom	string	L	Contiene la cadena de caracteres del id.
expresion	nom	string	S	Contiene la cadena de caracteres de la expresión.
	true	lista de enteros	S	Contiene las referencias a los goto's a rellenar si la expresión es cierta, en caso de que la expresión no sea booleana es una lista vacía.
	false	lista de enteros	S	Contiene las referencias a los gotos a rellenar si la expresión es falsa, en caso de que la expresión no sea booleana es una lista vacía.
	tipo	string	S	Indica el tipo que adopta la expresión: entero, real, booleano o error.
	esVar	booleano	S	Indica si la expresión referencia directamente a una variable.
bloque	exit	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un exit.
	conti	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un continue.

lista_sentencias	exit	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un exit.
	conti	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un continue.
sentencia	exit	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un exit.
	conti	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un continue.
M	ref	num entero	S	Contiene la referencia a una instrucción de código. Se utiliza para completar los goto.
lista_de_ident	inom	lista de strings	S	Contiene la lista de los identificadores.
resto_lista_id	inom	lista de strings	S	Contiene la lista de los identificadores.
tipo	clase	string	S	Cadena de carácteres que indica el tipo de numero (int o float).
clase_par	tipo	string	S	Cadena de caracteres que indica el tipo de la clase par (val o ref).
num_entero	nom	num entero	L	Contiene el valor numérico (entero).
num_real	nom	num real	L	Contiene el valor numérico (real).
variable	nom	string	S	Cadena de caracteres con el nombre de variable .
N	next	lista de enteros	S	Lista que contiene la referencia al goto que genera este no terminal.
parte_else	exit	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un exit.
	conti	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un continue.
cte	nom	num entero o num real	S	Contiene el valor constante de un entero o un real.
	tipo	string	S	Indica el tipo que adopta la constante: entero o real.

cases	gotos	lista de enteros	S	Lista que contiene las referencias a dónde están los goto's condicionales que se deben rellenar de la estructura switch-case.
	exit	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un exit.
	conti	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un continue.
	tipes	lista de strings	S	Indica el tipo de la constante de cada cláusula case de un switch-case.
r_cases	gotos	lista de enteros	S	Lista que contiene las referencias a dónde están los goto's condicionales que se deben rellenar de la estructura switch-case.
	exit	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un exit.
	ini	num entero	S	Referencia la línea en la que las sentencias de la cláusula comienzan.
	conti	lista de enteros	S	Contiene las referencias de los goto's a completar para terminar la sentencia y simular un continue.
	tipes	lista de strings	S	Indica el tipo de la constante de cada cláusula case de un switch-case.
lista_expresiones	exprs	lista de strings	S	Lista que contiene el nombre de todas las expresiones de la lista.
	vars	lista de booleanos	S	Lista que contiene los booleanos que indican, para cada expresión de la lista, si referencia directamente a una variable o no.
	tipes	lista de strings	S	Lista que contiene el tipo de cada expresión.

resto_lista_expr	exprs	lista de strings	S	Lista que contiene el nombre de todas las expresiones de la lista.
	vars	lista de booleanos	S	Lista que contiene los booleanos que indican, para cada expresión de la lista, si referencia directamente a una variable o no.
	tipes	lista de strings	S	Lista que contiene el tipo de cada expresión.

5.3. Abstracciones funcionales

obtenref: código → ref_código

Descripción: Dado una estructura de código numerado devuelve la referencia a la instrucción inmediatamente siguiente.

Tipo de los argumentos:

código: Estructura numerada de Strings

añadir_inst: código x inst → código

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

Tipo de los argumentos:

código: Estructura numerada de Strings
inst: String de la instrucción

añadir_declaraciones: código x listaids x tipo → código

Descripción: Dados una estructura de código numerada, una lista de identificadores, y un tipo de dato, escribe las declaraciones en la estructura de código.

Tipo de los argumentos:

código: Estructura numerada de Strings
listaids: Lista de Strings que representa los distintos ids
tipo: String que indica si es un entero o real

añadir: lista x elemento → lista

Descripción: Dados una lista de elementos y un nuevo elemento, añade al inicio de la lista lista el elemento dado.

Tipo de los argumentos:

lista: Lista de elementos
elemento: Cualquiera (string o int)

añadir_params: código x listaids x clase x tipo → código

Descripción: Dados una estructura de código numerada, una lista de identificadores, una clase, y un tipo de dato, escribe las declaraciones de los parámetros en la estructura de código.

Tipo de los argumentos:

código: Estructura numerada de Strings
listaids: Lista de Strings que representa los distintos ids
clase: val/ref
tipo: String que indica si es un entero o real

unir: lista x lista → lista

Descripción: Dadas dos listas de elemento devuelve una lista que contiene todos los elementos de las dos listas iniciales.

Tipo de los argumentos:

lista: Lista de elementos

completar: código x lista_refs x ref_código → código

Descripción: Dados una lista de referencias del código y una referencia del código, completa las instrucciones goto que contienen las referencias de la lista con la referencia dada.

Tipo de los argumentos:

código: Estructura numerada de Strings
lista_refs: Lista de referencias del código que indican los goto's a completar
ref_código: Referencia de código con la que se completarán los goto's

lista_vacia: → lista

Descripción: Inicializa una lista vacía

inilista: elemento → lista

Descripción: Dado un elemento inicializa una lista que contiene dicho elemento

Tipo de los argumentos:

elemento: Elemento de cualquier tipo aceptado

objetivos: Iker Pintado

nuevo_id: código → string

Descripción: Genera un nuevo identificador para una expresión

Tipo de los argumentos:
código: Estructura numerada de Strings

escribir: código →

Descripción: Escribe el código generado en pantalla

Tipo de argumentos:
código: Estructura numerada de Strings

completar_switch: código x lista_refs x string → código

Descripción: Dada una lista de referencia a gotos condicionales incompletos, completa la condición con el string dado y el goto saltará exáctamente dos líneas

Tipo de argumentos:
código: Estructura numerada de Strings
lista_refs: Lista de referencias al código que indican a donde está
cada goto condicional que se rellenará
string: Valor de la expresión

añadir_error: error_id →

Descripción: Dado un identificador de error, lo registra para que se lance al final de la compilación. Dependiendo del identificador, puede llegar a gestionar más o menos cosas relacionadas con el error. Por ejemplo, los tipos.

Tipo de argumentos:
error_id: Identificador entero

imprimir_errores: lista_de_errores →

Descripción: Dado un registro de errores semánticos, los escribe por pantalla

Tipo de argumentos:
lista_de_errores: Lista que lleva el registro de identificadores de error

objetivos: Iker Pintado

obtenerrores: → num_errores

Descripción: Devuelve el número de errores encontrados

setProcActual: string →

Descripción: Establece el procedimiento que se está procesando en el momento

Tipo de argumentos:
string: Identificador del procedimiento

getProcActual: → string

Descripción: Devuelve el nombre del procedimiento que se está procesando en el momento

vacía: lista → bool

Descripción: Indica si una lista está vacía o no

Tipo de argumentos:
lista: Una lista cualquiera de elementos

empilar: pila_tabla_símbolos x tabla_símbolos → pila_tabla_símbolos

Descripción: Empila la nueva tabla de símbolos en la pila

Tipo de argumentos:
pila_tabla_símbolos: Pila de tablas de símbolos
tabla_símbolos: Tabla de símbolos

desempilar: pila_tabla_símbolos → pila_tabla_símbolos

Descripción: Desempila la tabla de símbolos del tope

Tipo de argumentos:
pila_tabla_símbolos: Pila de tablas de símbolos

objetivos: Iker Pintado

obtenerTipo: pila_tabla_símbolos x string → string

Descripción: Devuelve el tipo de la variable cuyo identificador es pasado como parámetro

Tipo de argumentos:
pila_tabla_símbolos: Pila de tablas de símbolos
string: identificador

añadirParametro: pila_tabla_símbolos x string x string x string x string →

Descripción: Dado un procedimiento y toda la información de una variable, registra esta variable como parámetro del procedimiento dado

Tipo de argumentos:
pila_tabla_símbolos: Pila de tablas de símbolos
string: Identificador del procedimiento
string: Identificador de la variable
string: Paridad de la clase
string: Tipo de la variable

TablaSimbolos: → tabla_símbolos

Descripción: Crea una tabla de símbolos nueva

añadirVariable: tabla_símbolos x string x string →

Descripción: Añade una variable a la tabla

Tipo de argumentos:
tabla_símbolos: Tabla de símbolos
string: Identificador de la variable
string: Tipo de la clase

añadirProc: tabla_símbolos x string →

Descripción: Añade un procedimiento a la tabla

Tipo de argumentos:
tabla_símbolos: Tabla de símbolos
string: Identificador del procedimiento

objetivos: Iker Pintado

existeId: tabla_símbolos x string → bool

Descripción: Dado un id, dice si está en la tabla de símbolos o no

Tipo de argumentos:
tabla_símbolos: Tabla de símbolos
string: Identificador

existeIdPila: pila_tabla_símbolos x string → bool

Descripción: Dado un id, dice si está en la pila de tablas de símbolos o no

Tipo de argumentos:
pila_tabla_símbolos: Pila de tablas de símbolos
string: Identificador

borrarId: tabla_símbolos x string → bool

Descripción: Dado un id, lo elimina de la tabla

Tipo de argumentos:
tabla_símbolos: Tabla de símbolos
string: Identificador

obtenerTiposParam: pila_tabla_símbolos x string x int→ string x string

Descripción: Dado un identificador de procedimiento e índice de parámetro, devuelve el tipo y la paridad correspondiente al mismo

Tipo de argumentos:
pila_tabla_símbolos: Pila de tablas de símbolos
string: Identificador del procedimiento
int: Índice del parámetro

numArgsProc: pila_tabla_símbolos x string → int

Descripción: Dado un identificador de procedimiento, devuelve el número de parámetros del mismo

Tipo de argumentos:
pila_tabla_símbolos: Pila de tablas de símbolos
string: Identificador del procedimiento

length: Lista → int

Descripción: Dada una lista, devuelve su longitud

Tipo de argumentos:
lista: Una lista cualquiera

5.4. ETDS

M → ξ { M.ref :=obtenref(); } (1)

N → ξ { N.next :=inilista(obtenref()); añadir_inst(goto); } (2)

```
programa → def main ( ) :    { añadir_inst(goto || 11);
            añadir_inst(write || error, el salto de un bucle for debe ser distinto de 0);
            añadir_inst(writeln);
            añadir_inst(goto);
            añadir_inst(write || error, el bucle tipo for es infinito);
            añadir_inst(writeln);
            añadir_inst(goto);
            añadir_inst(write || error, división entre 0. Resultado indefinido);
            añadir_inst(writeln);
            añadir_inst(goto);
            añadir_inst( proc main ) ;
            empilar(TablaSimbolos());} (3)
            bloque_ppl          { desempilar();
              if (obtenerrores()>0) imprimir_errores();
              else{
              tmp:= inilista(4);
              tmp := añadir(tmp, 7);
              tmp := añadir(tmp, 10);
              completar(tmp, obtenref());
              añadir_inst( halt );
              escribir();}} (4)

bloque_ppl → decl_bl {
                decl_de_subprogs
                lista_de_sentencias
                } {if (not vacía(lista_de_sentencias.exit)) añadir_error(2);
                  if (not vacía(lista_de_sentencias.conti)) añadir_error(3);} (5)

bloque → {
                lista_de_sentencias
            } {bloque.exit := lista_de_sentencias.exit;
                bloque.conti := lista_de_sentencias.conti;} (6)
```

objetivos: Iker Pintado

decl_bl → **let** declaraciones **in**

| ξ

declaraciones → declaraciones **;** lista_de_ident **:** tipo
{ foreach id in lista_de_ident.Inom{
if (not existeId(id) añadirVariable(id, tipo.clase);
else añadir_error(1);
}
añadir_declaraciones(lista_de_ident.lnom, tipo.clase)} (7)

| lista_de_ident **:** tipo
{foreach id in lista_de_ident.Inom{
if (not existeId(id) añadirVariable(id, tipo.clase);
else añadir_error(1);
}
añadir_declaraciones(lista_de_ident.lnom, tipo.clase)} (8)

lista_de_ident → **id** resto_lista_id
{lista_de_ident.Inom := añadir(resto_lista_id.lnom,id.nom);} (9)

resto_lista_id → **,** **id** resto_lista_id
{resto_lista_id.Inom := añadir(resto_lista_id1.lnom,id.nom);} (10)

| ξ {resto_lista_id := lista_vacía(); } (11)

tipo → **integer** {tipo.clase := "int";} (12)

| **float** {tipo.clase := "real";} (13)

decl_de_subprogs → decl_de_subprograma decl_de_subprogs

| ξ

decl_de_subprograma →**def id** { setProcActual(id.nom);
if (not existeId(id.nom) añadirProc(id.nom);
else añadir_error(1);
empilar(TablaSimbolos());
añadir_inst(proc || id.nom);} (14)
argumentos : bloque_ppl {
desempilar();
añadir_inst(endproc || id.nom);} (15)

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

| ξ

lista_de_param → lista_de_ident : clase_par tipo
{ foreach id in lista_de_ident.Inom {
if (not existeId(id)) añadirParam(getProcActual(), id, clase_par.tipo, tipo.clase);
else añadir_error(1);
}
añadir_params(lista_de_ident.Inom, clase_par.tipo, tipo.clase); } (16)
resto_lis_de_param

clase_par → **ε** {clase_par.tipo := "val";} (17)

| **&** {clase_par.tipo := "ref";} (18)

resto_lis_de_param → ; lista_de_ident : clase_par tipo
{ foreach id in lista_de_ident.Inom {
if (not existeId(id)) añadirParam(procActual(), id, clase_par.tipo, tipo.clase);
else añadir_error(1);
}
añadir_params(lista_de_ident.Inom, clase_par.tipo, tipo.clase); } (19)
resto_lis_de_param

| ξ

lista_de_sentencias → sentencia lista_de_sentencias
{ lista_de_sentencias.exit := unir(sentencia.exit,lista_de_sentencias.exit) ;
lista_de_sentencias.conti := unir(sentencia.conti,lista_de_sentencias.conti); } (20)

| ξ
{lista_de_sentencias.exit := lista_vacía() ;
lista_de_sentencias.conti := lista_vacía();} (21)

sentencia → variable = expresion **;**{if (not existeIdPila(variable.nom)) añadir_error(4);
else if (obtenerTipo(variable.nom) != expresion.tipo) añadir_error(5);
else añadir_inst(variable.nom || := || expresion.nom);
sentencia.exit := lista_vacía() ;
sentencia.conti := lista_vacía(); } (22)

objetivos: Iker Pintado

| **if** expresion **:** M bloque M
{if (expresion.tipo != “bool”) añadir_error(6);
completar(expresion.true,M1.ref);
completar(expresion.false,M2.ref);
sentencia.exit := bloque.exit ;
sentencia.conti := bloque.conti;} (23)

| **forever :** M bloque M
{añadir_inst(goto || M1.ref);
completar(bloque.exit, M2.ref + 1);
sentencia.exit := lista_vacía() ;
sentencia.conti := bloque.conti;} (24)

| **while** M expresion **:** M bloque M
{if (expresion.tipo != “bool”) añadir_error(6);
añadir_inst(goto || M1.ref);} (25)
else: bloque M
{completar(expresion.true, M2.ref);
completar(expresion.false, M3.ref + 1);
completar(bloque1.exit, M3.ref +1);
completar(bloque1.conti, M1.ref);
completar(bloque2.exit, M4.ref);
completar(bloque2.conti, M1.ref);
sentencia.exit := lista_vacia() ;
sentencia.conti := lista_vacia() ;} (26)

| **break if** expresion M**;** { if (expresion.tipo != “bool”) añadir_error(6);
completar(expresion.false, M1.ref);
sentencia.exit := expresion.true;
sentencia.conti := lista_vacía();} (27)

| **continue;**{sentencia.conti := inilista(obtenref());
sentencia.exit := lista_vacia();
añadir_inst(goto); } (28)

| **read (** variable **) ;** { if (not existeIdPila(variable.nom)) añadir_error(4);
else añadir_inst(read || variable.nom);
sentencia.exit := lista_vacía() ;
sentencia.cont := lista_vacía();} (29)

| **println (** expresion **) ;** { if (expresion.tipo == “bool”) añadir_error(8);
else {añadir_inst(write || expresion.nom) ; añadir_inst(writeln) ;}
sentencia.exit := lista_vacía() ;
sentencia.cont := lista_vacía();} (30)

```
| if expresion : M bloque N M parte_else M {
  if (expresion.tipo != "bool") añadir_error(6);
  sentencia.exit := unir(bloque.exit, parte_else.exit);
  sentencia.conti := unir(bloque.conti, parte_else.conti);
  completar(N.next, M3.ref);
  completar(expresion.false, M2.ref);
  completar(expresion.true, M1.ref);
} (31)

| for variable in expresion .. expresion .. expresion : {
if (existeId(variable.nom)) añadir_error(1);
else {
  añadirVariable(variable.nom, "int");
  if (expresion1.tipo != "int" or expresion2.tipo != "int" or expresion3.tipo !=
     "int") añadir_error(7);
}
  añadir_inst(if || expresion3.nom || = || 0 || goto || 2);
  añadir_inst(if || expresion3.nom || > || 0 || goto || obtenref() + 2);
  añadir_inst(goto ||  obtenref() + 3);
  añadir_inst(if || expresion1.nom || > || expresion2.nom || goto || 5);
  añadir_inst(goto ||  obtenref() + 2);
  añadir_inst(if || expresion1.nom || < || expresion2.nom || goto || 5);
  añadir_inst(int || variable.nom);
  añadir_inst(variable.nom || := || expresion1.nom);
  añadir_inst(if || expresion3.nom || > || 0 || goto || obtenref() + 2);
  añadir_inst(goto || obtenref() + 5);
  añadir_inst(if || variable.nom || >= ||expresion1.nom|| goto || obtenref()+2);
  añadir_inst(goto)
  añadir_inst(if || variable.nom || <= ||expresion2.nom|| goto || obtenref()+6);
  añadir_inst(goto)
  añadir_inst(if || variable.nom || >= ||expresion2.nom|| goto || obtenref()+2);
  añadir_inst(goto)
  añadir_inst(if || variable.nom || <= ||expresion1.nom|| goto || obtenref()+2);
  añadir_inst(goto)
} (32)
M bloque M{borrarId(variable.nom);
añadir_inst(variable.nom || := || variable.nom || + || expresion3.nom);
completar(bloque.conti, M2.ref);
completar(bloque.exit, M2.ref + 2);
añadir_inst(goto || M1.ref - 10);
tmp  := inilista(M1.ref -1);
tmp := añadir(tmp, M1.ref - 3);
tmp := añadir(tmp, M1.ref - 5);
tmp := añadir(tmp, M1.ref - 7);
completar(tmp, M2.ref + 2);
sentencia.exit := lista_vacia();
sentencia.conti := lista_vacia();} (33)
```

```
| M break; {sentencia.exit := inilista(obtenref());
        añadir_inst(goto);
        sentencia.conti := lista_vacia();} (34)

| switch expresion : { cases M } {
  if (expresion.tipo == "bool") añadir_error(8);
else
  foreach tipo in cases.tipes{
        if (tipo != expresion.tipo) añadir_error(9);
  }
  completar(cases.exit, M2.ref);
  sentencia.conti := cases.conti;
  sentencia.exit := lista_vacía();
  completar_switch(cases.gotos, expresion.nom);
} (35)

| id ( lista_expresiones ) ; {
if(not existeIdPila(id.nom)) añadir_error(4);
else if (numArgsProc(id.nom) != length(lista_expresiones.exprs)) añadir_error(10)
else {
  for i in 1..numArgsProc(id.nom){
        parametro = obtenerTiposParam(id.nom, i);
        if (lista_expresiones.tipes[i] != parametro.tipo) añadir_error(5);
        else if (parametro.clase_par == "ref" and not lista_expresiones.vars[i])
              añadir_error(11);
        else
              añadir_inst(param || _ || parametro.clase_par || _ ||
                    lista_expresiones.exprs[i]);
  }
  añadir_inst(call || id.nom);
}
sentencia.exit := lista_vacía();
sentencia.conti := lista_vacía();
} (36)

lista_expresiones → expresion resto_lista_expr {
        lista_expresiones.vars := unir(inilista(expresion.esVar), resto_lista_expr.vars);
        lista_expresiones.exprs := unir(inilista(expresion.nom), resto_lista_expr.exprs);
        lista_expresiones.tipes := unir(inilista(expresion.tipo), resto_lista_expr.tipes);
        } (37)

        | ξ {
        lista_expresiones.vars := lista_vacía();
        lista_expresiones.exprs := lista_vacía();
        lista_expresiones.tipes :=  lista_vacía();
        } (38)
```

resto_lista_expr → , expresion resto_lista_expr {
resto_lista_expr.vars := unir(inilista(expresion.esVar), resto_lista_expr1.vars);
resto_lista_expr.exprs := unir(inilista(expresion.nom), resto_lista_expr1.exprs);
resto_lista_expr.tipes := unir(inilista(expresion.tipo), resto_lista_expr1.tipes);
} (39)

| ξ {
resto_lista_expr.vars := lista_vacía();
resto_lista_expr.exprs := lista_vacía();
resto_lista_expr.tipes := lista_vacía();
} (40)

cte → **num_entero** {cte.tipo := "int"; cte.nom := num_entero.nom;} (41)

| **num_real** {cte.tipo := "real"; cte.nom := num_real.nom;} (42)

cases → **case** cte **:** M {
añadir_inst(if || cte.nom || =);
añadir_inst(goto);
} (43) lista_de_sentencias {
añadir_inst(goto);
} (44) M **;** r_cases { cases.tipes := unir(inilista(cte.tipo), r_cases.tipes);
cases.gotos = unir(r_cases.gotos, inilista(M1.ref));
completar(inilista(M1.ref +1), M2.ref);
cases.exit = unir(lista_de_sentencias.exit, r_cases.exit);
completar(inilista(M2.ref - 1), r_cases.ini);
cases.conti = unir(lista_de_sentencias.conti, r_cases.conti);
} (45)

| **default :** lista_de_sentencias **;** { cases.tipes := lista_vacía();
cases.gotos := lista_vacía();
cases.exit := lista_de_sentencias.exit;
cases.conti := lista_de_sentencias.conti;
} (46)

objetivos: Iker Pintado

```
r_cases → default : M  lista_de_sentencias ; {
            r_cases.tipes := lista_vacía();
            r_cases.gotos := lista_vacía();
            r_cases.ini := M.ref;
            r_cases.exit := lista_de_sentencias.exit;
            r_cases.conti := lista_de_sentencias.conti;
          } (47)

          |case cte : M {
          añadir_inst(if || cte.nom || =);
          añadir_inst(goto);
          } (48) lista_de_sentencias {
            añadir_inst(goto);
          } (49) M ; r_cases { r_cases.tipes := unir(inilista(cte.tipo), r_cases.tipes);
          r_cases.gotos = unir(r_cases.gotos, inilista(M1.ref));
          completar(inilista(M1.ref +1), M2.ref);
          r_cases.exit = unir(lista_de_sentencias.exit, r_cases.exit);
          completar(inilista(M2.ref - 1), r_cases.ini);
          r_cases.conti = unir(lista_de_sentencias.conti, r_cases.conti);
          r_cases.ini := M1.ref + 2;
          } (50)

          | ξ  { r_cases.tipes := lista_vacía();
            r_cases.gotos := lista_vacía();
            r_cases.ini := obtenref();
            r_cases.exit := lista_vacía();
            r_cases.conti := lista_vacía();
          } (51)


parte_else → elseif expresion : M bloque N M parte_else M{
            if (expresion.tipo != "bool") añadir_error(6);
            parte_else.exit := unir(bloque.exit, parte_else1.exit);
            parte_else.conti := unir(bloque.conti, parte_else1.conti);
            completar(N.next, M3.ref);
            completar(expresion.false, M2.ref);
            completar(expresion.true, M1.ref);
          } (52)

          | else : bloque{
            parte_else.exit := bloque.exit;
            parte_else.conti := bloque.conti;
          } (53)
```

| **elseif** expresion **:** M bloque M{ if (expresion.tipo != “bool”) añadir_error(6);
parte_else.exit := bloque.exit;
parte_else.conti := bloque.conti;
completar(expresion.false, M2.ref);
completar(expresion.true, M1.ref);} (54)

variable → **id** {variable.nom := id.nom} (55)

expresion → expresion == expresion{ expresion.tipo:= “bool”;
if (expresion1.tipo == “bool” or expresion2.tipo == “bool”) {añadir_error(8);
expresion.true := lista_vacía();
expresion.false := lista_vacía();
}else{
expresion.true := inilista(obtenref());
expresion.false := inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || = || expresion2.nom || goto);
añadir_inst(goto);
}
expresion.esVar := False;
expresion.nom := “”; } (56)

| expresion > expresion{ expresion.tipo:= “bool”;
if (expresion1.tipo == “bool” or expresion2.tipo == “bool”) {añadir_error(8);
expresion.true := lista_vacía();
expresion.false := lista_vacía();
}else{
expresion.true := inilista(obtenref());
expresion.false := inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || = || expresion2.nom || goto);
añadir_inst(goto);
}
expresion.esVar := False;
expresion.nom := “”; } (57)

| expresion < expresion{ expresion.tipo:= “bool”;
if (expresion1.tipo == “bool” or expresion2.tipo == “bool”) {añadir_error(8);
expresion.true := lista_vacía();
expresion.false := lista_vacía();
}else{
expresion.true := inilista(obtenref());
expresion.false := inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || = || expresion2.nom || goto);
añadir_inst(goto);
}
expresion.esVar := False;

expresion.nom := “”; } (58)

| expresion >= expresion{ expresion.tipo:= “bool”;
if (expresion1.tipo == “bool” or expresion2.tipo == “bool”) {añadir_error(8);
expresion.true := lista_vacía();
expresion.false := lista_vacía();
}else{
expresion.true := inilista(obtenref());
expresion.false := inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || = || expresion2.nom || goto);
añadir_inst(goto);
}
expresion.esVar := False;
expresion.nom := “”; } (59)

| expresion <= expresion{ expresion.tipo:= “bool”;
if (expresion1.tipo == “bool” or expresion2.tipo == “bool”) {añadir_error(8);
expresion.true := lista_vacía();
expresion.false := lista_vacía();
}else{
expresion.true := inilista(obtenref());
expresion.false := inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || = || expresion2.nom || goto);
añadir_inst(goto);
}
expresion.esVar := False;
expresion.nom := “”; } (60)

| expresion /= expresion { expresion.tipo:= “bool”;
if (expresion1.tipo == “bool” or expresion2.tipo == “bool”) {añadir_error(8);
expresion.true := lista_vacía();
expresion.false := lista_vacía();
}else{
expresion.true := inilista(obtenref());
expresion.false := inilista(obtenref()+1);
añadir_inst(if || expresion1.nom || = || expresion2.nom || goto);
añadir_inst(goto);
}
expresion.esVar := False;
expresion.nom := “”; } (61)

```
| expresion or M expresion {
if (expresion1.tipo != "bool" or expresion2.tipo != "bool") añadir_error(6);
else expresion.tipo:= "bool";
expresion.esVar := False;
expresion.nom := "";
completar( expresion1.false, M.ref);
expresion.false := expresion2.false;
expresion.true := unir(expresion1.true, expresion2.true);}
```

62

```
| expresion and M expresion {
if (expresion1.tipo != "bool" or expresion2.tipo != "bool") añadir_error(6);
else expresion.tipo:= "bool";
expresion.esVar := False;
expresion.nom := "";
completar( expresion1.true, M.ref);
expresion.true := expresion2.true;
expresion.false := unir(expresion1.false, expresion2.false);}
```

63

```
| not expression {
if (expresion.tipo != "bool") añadir_error(6);
else expresion.tipo:= "bool";
expresion.esVar := False;
expresion.nom := "";
expresion.true := expresion1.false;
expresion.false := expresion1.true;}
```

64

```
  | expresion + expresion   {
if(expresion1.tipo == "error" or expresion2.tipo == "error")
        expresion.tipo := "error";
else if (expresion1.tipo != "bool" or expresion2.tipo != "bool")
        añadir_error(8);
else if (expresion1.tipo != expresion2.tipo)
          expresion.tipo := "real";
else  expresion.tipo := expresion1.tipo";
expresion.esVar := False;
expresion.nom := nuevo_id();
añadir_inst(expresion.nom || := || expresion1.nom || + || expresion2.nom);
expresion.true := lista_vacía();
expresion.false := lista_vacía(); }
```

65

objetivos: Iker Pintado

```
| expresion - expresion    {
if(expresion1.tipo == “error” or expresion2.tipo == “error”)
        expresion.tipo := “error”;
else if (expresion1.tipo != “bool” or expresion2.tipo != “bool”)
        añadir_error(8);
else if (expresion1.tipo != expresion2.tipo)
         expresion.tipo := “real”;
else  expresion.tipo := expresion1.tipo”;
expresion.esVar := False;
expresion.nom := nuevo_id();
añadir_inst(expresion.nom || := || expresion1.nom || - || expresion2.nom);
expresion.true := lista_vacía();
expresion.false := lista_vacía(); }  (66)

  | expresion * expresion    {
if(expresion1.tipo == “error” or expresion2.tipo == “error”)
        expresion.tipo := “error”;
else if (expresion1.tipo != “bool” or expresion2.tipo != “bool”)
        añadir_error(8);
else if (expresion1.tipo != expresion2.tipo)
         expresion.tipo := “real”;
else  expresion.tipo := expresion1.tipo”;
expresion.esVar := False;
expresion.nom := nuevo_id();
añadir_inst(expresion.nom || := || expresion1.nom || * || expresion2.nom);
expresion.true := lista_vacía();
expresion.false := lista_vacía(); }  (67)


  | expresion / expresion    {
if(expresion1.tipo == “error” or expresion2.tipo == “error”)
        expresion.tipo := “error”;
else if (expresion1.tipo != “bool” or expresion2.tipo != “bool”)
        añadir_error(8);
else if (expresion1.tipo != expresion2.tipo)
         expresion.tipo := “real”;
else  expresion.tipo := expresion1.tipo”;
añadir_inst(if || expresion2.nom || = || 0 || goto || 8);
expresion.esVar := False;
expresion.nom := nuevo_id();
añadir_inst(expresion.nom || := || expresion1.nom || / || expresion2.nom);
expresion.true := lista_vacía();
expresion.false := lista_vacía(); }  (68)
```

| variable { if (not existeIdPila(variable.nom)) añadir_error(4);
else expresion.tipo := obtenerTipo(variable.nom);
expresion.esVar := True;
expresion.nom := variable.nom;
expresion.true := lista_vacía();
expresion.false := lista_vacía();} (69)

| **num_entero** {expresion.tipo := "int";
expresion.esVar := False;
expresion.nom := num_entero.nom;
expresion.true := lista_vacía();
expresion.false := lista_vacía();} (70)

| **num_real** {expresion.tipo := "real";
expresion.esVar := False;
expresion.nom := num_real.nom;
expresion.true := lista_vacía();
expresion.false := lista_vacía();} (71)

| **(** expresion **)** {expresion.tipo :=expresion1.tipo;
expresion.esVar := False;
expresion.nom := expresion1.nom;
expresion.true := expresion1.true;
expresion.false := expresion1.false;} (72)

6. Casos de prueba

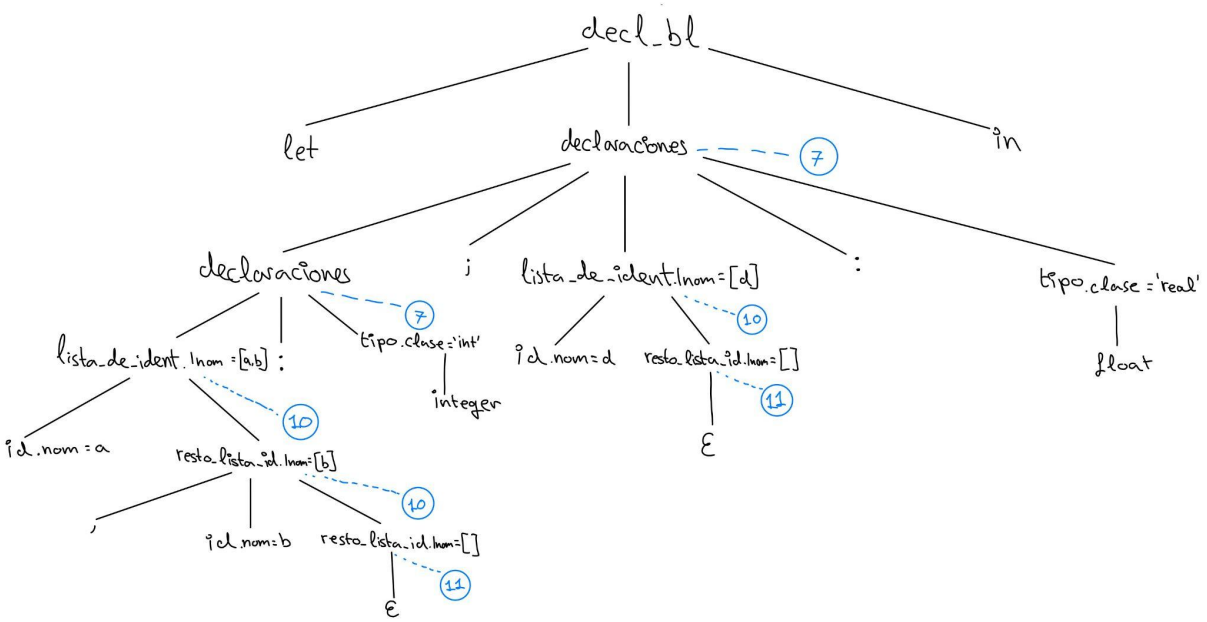
Todos los casos de prueba tendrán una imagen del árbol decorado de mejor calidad al final del documento. Para ser exactos, en la sección de anexos. También, en el árbol decorado de cada caso de prueba, siempre que salten instrucciones, aparecerá una referencia al bloque de código que salta. Estas referencias se pueden revisar en el ETDS. Tienen forma circular y contienen un número identificador dentro, siempre se encuentran al finalizar el bloque de código.

6.1. Prueba 1: Declaraciones

6.1.1. Programa

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

6.1.2. Árbol decorado



6.1.3. Código intermedio generado

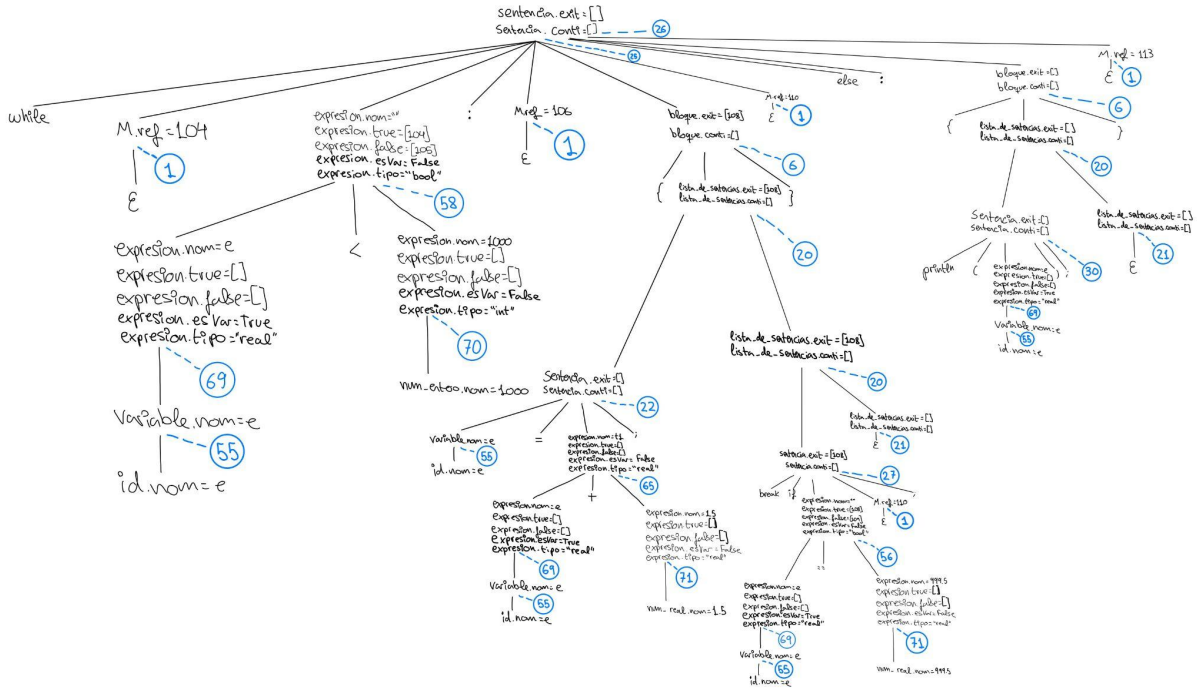
1. .
2. int a;
3. int b;
4. real d;
5. .

6.2. Prueba 2: While-else & break-if

6.2.1. Programa

```
.
.
.
while e<1000 :{
  e = e + 1.5;
  break if e == 999.5;
} else : {
  println(e);
}
.
.
.
```

6.2.2. Árbol decorado



6.2.3. Código intermedio generado

103. .
104. if e<1000 goto 106;
105. goto 111;

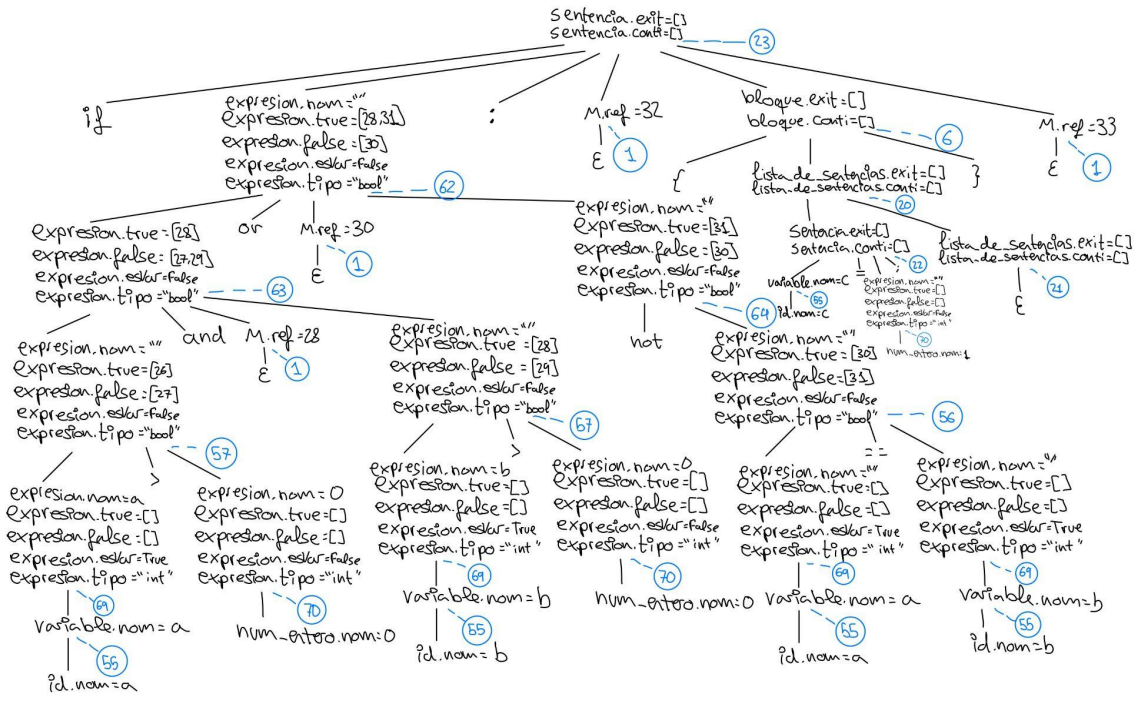
```
106. _t1 := e + 1.5;
107. e := _t1;
108. if e=999.5 goto 111;
109. goto 110;
110. goto 104;
111. write e;
112. writeln;
113. .
```

6.3. Prueba 3: Booleanas

6.3.1. Programa

```
.
.
.
if a>0 and b>0 or not a==b:{
  c = 1;
}
.
.
.
```

6.3.2. Árbol decorado



6.3.3. Código intermedio generado

25. .
26. if a>0 goto 28;
27. goto 30;
28. if b>0 goto 32;
29. goto 30;
30. if a=b goto 33;

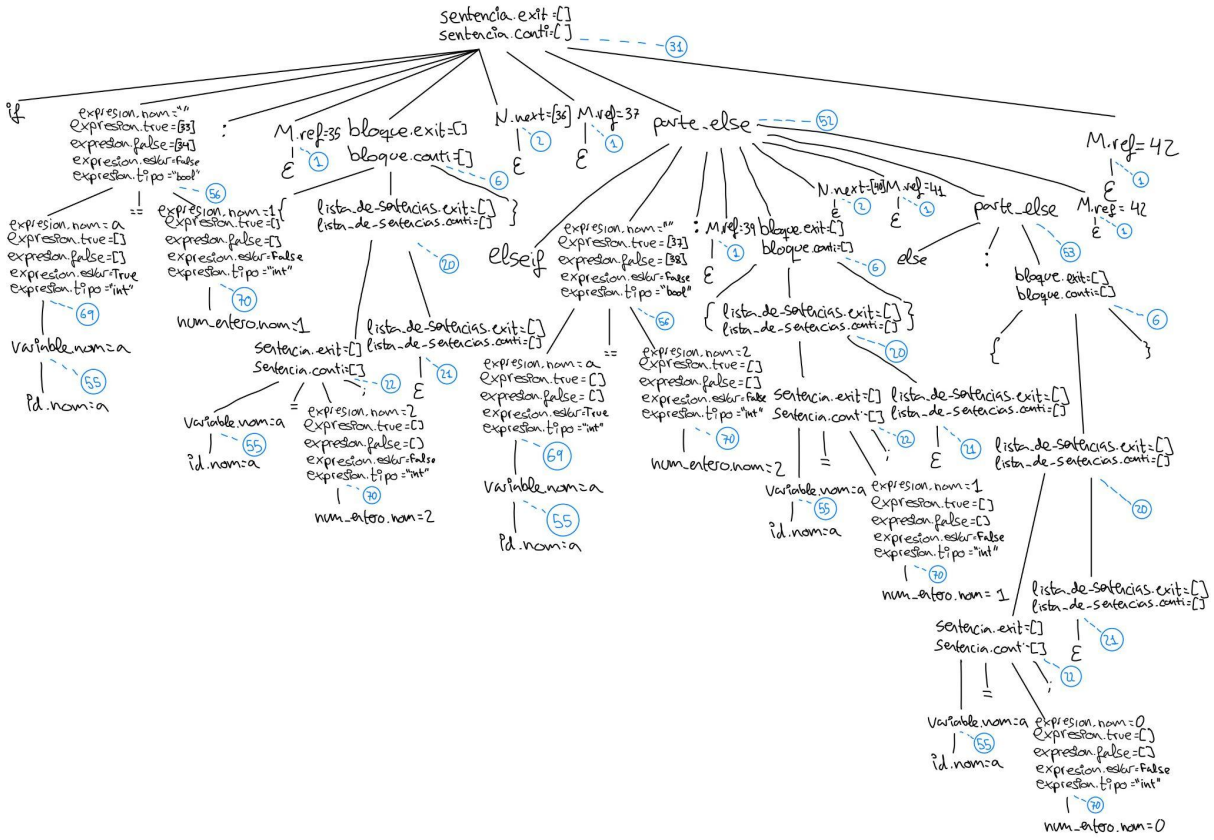
31. goto 32;
32. c := 1;
33. .

6.4. Prueba 4: If-elseif-else

6.4.1. Programa

```
.
.
.
if a==1:{
    a = 2;
} elseif a==2: {
    a = 1;
} else: {
    a = 0;
}
.
.
.
```

6.4.2. Árbol decorado



6.4.3. Código intermedio generado

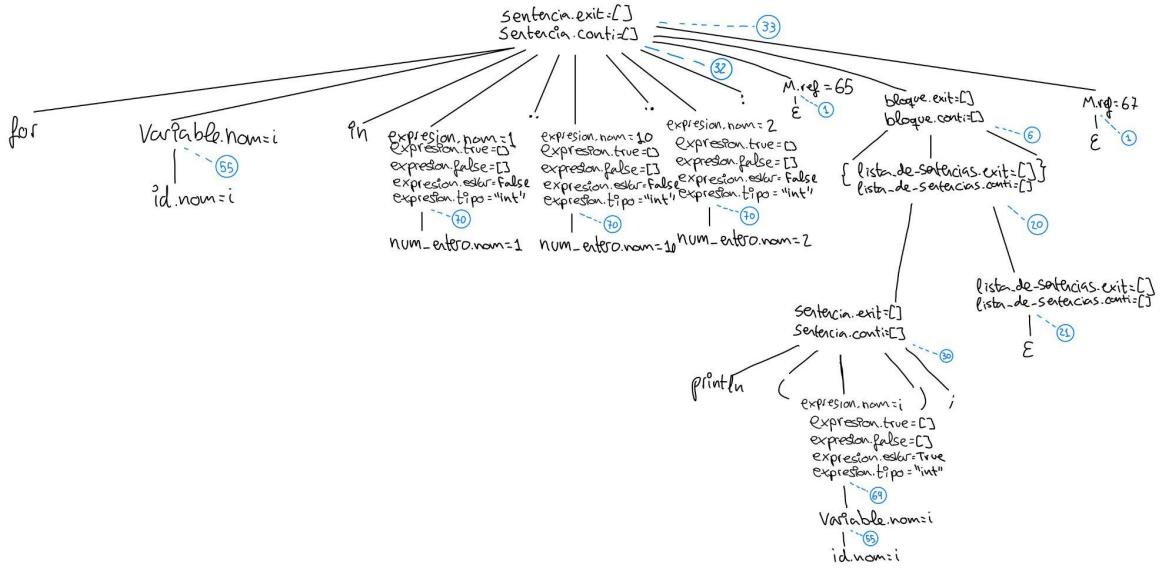
```
32. .
33. if a=1 goto 35;
34. goto 37;
35. a := 2;
36. goto 42;
37. if a=2 goto 39;
38. goto 41;
39. a := 1;
40. goto 42;
41. a := 0;
42. .
```

6.5. Prueba 5: For

6.5.1. Programa

```
.
.
.
for i in 1..10..2 :{
   println(i);
}
.
.
.
```

6.5.2. Árbol decorado



6.5.3. Código intermedio generado

46. .
47. if 2=0 goto 2;
48. if 2>0 goto 50;
49. goto 52;
50. if 1>10 goto 5;
51. goto 53;
52. if 1<10 goto 5;
53. int i;
54. i := 1;

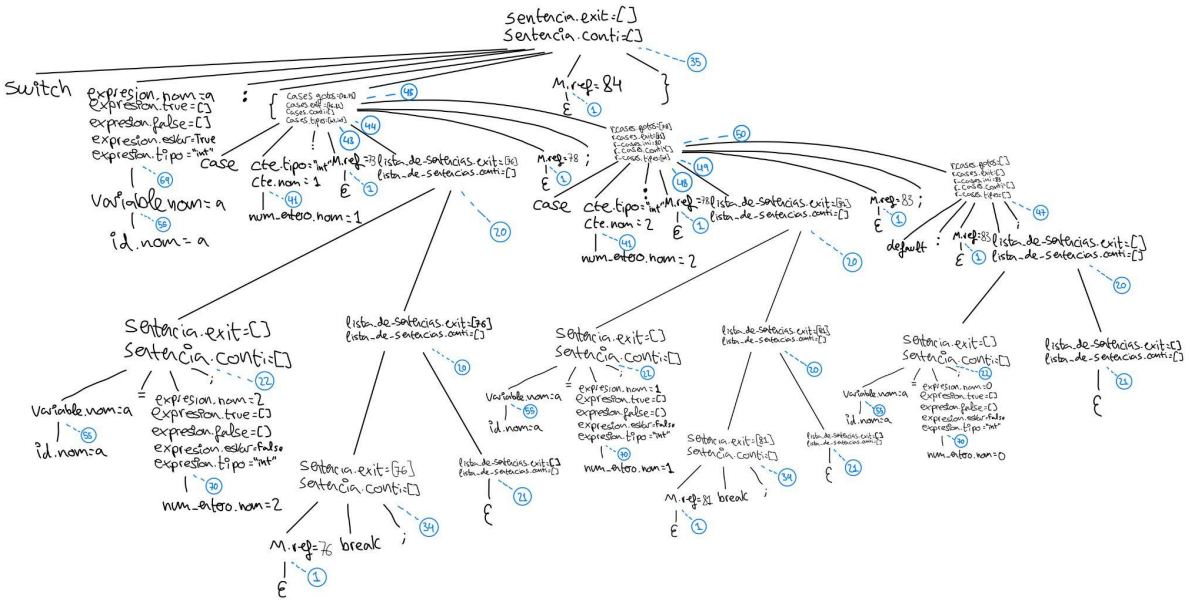
```
55. if 2>0 goto 57;
56. goto 69;
57. if i >= 1 goto 59;
58. goto 69;
59. if i <= 10 goto 65;
60. goto 69;
61. if i >= 10 goto 63;
62. goto 69;
63. if i <= 1 goto 65;
64. goto 69;
65. write i;
66. writeln;
67. i := i + 2;
68. goto 55;
69. .
```

6.6. Prueba 6: Switch-case

6.6.1. Programa

```
.
.
.
switch a:{
    case 1:
      a = 2;
      break;
    ;
    case 2:
      a = 1;
      break;
    ;
    default:
      a = 0;
    ;
}
.
.
.
```

6.6.2. Árbol decorado



6.6.3. Código intermedio generado

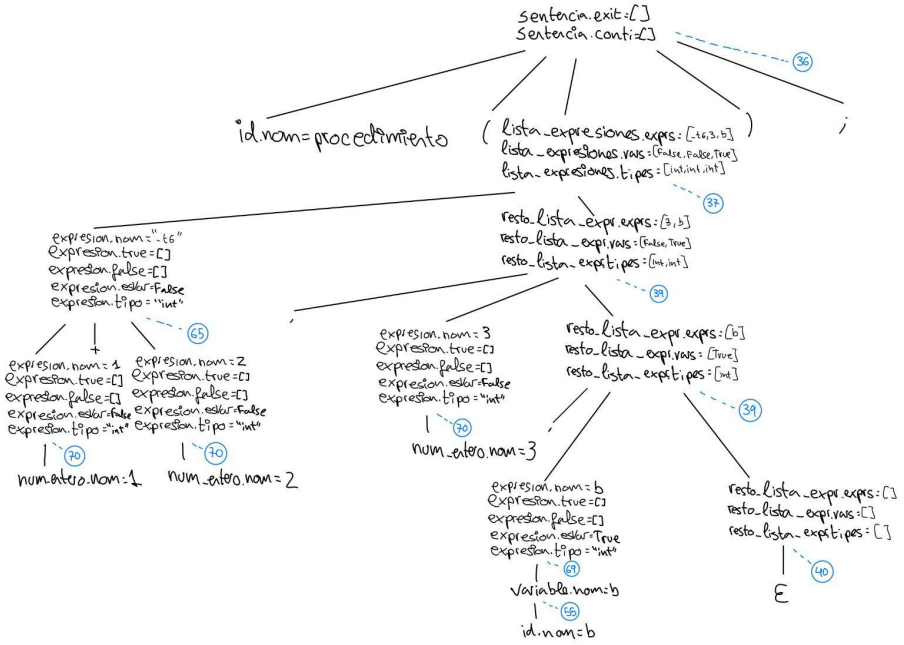
72. .
73. if 1 = a goto 75;
74. goto 78;
75. a := 2;
76. goto 84;
77. goto 80;
78. if 2 = a goto 80;
79. goto 83;
80. a := 1;
81. goto 84;
82. goto 83;
83. a := 0;
84. .

6.7. Prueba 7: Procedimiento

6.7.1. Programa

```
.
.
.
procedimiento(1+2,3,b);
.
.
.
```

6.7.2. Árbol decorado



6.7.3. Código intermedio generado

86. .
87. _t6 := 1 + 2;
88. param_val _t6;
89. param_val 3;
90. param_ref b;
91. call procedimiento;
92. .

7. Descripción de objetivos añadidos

7.1. Tratamiento de booleanas

Se han añadido las operaciones booleanas **and**, **or** y **not**. Estas se representan en el lenguaje especificado con las palabras reservadas and, or y not respectivamente, imitando la sintaxis de python. La precedencia de estos, de mayor a menor, sería not, and y or. A su vez, este nuevo set, serían los que menos precedencia tienen entre todos los demás operadores de expresiones (suma, resta, multiplicación, división y comparaciones).

Estas operaciones se han implementado única y exclusivamente en el no terminal expresion. Se han implementado mediante saltos; es decir, mediante un tratamiento estratégico de las instrucciones *goto* (condicionales e incondicionales). De esta manera, podremos simular las operaciones booleanas perfectamente.

En el caso de la operación *and*, si la primera expresión es cierta, se saltará a la línea que comienza a evaluar la siguiente expresión. En caso de ser falsa, no sería necesario evaluar nada más y se saltará a donde corresponda. Justo todo lo contrario a lo que pasa en el caso del *or*, ya que si la primera expresión es cierta se saltará a donde corresponda, pero si es falsa, el salto llevará a la siguiente expresión. Por último, la operación *not* únicamente intercambia las listas de los saltos a rellenar (expresion.true y expresion.false).

7.2. Nuevas estructuras de control

Se han implementado 3 estructuras de control nuevas que pueden recordarnos a algunas ya existentes en otros lenguajes populares. Todas estas estructuras han sido comentadas y consensuadas en tutoría.

7.2.1. Sentencia if-elseif-else

Esta sentencia recuerda al **if** ya existente en los objetivos básicos de la práctica. Esta estructura lo que permite es crear una sentencia condicional, pero si esta condición no se cumple, se pasará a evaluar la siguiente condición y así hasta encontrarnos con la última condición o con una cláusula **else** que recoja todo aquello que no ha cumplido ninguna de las condiciones.

Esto ha sido implementado en los no terminales sentencia y parte_else. Se ha seguido una estructura muy parecida a la del if-else implementado en las transparencias pero ampliándola para que acepte innumerados **elseif** de por medio; es decir, después de cada cláusula condicional, si esta condición se cumple, se salta al bloque de código, sino, se salta a la siguiente cláusula. Y, al final de cada bloque de código, se añade un salto al final de laa estructura para que una vez se haya ejecutado este código, no se evalúe nada más. Este último salto no se da en la cláusula **else** ya que después de esta, la estructura siempre termina.

7.2.2. Sentencia for

Esta estructura de control es un bucle. En casi cualquier lenguaje de programación, existe una definición de un bucle for. En el caso de este lenguaje, el bucle se ha definido de una manera mixta entre python y ADA. Al inicio del bucle, se declara una variable (ésta siempre se considerará de tipo entero) y de forma seguida hay 3 expresiones que definen cómo será el bucle. La primera y segunda expresión definen los límites de la variable definida. Esto significa que la variable al principio tomará el valor de la primera expresión y en cada iteración se acercará al valor de la segunda hasta llegar a este, o pasarse, y terminar el bucle. La tercera expresión define el salto entero que tomará la variable entre iteraciones. De esta manera, el bucle podrá ser común o reverso dependiendo del signo de esta expresión y, además, se podrán definir bucles en los que la variable aumenta disminuye una cantidad absoluta mayor que 1. Estas expresiones solo se evalúan una vez, al principio del bucle.

Su implementación toma lugar en el no terminal sentencia. Esta es la implementación común de un bucle que acepta tanto *break* como *continue* solo que con unas pocas particularidades. La primera sería que la comprobación de si la variable cumple con los límites de las expresiones se haría dinámicamente ya que dependerá completamente del signo del salto. La siguiente, y la última, sería que cuando existe un *continue*, el salto se hace al final del bloque ya que es ahí donde se actualiza la variable y, posteriormente, se salta para ver si sigue cumpliendo con los márgenes.

7.2.3. Sentencia switch-case

Esta estructura sigue una semántica casi idéntica de su homónima en C. Evalúa una expresión (entera o real) y comprueba si coincide con diversas constantes definidas en las cláusulas **case**. Se ejecutará todo el código de la estructura a partir de la primera coincidencia a pesar de que las constantes de las siguientes cláusulas **case** no coincidan. Esto se puede cortar con un *break* (el break incondicional es una sentencia que se ha tenido que introducir para simplificar el uso de esta estructura), que nos llevará al final de la estructura. Al final, se puede introducir una cláusula **default** que recogerá todos los casos que no hayan tenido ninguna coincidencia. No será posible utilizar una sentencia switch-case si dentro no hay ninguna cláusula case o default.

Su implementación se recoge en los no terminales sentencia, cte, cases y r_cases. Lo que se hace principalmente es un juego con los saltos; esto es, al principio se va a saltar de comprobación en comprobación hasta que suceda la primera coincidencia entre la expresión y una constante. En este caso, se ejecutará el bloque correspondiente y, al final de cada bloque de código hay un salto que evita la comprobación de la expresión con la siguiente constante y que salta directamente al siguiente bloque, simulando así, una sentencia switch-case. En caso de haber algún tipo de *break*, se saltará al final de la estructura.

7.3. Llamada a procedimientos

Se han implementado las llamadas a procedimientos de la manera que se pedía en el enunciado de la práctica. Para hacer la llamada simplemente se debe escribir el identificador del método seguido de la lista de expresiones, entre paréntesis, que encarnen los parámetros. En el caso de no haber parámetros, únicamente se escribirá el identificador del procedimiento seguido de paréntesis vacío.

Esto ha sido implementado en los no terminales sentencia, lista_expresiones y resto_lista_expr. Para ello, se ha utilizado la tabla de símbolos, que nos ayuda a saber que parámetros de la función son por valor o referencia. De esta manera, se puede hacer la llamada correctamente.

7.4. Análisis semántico

El compilador recogerá todos los errores estáticos encontrados durante el proceso de traducción y los imprimirá todos a la vez al final de este. Este método puede hacer que se generen errores colaterales, pero también hace más fácil la corrección completa del programa.

Para posibilitar la realización de un análisis semántico completo, se ha implementado la pila de tablas de símbolos que ayuda a gestionar los procedimientos, parámetros de procedimientos y variables. La tabla de símbolos en el tope siempre corresponderá a los elementos de mayor profundidad en ese momento, y las siguientes tablas de la pila serán de menor profundidad progresivamente.

7.4.1. Práctica básica

Para el análisis semántico de la parte básica de la práctica se han creado los siguientes errores:

- **Error estático 1:** Este error se lanza siempre que se intente declarar un identificador que ya haya sido declarado antes en el mismo nivel de profundidad.
- **Error estático 2:** Este error se lanza siempre que haya *breaks* (de cualquier tipo) que no correspondan a ninguna estructura de control que los acepte.
- **Error estático 3:** Este error se lanza siempre que haya *continues* que no correspondan a ninguna estructura de control que los acepte.
- **Error estático 4:** Este error se lanza cuando se referencia a un identificador que no existe en ningún nivel de profundidad igual o inferior.
- **Error estático 5:** Este error se lanza si a una variable se le asigna una expresión de un tipo que no sea el de la propia variable.

- **Error estático 6:** Este error se lanza siempre que se requiera una expresión booleana (en un if, while-else…) y se reciba una expresión de cualquier otro tipo.
- **Error estático 8:** Este error se lanza cuando se espera una expresión entera o real y, en cambio, se recibe una de otro tipo.
- **Error dinámico 1:** Este error se lanzará siempre que se haga una división por 0, ya que es una operación no definida.

7.4.2. Tratamiento de booleanas

Este objetivo no define ningún error nuevo, únicamente comprueba el tipo de las expresiones involucradas en las nuevas operaciones booleanas y lanza el **error 6** en caso de que alguno de esos tipos no sea el booleano.

7.4.3. Nuevas estructuras de control

Las nuevas estructuras de control reciclan los errores de la parte básica de la práctica y, además, añaden alguno.

7.4.3.1. Sentencia if-elseif-else

Esta estructura de control contiene exáctamente los mismos errores que tiene el if común. Esto quiere decir que solo se lanza el **error 6** en caso de que las expresiones no sean booleanas.

7.4.3.2. Sentencia for

Esta estructura de control lanza el **error 1** en caso de que la variable del bucle ya se encuentre declarada en la tabla de símbolos de la profundidad actual. También, una vez terminada la estructura, borrará de la tabla de símbolos la declaración de esta variable, ya que ésta solo existirá en el entorno del bucle. Asimismo, también se lanzan los siguientes errores nuevos:

- **Error estático 7:** Este error solo se lanzará si alguna de las tres expresiones características del bucle no son de tipo entero. Ya que esto es un requisito.
- **Error dinámico 2:** Este error salta si la tercera expresión del bucle (la que hace de salto), vale 0. Esto se debe a que el valor 0 como salto volvería el bucle infinito.
- **Error dinámico 3:** Este error se da cuándo, teniendo en cuenta la expresión de inicio, la de final y la de salto, se calcula que la variable nunca podrá llegar desde la expresión primera hasta la segunda con el salto definido. Esto se da si la primera expresión es mayor que la segunda y el salto es positivo o, si la primera es menor que la segunda y el salto es negativo.

7.4.3.3. Sentencia switch-case

Esta estructura lanza el **error 8** si la expresión del switch es de un tipo no compatible (distinto de entero o real). También define un nuevo tipo de error:

- **Error estático 9:** Este error se da siempre que una constante de una cláusula case no cuadre exactamente con el tipo que posee la expresión del switch.

7.4.4. Llamada a procedimientos

Las llamadas a procedimientos hacen uso de los **errores**, ya definidos, **4** y **5**. Estos se dan si el identificador del procedimiento no está definido en ninguno de los niveles de profundidad accesibles desde la pila de tablas de símbolos y si el tipo de alguno de los parámetros no cuadra exactamente con la expresión que le corresponde, respectivamente. También se definen nuevos errores para este objetivo:

- **Error estático 10:** Este error salta si, al llamar al procedimiento, se le ha llamado con un número de parámetros diferente al establecido en la definición del mismo.
- **Error estático 11:** Este error se da en el caso de que el procedimiento requiera como parámetro un valor por referencia y no se le pase una variable. Esto se debe a que una simple expresión no ocupa lugar en memoria al que se pueda referenciar.

8. Anexos

8.1 Anexo primero

Pdf con mejor calidad del árbol decorado de la primera prueba. click aquí.

8.2 Anexo segundo

Pdf con mejor calidad del árbol decorado de la segunda prueba. click aquí.

8.3 Anexo tercero

Pdf con mejor calidad del árbol decorado de la tercera prueba. click aquí.

8.4 Anexo cuarto

Pdf con mejor calidad del árbol decorado de la cuarta prueba. click aquí.

8.5 Anexo quinto

Pdf con mejor calidad del árbol decorado de la quinta prueba. click aquí.

8.6 Anexo sexto

Pdf con mejor calidad del árbol decorado de la sexta prueba. click aquí.

8.7 Anexo séptimo

Pdf con mejor calidad del árbol decorado de la séptima prueba. click aquí.