```
//*****************************************************************
// File:   cl.jj
// Author: Procesadores de Lenguajes-University of Zaragoza
// Date:   julio 2023
// Coms:   compilar mediante "ant"
//*****************************************************************

options {
IGNORE_CASE = true;
COMMON_TOKEN_ACTION = false;
DEBUG_PARSER = true;
}

// ----------------------------------------------------------------------------


PARSER_BEGIN(alike)

package traductor;

import lib.symbolTable.*;
import lib.symbolTable.exceptions.*;
import lib.attributes.*;
import java.util.ArrayList;
import lib.errores.ErrorSemantico;
import lib.tools.codeGeneration.CodeBlock;
import lib.tools.codeGeneration.PCodeInstruction.OpCode;
import lib.tools.codeGeneration.CGUtils;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

//...

public class alike {
//...

// Para los mensajes de depuración:
public static final String ANSI_RESET = "\u001B[0m";
public static final String ANSI_RED = "\u001B[31m";
public static final String ANSI_YELLOW = "\u001B[33m";

static SymbolTable st;

private static void initSymbolTable() {
boolean b;
String[] palsRes = {
"boolean","char","character", "integer", "null", "array",
"mod", "not", "and", "or",
"if", "elsif", "else", "then", "while", "loop", "true", "false",
"procedure", "function", "is", "ref", "of", "begin", "end", "return",
"skip_line", "put", "put_line", "get", "char2int", "int2char"
};

//st.insertReservedWords(palsRes);
}

public static String obtenerNombreArchivo(String ruta) {
File archivo = new File(ruta);
```

```
return archivo.getName();
}

private static void iterarYanadirEnTablaDeSimbolos(ArrayList<Token> ids, Attributes at){
Symbol s = null;
for (Token t : ids) {
if (at.isArray) {
s = new SymbolArray(t.image, at.intList.get(0), at.intList.get(1), at.type, at.parClass);
at.parList.add(s);
}
else {
if (at.type == Symbol.Types.BOOL) {
s = new SymbolBool(t.image, at.parClass);
at.parList.add(s);
}
else if (at.type == Symbol.Types.INT) {
s = new SymbolInt(t.image, at.parClass);
at.parList.add(s);
}
else if (at.type == Symbol.Types.CHAR) {
s = new SymbolChar(t.image, at.parClass);
at.parList.add(s);
}
}
try {
st.insertSymbol(s);
}
catch(AlreadyDefinedSymbolException e) {
System.err.println("Already Defined: " + e.getMessage());
}
}
}

public static void escribirEnNuevoArchivo(String nomFich, String msg) throws IOException {
String nomFichPCODE = obtenerNombreArchivoPCode(nomFich);
File nuevoArchivo = new File(nomFichPCODE);
try (BufferedWriter writer = new BufferedWriter(new FileWriter(nuevoArchivo))) {
writer.write(msg);
}
}

public static String obtenerNombreArchivoPCode(String nomFich) {
return nomFich.replaceAll("\\.\\w+$", ".pcode");
}

public static void main(String[] args) {
alike parser = null;
Attributes at = new Attributes();
String nombreFichero = obtenerNombreArchivo(args[0]);

st = new SymbolTable();
initSymbolTable();

try {
if(args.length == 0) {
parser = new alike(System.in);
}
else {
parser = new alike(new java.io.FileInputStream(args[0]));
}
```

```
//Programa es el símbolo inicial de la gramática
parser.Programa(at);

if(ErrorSemantico.ERR_SEMANTICO){
System.out.println(ANSI_RED + "***** Analisis terminado sin exito, errores semanticos *****" + ANSI_RESET
}
else{
System.out.println("***** Compilacion finalizada. Se ha generado el fichero " + obtenerNombreArchivoPCode
try {
escribirEnNuevoArchivo(nombreFichero, at.code.toString());
} catch (IOException e) {
System.out.println("Error al escribir en el archivo: " + e.getMessage());
}
}
}
catch (java.io.FileNotFoundException e) {
System.err.println ("Fichero " + args[0] + " no encontrado.");
}
catch (TokenMgrError e) {
System.err.println("LEX_ERROR: " + e.getMessage());

}
catch (ParseException e) {
System.err.println("Parse_exception: " + e.getMessage());
}
}
}
PARSER_END(alike)

// ----------------------------------------------------------------------

TOKEN : {
< #LETTER: ["a"-"z","A"-"Z"] >
|■< #DIGIT: ["0"-"9"] >
|■< #UNDERSCORE: "_" >
}

SKIP : {
< tSPACE: " " >
| ■< tNL: "\n" >
|■< tTAB: "\t" >
|■< tENTER: "\r" >
| ■< tCOMENTARIO: "--"(~["\n"])* >
}

TOKEN : { // Tipos
< tBOOL: "boolean" >
|■< tCHAR: "char" >
|■< tCHARACTER: "character" >
|   < tINTEGER: "integer" >
|■< tSTRING: "string" >
|■< tNULL: "null" >
|■< tARRAY: "array" >
|■< tCONST_INT: ["0"-"9"])+ >
|■< tCONST_CHAR: "\'"(~["\n","\t","\r","\f","\b","\'","\""] | ("\'\"") | ("\'\'"))"\'" >
| ■< tCONST_STRING: "\""(~["\n","\t","\r","\f","\b","\'","\""] | ("\'\"") | ("\'\'"))*"\"" >
}

TOKEN : { // Operadores
< tASIGN: ":=" >
```

```
|■< tSUM: "+" >
|   < tRES: "-" >
|■< tEQU: "=" >
|■< tGT: ">" >
|■< tLT: "<" >
|■< tGE: ">=" >
|■< tLE: "<=" >
|■< tDIF: "/=" >
|■< tMUL: "*" >
|■< tMOD: "mod" >
|■< tDIV: "/" >
|■< tNOT: "not" >
|■< tAND: "and" >
|■< tOR: "or" >
}

TOKEN : { // Sintaxis reservada estructuras simples
< tIF: "if" >
|■< tELIF: "elsif" >
|■< tELSE: "else" >
|■< tTHEN: "then" >
|■< tWHILE: "while" >
|■< tLOOP: "loop" >
|■< tTRUE: "true" >
|■< tFALSE: "false" >
}

TOKEN : { // Sintaxis reservada funciones
< tPROCEDURE: "procedure" >
|■< tFUNCTION: "function" >
|■< tIS: "is" >
|■< tREF: "ref" >
|■< tOF: "of" >
|■< tBEGIN: "begin" >
|■< tEND: "end" >
|■< tRETURN: "return" >
}

TOKEN : { // Instrucciones E/S
< tSKIP_LN: "skip_line" >
|■< tPUT: "put" >
|■< tPUT_LN: "put_line" >
|■< tGET: "get" >
|■< tCHAR2INT: "char2int" >
|■< tINT2CHAR: "int2char" >
}

TOKEN : { // Separadores
< tPUNTO: "." >
|■< tCOMA:■"," >
|■< tDOSPUNTOS: ".." >
|■< tDOBLEPUNTO: ":" >
|■< tPUNTOCOMA: ";" >
|■< tCORCHETES_OPEN: "[" >
|■< tCORCHETES_CLOSE: "]" >
|■< tPARENTESIS_OPEN: "(" >
|■< tPARENTESIS_CLOSE: ")" >
|■< tID: ["a"-"z","A"-"Z","_"](["a"-"z","A"-"Z","0"-"9","_"])* >
}
```

```
TOKEN_MGR_DECLS : {
static void CommonTokenAction(Token t) {

String[] nombresTokens = {
"letra", "digito", "barra baja", "espacio", "fin de linea", "tabulador", "retorno", "comentario",
"boolean", "char", "character", "integer", "null", "array", "const integer", "const char", "const string",
"asignacion", "suma", "resta", "comparacion igualdad", "comparacion mayor estricto", "comparacion menor e
"comparacion mayor o igual", "comparacion menor o igual", "comparacion diferencia", "multiplicacion", "mod
"operador logico", "operador logico", "token", "token", "token", "token", "token", "token", "constante aritmetica
"token", "token", "token", "token", "token", "token", "token", "token", "funcion reservada", "funcion reservada",
"funcion reservada", "funcion reservada", "funcion reservada", "token separador", "token separador", "token
"token separador", "token separador", "token separador", "token separador", "token separador", "identificado
};

int linea = t.beginLine;
int columna = t.beginColumn;
String image = t.image;
if (t.kind != 0){ // Si es distinto de fin de linea
System.out.println("(" + linea + "," + columna + "): " + nombresTokens[t.kind-1] + " \"" + image + "\"");
}
}
}


//---------- Símbolo inicial de la gramática. Para análisis léxico no hace falta más
/*void Programa() :
{
//Token t;
}
{
(
< tBEGIN >
|
< tBOOL > | < tCHAR > | < tCHARACTER > | < tINTEGER > | < tNULL > | < tARRAY > | < tCONST_INT > |
|
< tASIGN > | < tSUM > | < tRES > | < tEQU > | < tGT > | < tLT > | < tGE > | < tLE > | < tMUL > | < tMOD > | <
|
< tIF > | < tELIF > |■< tTHEN > |■< tWHILE > | < tLOOP > | < tTRUE > | < tFALSE > | <tELSE>
|
< tPROCEDURE > | < tIS > | < tREF > | < tEND > | < tRETURN > | < tOF > | <tFUNCTION>
|
< tSKIP_LN > | < tPUT > | < tPUT_LN > | < tGET > | <tCHAR2INT> | <tINT2CHAR>
|
< tPUNTO > |■< tCOMA > |■< tDOSPUNTOS > | < tDOBLEPUNTO > | < tPUNTOCOMA > |■< tCORCHET
)+
< EOF >
}*/


void Programa(Attributes at) :
{
Token t;
Attributes at1 = new Attributes();

}
{
<tPROCEDURE>
t = <tID> {
Symbol s;
at1.parList = new ArrayList<Symbol>();
```

```
s = new SymbolProcedure(t.image,at1.parList,true);
try {
st.insertSymbol(s);
}
catch (AlreadyDefinedSymbolException e) {
ErrorSemantico.deteccion(e, t.image);
}
// ---------------------------------------------
String etiqINI_PROGRAM = CGUtils.newLabel();
at.code.addInst(OpCode.ENP, etiqINI_PROGRAM);
}

<tIS>
( declaracion_variables() )?
( declaracion_procs_funcs(at) )?
<tBEGIN> {at.code.addLabel(etiqINI_PROGRAM);}
instrucciones(at1) {at.code.addBlock(at1.code);}
<tEND>
<tPUNTOCOMA>
{
System.err.println(st.toString());
st.removeBlock();
}
<EOF> {at.code.addInst(OpCode.LVP);}
}

void declaracion_procs_funcs(Attributes at) :
{
Attributes at1 = new Attributes();
}
{
( declaracion_proc_func(at1) {at.code.addBlock(at1.code);} )+
}

void declaracion_proc_func(Attributes at) :
{

}
{
( declaracion_proc(at) | declaracion_func(at) )
}

void declaracion_func(Attributes at):
{
Attributes at1 = new Attributes(), at2 = new Attributes(), at3 = new Attributes(), at4 = new Attributes();
}
{
cabecera_funcion(at1)
( declaracion_variables() )?
( declaracion_proc_func(at3) )*
<tBEGIN>
instrucciones_return(at4)
<tEND>
<tPUNTOCOMA>
{
System.err.println(st.toString());
st.removeBlock();
at.code.addBlock(at1.code);
//at.code.addBlock(at2.code);
at.code.addBlock(at3.code);
```

```
at.code.addBlock(at4.code);
}
}

void declaracion_proc(Attributes at):
{
Attributes at1 = new Attributes(), at2 = new Attributes(), at3 = new Attributes(), at4 = new Attributes();
}
{
cabecera_procedimiento(at1)
( declaracion_variables() )?
( declaracion_proc_func(at3) )*
<tBEGIN>
instrucciones(at4)
<tEND>
<tPUNTOCOMA>
{
System.err.println(st.toString());
st.removeBlock();
at.code.addBlock(at1.code);
//at.code.addBlock(at2.code);
at.code.addBlock(at3.code);
at.code.addBlock(at4.code);
}
}

ArrayList<Token> lista_ids():
{
ArrayList<Token> ids = new ArrayList<Token>();
Token t;
}
{

t = <tID> {ids.add(t);} (<tCOMA> t = <tID> {ids.add(t);})*  //Aceptamos que una lista de ids pueda estar forma
{return ids;}
}

void declaracion_variables() :
{

}
{
( declaracion_var() )+
}

void declaracion_variables_puntocoma() :
{
Attributes at = new Attributes();
}
{
( declaracion_var_puntocoma(at) )+
}

void tipo_variable(Attributes at) :
{

}
{
<tBOOL> { at.type = Symbol.Types.BOOL; }
| ■<tCHAR> { at.type = Symbol.Types.CHAR; }
```

```
| ■<tCHARACTER> { at.type = Symbol.Types.CHAR; }
| ■<tINTEGER> { at.type = Symbol.Types.INT; }

}

void rango(Attributes at) :
{
Token t1, t2;
Boolean res1 = false;
Boolean res2 = false;
}
{
(<tRES> {res1 = true;})? t1 = <tCONST_INT> <tDOSPUNTOS> (<tRES> {res2 = true;})? t2 = <tCONST_INT
{
Integer inicio,fin;
inicio = Integer.valueOf(t1.image);
fin = Integer.valueOf(t2.image);
if(res1){
inicio = inicio * -1;
}
if(res2){
fin = fin * -1;
}
if(inicio > fin) {
//System.out.println(ANSI_YELLOW + inicio + ", " + fin + ANSI_RESET);
ErrorSemantico.deteccion("Rango invalido");
}
else {
at.intList.add(inicio);
at.intList.add(fin);
}
}
}

void estructura_array(Attributes at):
{
}
{
<tARRAY> <tPARENTESIS_OPEN> rango(at) <tPARENTESIS_CLOSE> <tOF> tipo_variable(at)
{
at.isArray = true;
// Faltarían más cosas de atribuir a at?
}
}

void declaracion_var_puntocoma(Attributes at):
{
ArrayList<Token> ids;
Attributes at1 = new Attributes(), at2 = new Attributes();
Symbol s;
}
{
ids = lista_ids() <tDOBLEPUNTO> {
at1.parClass = Symbol.ParameterClass.VAL;
at2.parClass = Symbol.ParameterClass.VAL;
}
(<tREF> {
at1.parClass = Symbol.ParameterClass.REF;
at2.parClass = Symbol.ParameterClass.REF;
})?
```

```
( tipo_variable(at1)
{ iterarYanadirEnTablaDeSimbolos(ids,at1);
at.parList = at1.parList;
}
|
estructura_array(at2) // Aquí se supone que permitimos vectores ya sea por valor o ref como parametros
{ iterarYanadirEnTablaDeSimbolos(ids,at2);
at.parList = at2.parList;
}
)
}

void declaracion_var():
{
ArrayList<Token> ids;
Attributes at1 = new Attributes(), at2 = new Attributes();
}
{
ids = lista_ids()
<tDOBLEPUNTO>
( tipo_variable(at1) { iterarYanadirEnTablaDeSimbolos(ids,at1);■} |
estructura_array(at2) { iterarYanadirEnTablaDeSimbolos(ids,at2); })
<tPUNTOCOMA>
}

void lista_parametros_funcion_o_proc(Attributes at):
{
Attributes at1 = new Attributes(), at2 = new Attributes();
}
{
declaracion_var_puntocoma(at1)
{
for (Symbol s : at1.parList) {
at.parList.add(s);
}
}
(
<tPUNTOCOMA> declaracion_var_puntocoma(at2)
{
for (Symbol s : at2.parList) {
at.parList.add(s);
}
}
)*
}

void cabecera_procedimiento(Attributes at) :
{
Token t;
Attributes at1 = new Attributes(), at2 = new Attributes();
}
{
<tPROCEDURE>
t = <tID> {
Symbol s;
at.parList = new ArrayList<Symbol>();
s = new SymbolProcedure(t.image,at.parList);
try {
st.insertSymbol(s);
st.insertBlock();
```

```
}
catch (AlreadyDefinedSymbolException e) {
ErrorSemantico.deteccion(e,t.image);
}

at2.parList = at.parList;
}
(<tPARENTESIS_OPEN>
( lista_parametros_funcion_o_proc(at2) )
{
try {
Symbol aux = st.getSymbol(t.image);
if (aux instanceof SymbolProcedure) {
//System.err.println("Procedimiento");
SymbolProcedure procedure = (SymbolProcedure) aux;
procedure.parList = at2.parList;
//System.out.println(ANSI_YELLOW + "CABECERA_PROCEDIMIENTO: " + procedure.parList.size() + ANS
}
}
catch (SymbolNotFoundException e) {
ErrorSemantico.deteccion(e,t.image);
}
}
<tPARENTESIS_CLOSE>)?
<tIS>
}



void cabecera_funcion(Attributes at) :
{

Token t;
Attributes at1 = new Attributes(), at2 = new Attributes();

}
{
<tFUNCTION>
t = <tID> {
Symbol s;
at.parList = new ArrayList<Symbol>();
s = new SymbolFunction(t.image, at.parList, at1.type);
try {
st.insertSymbol(s);
st.insertBlock();
}
catch (AlreadyDefinedSymbolException e) {
ErrorSemantico.deteccion(e,t.image);
}
at2.parList = at.parList;
}
( <tPARENTESIS_OPEN> lista_parametros_funcion_o_proc(at2) <tPARENTESIS_CLOSE> ) ?
<tRETURN> tipo_variable(at1) <tIS>
{
try {
Symbol aux = st.getSymbol(t.image);
if (aux instanceof SymbolFunction) {
//System.err.println("Funcion");
SymbolFunction funcion = (SymbolFunction) aux;
funcion.returnType = at1.type;
```

```
funcion.parList = at2.parList;
}
}
catch (SymbolNotFoundException e) {
ErrorSemantico.deteccion(e, t.image);
}
}
}



void inst_leer(Attributes at):
{
ArrayList<Token> ids;
}
{
<tGET> <tPARENTESIS_OPEN>

ids = lista_ids() { // NO, es una lista de asignables, o son IDs o son un ID(expresion)
for(Token t : ids) {
try {
Symbol s = st.getSymbol(t.image);
if (!((s.type == Symbol.Types.CHAR) || (s.type == Symbol.Types.INT))) {
ErrorSemantico.deteccion("Se esperaba caracter o entero <inst_leer>");
}
if (s.type == Symbol.Types.INT) {
at.code.addInst(OpCode.SRF, (st.level - s.nivel), (int) s.dir);
at.code.addInst(OpCode.RD, 1);
}
else if (s.type == Symbol.Types.CHAR) {
at.code.addInst(OpCode.SRF, (st.level - s.nivel), (int) s.dir);
at.code.addInst(OpCode.RD, 0);
}
}
catch(SymbolNotFoundException e){
ErrorSemantico.deteccion(e, t.image);
}
}
}
<tPARENTESIS_CLOSE>
}

ArrayList<Token> lista_asignables():
{
ArrayList<Token> ids = new ArrayList<Token>();
Token t;
}
{

t = <tID> {ids.add(t);} (<tCOMA> t = <tID> {ids.add(t);})*  //Aceptamos que una lista de ids pueda estar forma
{return ids;}
}



void inst_saltar_linea(Attributes at):
{

}
{
```

```
<tSKIP_LN> {
at.code.addInst(OpCode.RD, 0);
at.code.addInst(OpCode.POP);
}
}



void inst_escribir(Attributes at):
{
ArrayList<Attributes> ats = new ArrayList<Attributes>();
}
{
<tPUT> <tPARENTESIS_OPEN>
lista_una_o_mas_exps(ats)
{
for (Attributes att : ats) {
if (!((att.type == Symbol.Types.INT) || (att.type == Symbol.Types.BOOL) ||
(att.type == Symbol.Types.CHAR) || (att.type == Symbol.Types.STRING))) {
ErrorSemantico.deteccion("Se esperaba entero, booleano, caracter o string <inst_escribir>");
}
if (att.type == Symbol.Types.INT) {
at.code.addInst(OpCode.SRF, st.level, 3);
at.code.addInst(OpCode.DRF);
at.code.addInst(OpCode.WRT, 1);
}
else if (att.type == Symbol.Types.CHAR) {
at.code.addInst(OpCode.SRF, st.level, 3);
at.code.addInst(OpCode.DRF);
at.code.addInst(OpCode.WRT, 0);
}
}
}
<tPARENTESIS_CLOSE>
}



void inst_escribir_linea(Attributes at):
{
ArrayList<Attributes> ats = new ArrayList<Attributes>();
}
{
<tPUT_LN>
(<tPARENTESIS_OPEN>
lista_una_o_mas_exps(ats)
{
for (Attributes att : ats) {
if (!((att.type == Symbol.Types.INT) || (att.type == Symbol.Types.BOOL)
|| (att.type == Symbol.Types.CHAR) || (att.type == Symbol.Types.STRING))) {
System.err.println(att.type);
ErrorSemantico.deteccion("Se esperaba entero, booleano, caracter o string <inst_escribir_linea>");
}
if (att.type == Symbol.Types.INT) {
at.code.addInst(OpCode.SRF, st.level, 3);
at.code.addInst(OpCode.DRF);
at.code.addInst(OpCode.WRT, 1);
}
else if (at.type == Symbol.Types.CHAR) {
at.code.addInst(OpCode.SRF, st.level, 3);
```

```
at.code.addInst(OpCode.DRF);
att.code.addInst(OpCode.WRT, 0);
}
}
}
<tPARENTESIS_CLOSE>) ?
}



void inst_invocacion_o_asignacion(Attributes at):
{
Attributes at1 = new Attributes(), at2 = new Attributes();
}
{■// o componente de vector, o <tID> (cambiarlo por expresion), o procedimiento (con y sin parametros)
expresion(at1) {
at.code = at1.code;
try {
Symbol s = st.getSymbol(at1.name);
if ((s instanceof SymbolProcedure) && (((SymbolProcedure) s).principal)) {
ErrorSemantico.deteccion("El procedimiento principal no es invocable");
}
if ((s instanceof SymbolProcedure) || (s instanceof SymbolFunction)) {
//at.code.addInst(OpCode)
}
}
catch(SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at1.name);
}
}
(<tASIGN>
expresion(at2) {
Symbol s = null, s2 = null;
//System.out.println("----------->" + at1.name);
//System.out.println("----------->" + at2.name);
try {
s = st.getSymbol(at1.name);

// Evitamos también que entre si los nombres son los de las ctes, porque al intentar obtener el símbolo dara
if (at2.name != "" && at2.name != "TRUE" && at2.name != "FALSE" && at2.name != "CONST_INT"  && at2.r
s2 = st.getSymbol(at2.name);
if (s2 instanceof SymbolProcedure) {
ErrorSemantico.deteccion("No se puede asignar un procedimiento, no devuelve nada");
}
else if (s2 instanceof SymbolFunction){
// Si variable es escalar y tipos at1 y at2 iguales -> OK
// Doy por asumido que escalares agrupa tmb char, string y bool
if(!((at1.type == Symbol.Types.INT || at1.type == Symbol.Types.CHAR ||
at1.type == Symbol.Types.BOOL) && at1.type == ((SymbolFunction)s2).returnType)){
ErrorSemantico.deteccion("1.Asignacion con tipos distintos: " + at1.type + ":= " + ((SymbolFunction)s2).retur
}
// Si es una componente de vector y tipos at1 y at2 iguales -> OK
if(at1.isVecComp && (((SymbolArray) s).baseType != ((SymbolFunction)s2).returnType)){
ErrorSemantico.deteccion("La componente del vector no es del tipo del vector");
}
}
}

// Funciones y procedimientos no pueden ser asignables.
if (s instanceof SymbolFunction || s instanceof SymbolProcedure) {
```

```
ErrorSemantico.deteccion("Funcion o procedimiento no son asignables");
}
// Para el caso de que no se asigne una función, sino cualquier otro valor escalar.
else if (!(s2 instanceof SymbolFunction || s2 instanceof SymbolProcedure)){
// Si es una componente de vector y tipos at1 y at2 iguales -> OK
if(s instanceof SymbolArray){
if(at1.isVecComp && (((SymbolArray) s).baseType != at2.type)){
System.out.println(at1.name);
ErrorSemantico.deteccion("La componente del vector no es del tipo del vector");
}
}
else{
// Si variable es escalar y tipos at1 y at2 iguales -> OK
// Doy por asumido que escalares agrupa tmb char, string y bool
if(!((at1.type == Symbol.Types.INT || at1.type == Symbol.Types.CHAR ||
at1.type == Symbol.Types.BOOL) && at1.type == at2.type)){
ErrorSemantico.deteccion("2.Asignacion con tipos distintos: " + at1.type + ":= " + at2.type);
}
}
}
}
catch (SymbolNotFoundException e){
ErrorSemantico.deteccion(e, ("at1.name: " + at1.name + ", at2.name: " + at2.name));
}
// ---------------------- GENERACION DE CODIGO -----------------------------
//at.code.addBlock(at1.code);
at.code.addBlock(at2.code);
at.code.addInst(OpCode.ASG);
}
) ?

}



void inst_if(Attributes at):
{
Attributes at1 = new Attributes(), at2 = new Attributes(), at3 = new Attributes(), at4 = new Attributes(), at5 = r
}
{
<tIF> expresion(at1)
{
at.code = at1.code;
String etiqSINO = CGUtils.newLabel();
at.code.addInst(OpCode.JMF, etiqSINO);
if (at1.type != Symbol.Types.BOOL) {
ErrorSemantico.deteccion("Se esperaba booleano <if> 1");
}
}
<tTHEN> instrucciones_return(at3) {
String etiqFIN = CGUtils.newLabel();
at.code.addBlock(at3.code);
at.code.addInst(OpCode.JMP, etiqFIN);
at.code.addLabel(etiqSINO);
}
(<tELIF>  expresion(at2)
{
at.code.addBlock(at2.code);
String etiqSINO2 = CGUtils.newLabel();
at.code.addInst(OpCode.JMF, etiqSINO2);
```

```
if (at2.type != Symbol.Types.BOOL) {
ErrorSemantico.deteccion("Se esperaba booleano <if> 2");
}
}
<tTHEN> instrucciones_return(at4) {
at.code.addBlock(at4.code);
at.code.addInst(OpCode.JMP, etiqFIN);
at4.code = new CodeBlock();
at.code.addLabel(etiqSINO2);
})*
(<tELSE> instrucciones_return(at5) {
at.code.addBlock(at5.code);}
)?
<tEND> <tIF> {
at.code.addLabel(etiqFIN);
}

}



void inst_while(Attributes at):
{
Attributes at1 = new Attributes(), at2 = new Attributes();
}
{
<tWHILE> {
String etiqExp = CGUtils.newLabel();
at.code.addLabel(etiqExp);
}
expresion(at1) {
at.code.addBlock(at1.code);
if (at1.type != Symbol.Types.BOOL) {
ErrorSemantico.deteccion("Se esperaba booleano <while>");
}
String etiqFin = CGUtils.newLabel();
at.code.addInst(OpCode.JMF, etiqFin);
}
<tLOOP> // Verificar expresión, debe ser booleano
instrucciones_return(at2) {at.code.addBlock(at2.code);}
<tEND> <tLOOP> {
at.code.addInst(OpCode.JMP, etiqExp);
at.code.addLabel(etiqFin);
}
}



void inst_return(Attributes at):
{
//Attributes at = new Attributes();
}
{
<tRETURN> expresion(at) {
if (!((at.type == Symbol.Types.INT) || (at.type == Symbol.Types.BOOL) || (at.type == Symbol.Types.CHAR)))
String _error = "Tipo incompatible a devolver en return, (" + at.type.toString() + ") <inst_return>";
ErrorSemantico.deteccion(_error);
}
```

```
if (at.isConst) {
at.code.addInst(OpCode.STC, at.value);
at.code.addInst(OpCode.CSF);
}
else {
try {
Symbol s = st.getSymbol(at.name);

if (s instanceof SymbolFunction) {
at.code.addInst(OpCode.CSF);
}
else {
at.code.addInst(OpCode.SRF, (st.level - s.nivel), (int)s.dir);
at.code.addInst(OpCode.DRF);
at.code.addInst(OpCode.CSF);
}
} catch (SymbolNotFoundException e) {
ErrorSemantico.deteccion(e, at.name);
}
}
}
}



void inst_null(Attributes at):
{

}
{
<tNULL> {at.code.addInst(OpCode.NOP);}
}



void instruccion_return(Attributes at) :
{

}
{
( inst_leer(at) )
|   ( inst_saltar_linea(at) )
|■( inst_escribir(at) )
|   ( inst_escribir_linea(at) )
|   ( inst_invocacion_o_asignacion(at) )
|   ( inst_if(at) )
|   ( inst_while(at) )
|   ( inst_return(at) )
|   ( inst_null(at) )
}


void instruccion(Attributes at) :
{

}
{
( inst_leer(at) )
|   ( inst_saltar_linea(at) )
|■( inst_escribir(at) )
```

```
|   ( inst_escribir_linea(at) )
|   ( inst_invocacion_o_asignacion(at) )
|   ( inst_if(at) )
|   ( inst_while(at) )
|   ( inst_null(at) )
}


void instrucciones(Attributes at) :
{
Attributes at1 = new Attributes();
}
{
(instruccion(at1) {at.code.addBlock(at1.code);} <tPUNTOCOMA>)+
}

void instrucciones_return(Attributes at) :
{
Attributes at1 = new Attributes();
}
{
(instruccion_return(at1) {at.code.addBlock(at1.code);} <tPUNTOCOMA>)+
}


void expresion(Attributes at) :
{
Attributes at1 = new Attributes(), at2 = new Attributes();
Integer operador = -1;
}
{
relacion(at1) {
at.type = at1.type;
at.isVecComp = at1.isVecComp;
at.name = at1.name;
at.isConst = at1.isConst;
// ----------------------------------------
at.code = at1.code;
System.out.println(ANSI_YELLOW + "<expresion> " + at.code + ANSI_RESET);
}
(((<tAND> {operador = 0;}  relacion(at2)
{
//System.out.println(ANSI_YELLOW + at1.type + ", " + at2.type + ANSI_RESET);
if (!((at1.type == at2.type) && (at1.type == Symbol.Types.BOOL))) {
System.out.println(ANSI_YELLOW + "at1.type: " + at1.type + ", at2.type: " + at2.type + ANSI_RESET);
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Se esperaban booleanos");
}
else{
at.type = at1.type;
}
//at.code.addBlock(at1.code); sobra
at.code.addBlock(at2.code);
switch(operador) {
case 0:
at.code.addInst(OpCode.AND);
break;
case 1:
at.code.addInst(OpCode.OR);
break;
```

```
}
}
)+
|
(<tOR> {operador = 1;} relacion(at2)
{
//System.out.println(ANSI_YELLOW + at1.type + ", " + at2.type + ANSI_RESET);
if (!((at1.type == at2.type) && (at1.type == Symbol.Types.BOOL))) {
System.out.println(ANSI_YELLOW + "at1.type: " + at1.type + ", at2.type: " + at2.type + ANSI_RESET);
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Se esperaban booleanos");
}
else{
at.type = at1.type;
}
//at.code.addBlock(at1.code); //sobra
at.code.addBlock(at2.code);
switch(operador) {
case 0:
at.code.addInst(OpCode.AND);
break;
case 1:
at.code.addInst(OpCode.OR);
break;
}
}
)+))?
}



void lista_una_o_mas_exps(ArrayList<Attributes> ats):
{
Attributes at1 = new Attributes(), at2 = new Attributes();
}
{
expresion(at1) {ats.add(at1); /*System.out.println(ANSI_YELLOW + at1.name + ", " + at1.type + ANSI_RESE
(<tCOMA>
expresion(at2) {
ats.add(at2);
/*System.out.println(ANSI_YELLOW + at1.name + ", " + at1.type + ANSI_RESET);*/
}
)*

}



/* CREO QUE ESTÁ COMPLETADA, SI FALTA ALGO SERÍA COMPLETAR VALORES DE at */
void relacion(Attributes at) :
{
Attributes at1 = new Attributes(), at2 = new Attributes();
ArrayList<Integer> operador = new ArrayList<Integer>();
}
{
expresion_simple(at1)
{
if (at1.name != "TRUE" && at1.name != "FALSE" && at1.name != "CONST_INT"  && at1.name != "CONST_
try{
Symbol s = st.getSymbol(at1.name);
```

```
if(s instanceof SymbolFunction){
at.type = ((SymbolFunction)s).returnType;
}
else if (s instanceof SymbolArray){
at.type = ((SymbolArray)s).baseType;
}
else {
at.type = at1.type;
}
}
catch(SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at1.name);
}
}
else {
at.type = at1.type;
}
at.name = at1.name;
at.isConst = at1.isConst;
at.isVar = at1.isVar;
at.isVecComp = at1.isVecComp;
// ----------------------------------------------------
at.code = at1.code;

}
(
operador_relacional(operador)
expresion_simple(at2)
{
if ((at1.name != "TRUE" && at1.name != "FALSE" && at1.name != "CONST_INT"  && at1.name != "CONST_
(at2.name != "TRUE" && at2.name != "FALSE" && at2.name != "CONST_INT"  && at2.name != "CONST_CH
try{
Symbol s = st.getSymbol(at1.name);
Symbol s2 = st.getSymbol(at2.name);
if(s instanceof SymbolFunction){
if(s2 instanceof SymbolFunction){
if(((SymbolFunction)s).returnType == ((SymbolFunction)s2).returnType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else if (s2 instanceof SymbolArray){
if(((SymbolFunction)s).returnType == ((SymbolArray)s2).baseType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else{
if(((SymbolFunction)s).returnType == at2.type){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
```

```
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
}
else if (s instanceof SymbolArray){
if(s2 instanceof SymbolFunction){
if(((SymbolArray)s).baseType == ((SymbolFunction)s2).returnType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else if (s2 instanceof SymbolArray){
if(((SymbolArray)s).baseType == ((SymbolArray)s2).baseType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else{
if(((SymbolArray)s).baseType == at2.type){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
}
else {
if(s2 instanceof SymbolFunction){
if(at1.type == ((SymbolFunction)s2).returnType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else if (s2 instanceof SymbolArray){
if(at1.type == ((SymbolArray)s2).baseType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else{
if(at1.type == at2.type){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
```

```
}
}
}
catch(SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at1.name);
}
}
else if ((at1.name == "TRUE" || at1.name == "FALSE" || at1.name == "CONST_INT"  || at1.name == "CONS
(at2.name != "TRUE" && at2.name != "FALSE" && at2.name != "CONST_INT"  && at2.name != "CONST_CH
try{
Symbol s2 = st.getSymbol(at2.name);

if (s2 instanceof SymbolFunction){
if(at1.type == ((SymbolFunction)s2).returnType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else if (s2 instanceof SymbolArray){
if(at1.type == ((SymbolArray)s2).baseType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else{
if(at1.type == at2.type){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
}
catch(SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at2.name);
}
}
else if ((at1.name != "TRUE" && at1.name != "FALSE" && at1.name != "CONST_INT"  && at1.name != "CON
(!(at2.name != "TRUE" && at2.name != "FALSE" && at2.name != "CONST_INT"  && at2.name != "CONST_C
try{
Symbol s = st.getSymbol(at1.name);

if (s instanceof SymbolFunction){
if(at2.type == ((SymbolFunction)s).returnType){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else if (s instanceof SymbolArray){
if(at2.type == ((SymbolArray)s).baseType){
```

```
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
else{
if(at1.type == at2.type){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
}
catch(SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at1.name);
}

}

else{
if(at1.type == at2.type){
at.type = Symbol.Types.BOOL;
}
else {
at.type = Symbol.Types.UNDEFINED;
ErrorSemantico.deteccion("Tipos incompatibles ...");
}
}
at.isVar = false;
at.isVecComp = false;
at.isConst = true;
//System.out.println(ANSI_YELLOW + at.type + ANSI_RESET);

// ------------------------------------------------------------

//at.code.addBlock(at1.code);
at.code.addBlock(at2.code);
switch(operador.get(0)) {
case 0:
at.code.addInst(OpCode.EQ);
break;
case 1:
at.code.addInst(OpCode.LT);
break;
case 2:
at.code.addInst(OpCode.GT);
break;
case 3:
at.code.addInst(OpCode.LTE);
break;
case 4:
at.code.addInst(OpCode.GTE);
break;
case 5:
at.code.addInst(OpCode.NEQ);
break;
```

```
}
System.out.println(ANSI_YELLOW + at.code + ANSI_RESET);
}
)?
}



void operador_relacional(ArrayList<Integer> operador) :
{

}
{
<tEQU> {operador.add(0);}
| ■<tLT> {operador.add(1);}
| ■<tGT> {operador.add(2);}
| ■<tLE> {operador.add(3);}
| ■<tGE> {operador.add(4);}
| ■<tDIF> {operador.add(5);}
}


/* COMPLETA */
// En 'termino' sólo se comprueba si es int cuando se detecta que hay operaciones con mul, div o mod
// En el caso de que 'termino' solo sea un factor, no se comprueba el tipo, por eso se comprueba en
// esta función, porque podríamos recibir cualquier tipo de dato.
// Creo que se podría comprobar directamente en la función 'termino' que factor(at1) sea entero y así
// ya no tendríamos que comprobar nada en 'expresion_simple' pues siempre 'termino' será un int.
void expresion_simple(Attributes at) :
{
Attributes at1 = new Attributes(), at2 = new Attributes();
Integer operador = -1;
Integer operador2 =  -1;
}
{
( <tSUM> { operador2 = 0;} |■<tRES> { operador2 = 1;} )?
termino(at1) {
at.name = at1.name;
at.type = at1.type;
at.isConst = at1.isConst;
at.isVar = at1.isVar;
at.isVecComp = at1.isVecComp;

// ------------------------------------------------------------------------
at.code = at1.code;
switch(operador2) {
case 1:
at.code.addInst(OpCode.NGI); // Si nos llega un 5, entonces devolvemos un -5
break;
default: // Es un simbolo '+' o no se ha especificado un simbolo delante.
break;
}
}
( ( <tSUM> {operador = 0;} | <tRES> {operador = 1;} ) termino(at2) {

if((at1.name != "CONST_INT") && (at2.name != "CONST_INT")){
try{
Symbol s1 = st.getSymbol(at1.name);
Symbol s2 = st.getSymbol(at2.name);
if (s2 instanceof SymbolArray){
```

```
if(((SymbolArray)s2).baseType != Symbol.Types.INT){
// error: El primer factor no es un entero
//System.out.println(ANSI_YELLOW + ((SymbolArray)s2).baseType + ANSI_RESET);
ErrorSemantico.deteccion("El segundo termino no es un entero (tipos incompatibles - 1)");
}
if (s1 instanceof SymbolArray){
if(((SymbolArray)s1).baseType != ((SymbolArray)s2).baseType){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
else if (s1 instanceof SymbolFunction){
if(((SymbolFunction)s1).returnType != ((SymbolArray)s2).baseType){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
}
else if (s2 instanceof SymbolFunction){
if(((SymbolFunction)s2).returnType != Symbol.Types.INT){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El segundo termino no es un entero (tipos incompatibles - 2)");
}
if (s1 instanceof SymbolArray){
if(((SymbolArray)s1).baseType != ((SymbolFunction)s2).returnType){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
else if (s1 instanceof SymbolFunction){
if(((SymbolFunction)s1).returnType != ((SymbolFunction)s2).returnType){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
}
else {
if(at2.type != Symbol.Types.INT){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El segundo termino no es un entero (tipos incompatibles - 3)");
}
if (s1 instanceof SymbolArray){
if(((SymbolArray)s1).baseType != at2.type){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
else if (s1 instanceof SymbolFunction){
if(((SymbolFunction)s1).returnType != at2.type){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
}
}
catch(SymbolNotFoundException e) {
ErrorSemantico.deteccion(e, ("at1: " + at1.name + "; at2: " + at2.name));
}
}
else{
// Para el caso de tener alguna constante en la operación:
if(at1.name == "CONST_INT" && at2.name != "CONST_INT") {
try{
Symbol s2 = st.getSymbol(at2.name);
if (s2 instanceof SymbolArray){
if(((SymbolArray)s2).baseType != at1.type){
```

```
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
else if (s2 instanceof SymbolFunction){
if(((SymbolFunction)s2).returnType != at1.type){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
else if(at2.type != at1.type){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
catch (SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at2.name);
}
}
else if (at1.name != "CONST_INT" && at2.name == "CONST_INT"){
try{
Symbol s1 = st.getSymbol(at1.name);
if (s1 instanceof SymbolArray){
if(((SymbolArray)s1).baseType != at2.type){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
else if (s1 instanceof SymbolFunction){
if(((SymbolFunction)s1).returnType != at2.type){
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
else if(at2.type != at1.type){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El primer termino y el segundo son de distinto tipo");
}
}
catch (SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at1.name);
}
}
// El caso de que ambas son iguales ya nos indica que los dos son enteros
}

// ---------------------------------- GENERACION DE CODIGO ---------------------------------

//at.code.addBlock(at1.code);
at.code.addBlock(at2.code);
switch(operador) {
case 0:
at.code.addInst(OpCode.PLUS);
break;
case 1:
at.code.addInst(OpCode.SBT);
break;
}
}
)*
}


/* CREO QUE COMPLETADA */
```

```
void termino(Attributes at) :
{
Attributes at1 = new Attributes(), at2 = new Attributes();
ArrayList<Integer> operador = new ArrayList<Integer>();

}
{
factor(at1) {at.name = at1.name; at.type = at1.type; at.code = at1.code; at.isConst = at1.isConst; /*at = at1;*/
(
operador_multiplicativo(operador) {
// Aparece una operación de mul, div o mod, por lo tanto comprobamos
// que at1 sea entero.
if (at1.name != "CONST_INT"){
try{
Symbol s = st.getSymbol(at1.name);
if (s instanceof SymbolArray){
if(((SymbolArray)s).baseType != Symbol.Types.INT){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El primer factor no es un entero - Array");
}
}
else if (s instanceof SymbolFunction){
if(((SymbolFunction)s).returnType != Symbol.Types.INT){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El primer factor no es un entero - Function");
}
}
else {
if(at1.type != Symbol.Types.INT){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El primer factor no es un entero");
}
}
}
catch(SymbolNotFoundException e) {
ErrorSemantico.deteccion(e, at1.name);
}
}
}
factor(at2) {
if((at1.name != "CONST_INT") && (at2.name != "CONST_INT")){
try{
Symbol s1 = st.getSymbol(at1.name);
Symbol s2 = st.getSymbol(at2.name);
if (s2 instanceof SymbolArray){
if(((SymbolArray)s2).baseType != Symbol.Types.INT){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El segundo factor no es un entero (tipos incompatibles)");
}
if (s1 instanceof SymbolArray){
if(((SymbolArray)s1).baseType != ((SymbolArray)s2).baseType){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
else if (s1 instanceof SymbolFunction){
if(((SymbolFunction)s1).returnType != ((SymbolArray)s2).baseType){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
}
```

```
else if (s2 instanceof SymbolFunction){
if(((SymbolFunction)s2).returnType != Symbol.Types.INT){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El segundo factor no es un entero (tipos incompatibles)");
}
if (s1 instanceof SymbolArray){
if(((SymbolArray)s1).baseType != ((SymbolFunction)s2).returnType){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
else if (s1 instanceof SymbolFunction){
if(((SymbolFunction)s1).returnType != ((SymbolFunction)s2).returnType){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
}
else {
if(at2.type != Symbol.Types.INT){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El segundo factor no es un entero (tipos incompatibles)");
}
if (s1 instanceof SymbolArray){
if(((SymbolArray)s1).baseType != at2.type){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
else if (s1 instanceof SymbolFunction){
if(((SymbolFunction)s1).returnType != at2.type){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
}
}
catch(SymbolNotFoundException e) {
ErrorSemantico.deteccion(e, ("at1: " + at1.name + "; at2: " + at2.name));
}
}
else{
// Para el caso de tener alguna constante en la operación:
if(at1.name == "CONST_INT" && at2.name != "CONST_INT") {
try{
Symbol s2 = st.getSymbol(at2.name);
if (s2 instanceof SymbolArray){
if(((SymbolArray)s2).baseType != at1.type){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
else if (s2 instanceof SymbolFunction){
if(((SymbolFunction)s2).returnType != at1.type){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
else if(at2.type != at1.type){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
catch (SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at2.name);
}
```

```
}
else if (at1.name != "CONST_INT" && at2.name == "CONST_INT"){
try{
Symbol s1 = st.getSymbol(at1.name);
if (s1 instanceof SymbolArray){
if(((SymbolArray)s1).baseType != at2.type){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
else if (s1 instanceof SymbolFunction){
if(((SymbolFunction)s1).returnType != at2.type){
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
else if(at2.type != at1.type){
// error: El primer factor no es un entero
ErrorSemantico.deteccion("El primer factor y el segundo son de distinto tipo");
}
}
catch (SymbolNotFoundException e){
ErrorSemantico.deteccion(e, at1.name);
}
}
// El caso de que ambas son iguales ya nos indica que los dos son enteros
}

// ---------------------------------------------------------------------
// MUL = 0; MOD = 1; DIV = 2;

//System.out.println(ANSI_YELLOW + "at1.code: " + at1.code + ANSI_RESET);
at.code.addBlock(at2.code);
//System.out.println(ANSI_YELLOW + "at2.code: " + at2.code + ANSI_RESET);
switch(operador.get(0)) {
case 0:
at.code.addInst(OpCode.TMS);
break;
case 1:
at.code.addInst(OpCode.MOD);
break;
case 2:
at.code.addInst(OpCode.DIV);
break;
}
}
)*
}

void operador_multiplicativo(ArrayList<Integer> operador) :
{

}
{
<tMUL> {operador.add(0);} | <tMOD> {operador.add(1);} | <tDIV> {operador.add(2);}
}

void factor(Attributes at) :
{
}
{
primario(at) {
```

```
//System.out.println(ANSI_YELLOW + "at.code <factor>: " + at.code + ANSI_RESET );
}
|■<tNOT> primario(at)
{
// ---------------------- Semántico --------------------------
if (at.type != Symbol.Types.BOOL){
ErrorSemantico.deteccion("Debe ser booleano <factor>");
}
// ------------------------ Código ---------------------------
//at.code.addBlock(at.code);
at.code.addInst(OpCode.NGB);
}
}


void primario(Attributes at) :
{
Token t; // revisar
Token t1,t2,t3;
ArrayList<Attributes> ats = new ArrayList<Attributes>();
}
{
<tPARENTESIS_OPEN> expresion(at) <tPARENTESIS_CLOSE>

|■<tINT2CHAR> <tPARENTESIS_OPEN>
expresion(at)
{
if (at.type != Symbol.Types.INT) {
ErrorSemantico.deteccion("Se esperaba entero <int2char>");
}
else {
at.type = Symbol.Types.CHAR;
}
}
<tPARENTESIS_CLOSE>

|■<tCHAR2INT> <tPARENTESIS_OPEN>
expresion(at)
{
if (at.type != Symbol.Types.CHAR) {
ErrorSemantico.deteccion("Se esperaba caracter <char2int>");
}
else {
at.type = Symbol.Types.INT;
}
}
<tPARENTESIS_CLOSE>

|■LOOKAHEAD(2) t = <tID> <tPARENTESIS_OPEN> lista_una_o_mas_exps(ats) <tPARENTESIS_CLOSE
Symbol s = null;
try {
s = st.getSymbol(t.image);
if (s instanceof SymbolArray) {
// comprobar indice acceso de vector, habrá que obtener el rango del vector y comparar con el indice acced
// comprobar también que el indice sea un entero positivo?
at.isVecComp = true;
at.type = ((SymbolArray)s).baseType;

if(ats.size() == 0) {
ErrorSemantico.deteccion("No se ha especificado el indice del vector a acceder");
```

```
}
else{
// Si lo hay, sólo habrá 1 atributo en el array de atributos si se llama a una componente de vector
Attributes atA;
atA = ats.get(0);
if(atA.type != Symbol.Types.INT){
ErrorSemantico.deteccion("El indice de acceso al vector debe ser de tipo INT");
}
}
}
else if ((s instanceof SymbolProcedure)){

if(ats.size() != ((SymbolProcedure)s).parList.size()) {
//System.out.println(ANSI_YELLOW + ats.size() + ANSI_RESET);
//System.out.println(ANSI_YELLOW + ((SymbolProcedure)s).parList.size() + ANSI_RESET);
ErrorSemantico.deteccion("Numero incorrecto de parametros -");
}
else {
try {
for (int i = 0; i < ats.size(); i++) {
Attributes atP;
Symbol sP;
atP = ats.get(i);
sP = ((SymbolProcedure)s).parList.get(i);

// Evitamos llamar a getSymbol si los atributos son constantes escalares
if (atP.name != "TRUE" && atP.name != "FALSE" && atP.name != "CONST_INT"  && atP.name != "CONST_
Symbol symbol_atP = st.getSymbol(atP.name);
if(sP instanceof SymbolFunction){
if (symbol_atP instanceof SymbolFunction){
if (((SymbolFunction)symbol_atP).returnType != ((SymbolFunction)sP).returnType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + ((SymbolFunction)symbol_atP).retu
break;
}
}
else if (symbol_atP instanceof SymbolArray){
if (((SymbolArray)symbol_atP).baseType != ((SymbolFunction)sP).returnType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + ((SymbolArray)symbol_atP).baseT
break;
}
}
else{
if (atP.type != ((SymbolFunction)sP).returnType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + atP.type + "|" + ((SymbolFunction)s
break;
}
}
}
else if(sP instanceof SymbolArray){
if (symbol_atP instanceof SymbolFunction){
if (((SymbolFunction)symbol_atP).returnType != ((SymbolArray)sP).baseType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + ((SymbolFunction)symbol_atP).retu
break;
}
}
else if (symbol_atP instanceof SymbolArray){
if (((SymbolArray)symbol_atP).baseType != ((SymbolArray)sP).baseType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + ((SymbolArray)symbol_atP).baseT
break;
}
```

```
}
else {
if (atP.type != ((SymbolArray)sP).baseType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + atP.type + "|" + ((SymbolArray)sP).
break;
}
}

}
}
else{ // Casos en el que los atributos sean valores constantes (TRUE, FALSE, CONST_INT, ...)
if (atP.type != sP.type) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + atP.type + "|" + sP.type + "- Proced
break;
}
}
}
}
catch(SymbolNotFoundException e){
ErrorSemantico.deteccion(e, "Atributo atP");
}
}
}
else if (s instanceof SymbolFunction){

if(ats.size() != ((SymbolFunction)s).parList.size()) {
ErrorSemantico.deteccion("Numero incorrecto de parametros");
}
else {
try {
for (int i = 0; i < ats.size(); i++) {
Attributes atF;
Symbol sF;
atF = ats.get(i);
sF = ((SymbolFunction)s).parList.get(i);

// Evitamos llamar a getSymbol si los atributos son constantes escalares
if (atF.name != "TRUE" && atF.name != "FALSE" && atF.name != "CONST_INT"  && atF.name != "CONST_
Symbol symbol_atF = st.getSymbol(atF.name);
if(sF instanceof SymbolFunction){
if (symbol_atF instanceof SymbolFunction){
if (((SymbolFunction)symbol_atF).returnType != ((SymbolFunction)sF).returnType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + ((SymbolFunction)symbol_atF).retu
break;
}
}
else if (symbol_atF instanceof SymbolArray){
if (((SymbolArray)symbol_atF).baseType != ((SymbolFunction)sF).returnType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + ((SymbolArray)symbol_atF).baseT
break;
}
}
else{
if (atF.type != ((SymbolFunction)sF).returnType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + atF.type + "|" + ((SymbolFunction)s
break;
}
}
}
else if(sF instanceof SymbolArray){
```

```
if (symbol_atF instanceof SymbolFunction){
if (((SymbolFunction)symbol_atF).returnType != ((SymbolArray)sF).baseType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + ((SymbolFunction)symbol_atF).retu
break;
}
}
else if (symbol_atF instanceof SymbolArray){
if (((SymbolArray)symbol_atF).baseType != ((SymbolArray)sF).baseType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + ((SymbolArray)symbol_atF).baseT
break;
}
}
else {
if (atF.type != ((SymbolArray)sF).baseType) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + atF.type + "|" + ((SymbolArray)sF).
break;
}
}

}
}
else{ // Casos en el que los atributos sean valores constantes (TRUE, FALSE, CONST_INT, ...)
if (atF.type != sF.type) {
ErrorSemantico.deteccion("Los tipos de los parametros no coinciden: " + atF.type + "|" + sF.type + "- Functio
break;
}
}
}
}
catch(SymbolNotFoundException e){
ErrorSemantico.deteccion(e, "Atributo atF");
}
at.type = ((SymbolFunction)s).returnType;
}
}
else {
ErrorSemantico.deteccion("Se esperaba componente de vector, función o procedimiento ...");
}
}
catch (SymbolNotFoundException e) {
ErrorSemantico.deteccion(e, t.image);
}
at.isVar = false;
at.name = t.image;
// Procesar la lista de parametros reales ...
//...
}
|■t = <tID> {            //var. o func. sin pars
Symbol s = null;
try {
s = st.getSymbol(t.image);
at.isVar = true;
at.type = s.type;
at.name = t.image;
}
catch (SymbolNotFoundException e) {
ErrorSemantico.deteccion(e, t.image);
}
// ----------------------------------------
at.code.addInst(OpCode.SRF, (st.level - s.nivel), (int)s.dir);
```

```
at.code.addInst(OpCode.DRF);
//System.out.println(ANSI_YELLOW + s.name + ": " + (int)s.dir + ANSI_RESET);
}
|■t1 = <tCONST_INT> {
at.name = "CONST_INT";
at.isVar = false;
at.type = Symbol.Types.INT;
at.isConst = true;
// ------------------------------
at.code.addInst(OpCode.STC, Integer.valueOf(t1.image));
//System.out.println(ANSI_YELLOW + "STC" + ANSI_RESET);
}
|■t2 = <tCONST_CHAR> {
at.name = "CONST_CHAR";
at.isVar = false;
at.type = Symbol.Types.CHAR;
at.isConst = true;
// ------------------------------
at.code.addInst(OpCode.STC, ((int)(t2.image).charAt(1)) );
//System.out.println(ANSI_YELLOW + "STC2" + ANSI_RESET);
}
|■t3 = <tCONST_STRING> { //rn sf.primario_8(t);
at.name = "CONST_STRING";
at.isVar = false;
at.type = Symbol.Types.STRING;
at.isConst = true;
// ------------------------------
// Como se almacena un string?
}

|■<tTRUE> {
at.name = "TRUE";
at.isVar = false;
at.type = Symbol.Types.BOOL;
at.isConst = true;
// ------------------------------
at.code.addInst(OpCode.STC, 1);
//System.out.println(ANSI_YELLOW + "STC3" + ANSI_RESET);
}
|■<tFALSE> {
at.name = "FALSE";
at.isVar = false;
at.type = Symbol.Types.BOOL;
at.isConst = true;
// ------------------------------
at.code.addInst(OpCode.STC, 0);
//System.out.println(ANSI_YELLOW + "STC4" + ANSI_RESET);
}
}
```