Documentación práctica DLP

Fernando Palazuelo Ginzo

UO244588

Patrones del léxico

```
// ************  Patrones (macros) ********************
ConstanteEntera = [0-9]*
Identificador = [a-zA-ZñÑà-üÀ-Ü][a-zA-Z0-9ñÑà-üÀ-Ü]*
ConstanteReal = [0-9]+(\.[0-9]*)?((e|E)[-+]?[0-9]+)?
ConstanteCaracter = "'"(.|(\\[0-9]+)|(\\[nt]))"'"

//[0-9a-zA-ZñÑà-üÀ-Ü]



%%
// ************  Acciones ********************


//Palabras reservadas
read {this.yylval = yytext(); return Parser.READ;}
write {this.yylval = yytext(); return Parser.WRITE;}
while {this.yylval = yytext(); return Parser.WHILE;}
if {this.yylval = yytext(); return Parser.IF;}
else {this.yylval = yytext(); return Parser.ELSE;}
int {this.yylval = yytext(); return Parser.INT;}
float32 {this.yylval = yytext(); return Parser.FLOAT32;}
char {this.yylval = yytext(); return Parser.CHAR;}
var {this.yylval = yytext(); return Parser.VAR;}
struct {this.yylval = yytext(); return Parser.STRUCT;}
return {this.yylval = yytext(); return Parser.RETURN;}
func {this.yylval = yytext(); return Parser.FUNC;}
main {this.yylval = yytext(); return Parser.MAIN;}


//constante entera
{ConstanteEntera}  { this.yylval = yytext();
                          return Parser.CTE_ENTERA;  }

//constante real
{ConstanteReal} {this.yylval = yytext();
                               return Parser.CTE_REAL;}




 //Identificador
{Identificador}    {this.yylval = yytext();
                               return Parser.IDENT;}

//Constante caracter
{ConstanteCaracter} {this.yylval = yytext();
                                     return Parser.CTE_CARACTER;}
```

```
//Comentarios
"/*"~"*/" {}
"//".* {}

//operadores logicos
"=="   {this.yylval = yytext();return Parser.IGUAL_IGUAL;}
">="   {this.yylval = yytext();return Parser.MAYOR_IGUAL;}
"<="   {this.yylval = yytext();return Parser.MENOR_IGUAL;}
"!="   {this.yylval = yytext();return Parser.DISTINTO;}
"&&"   {this.yylval = yytext();return Parser.AND;}
"||"   {this.yylval = yytext();return Parser.OR;}



//Operadores simples

"+"|
"-"|
"<"|
">"|
"*"|
"%"|
"!"|
")"|
"("|
";"|
","|
"."|
"["|
"]"|
"/"|
"{"|
"}"|
"="          {this.yylval = yytext();return yytext().charAt(0);}



[ \n\t\r] {}

.     {error(yytext());}
```

Gramática

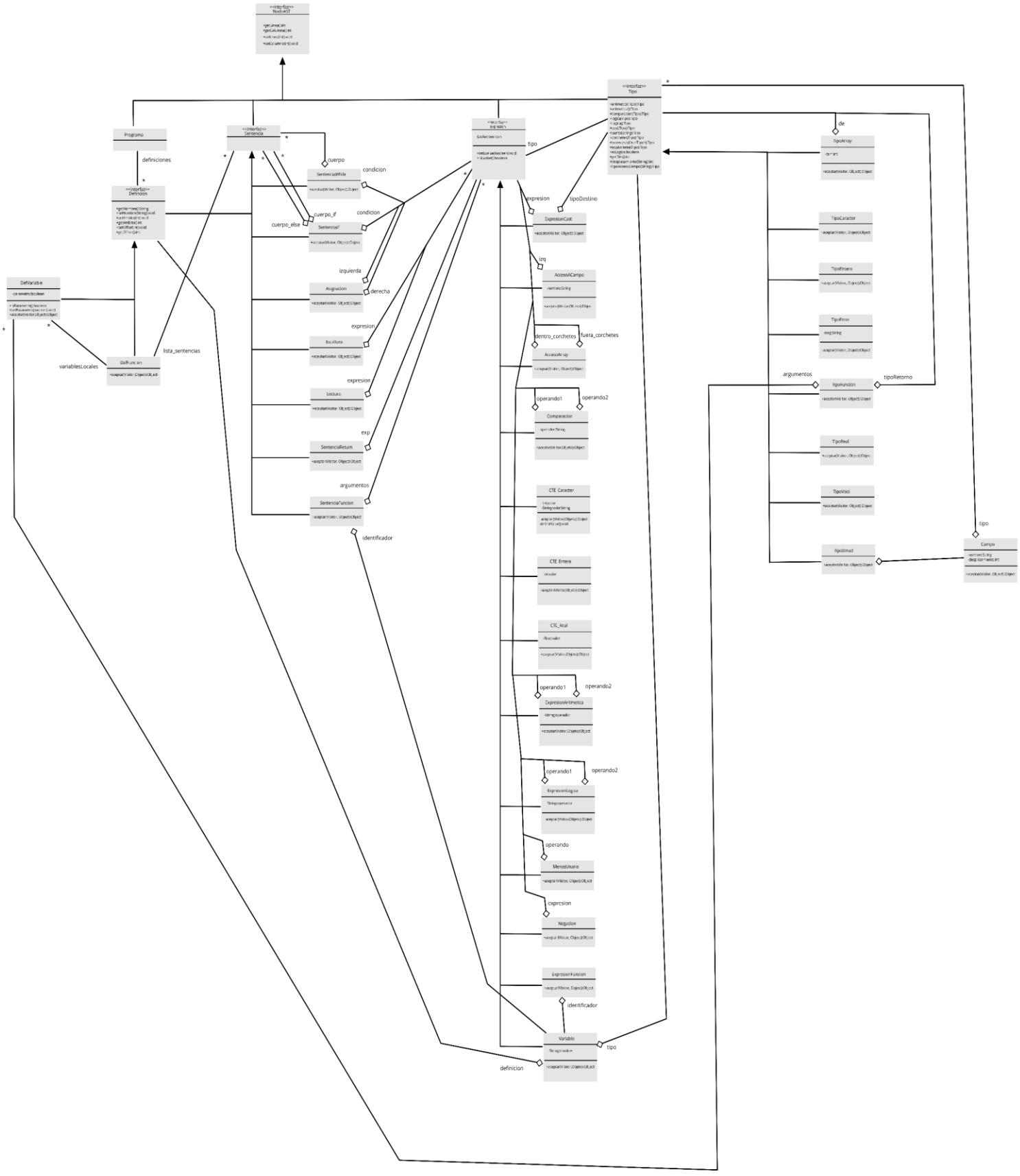
```
programa: lista_definiciones FUNC MAIN '(' ')' '{' lista_definicion_variables
lista_sentencias '}'
             ;
```

```
lista_definiciones: lista_definiciones definicion_variable
             | lista_definiciones definicion_funcion
             |
             ;


lista_definicion_variables: lista_definicion_variables definicion_variable
             |
             ;

definicion_variable: VAR variable ';'
             ;

variable: identificadores tipo_ampliado
             ;

tipo_ampliado: tipo
             | tipo_vector
             | tipo_struct
             ;

tipo: INT
      | FLOAT32
      | CHAR
      ;

tipo_vector: '[' CTE_ENTERA ']' tipo_ampliado
             ;


tipo_struct: STRUCT '{' registros_struct '}'
             ;

registros_struct: registros_struct identificadores tipo_ampliado ';'
             |
             ;


identificadores: IDENT
             | identificadores ',' IDENT
             ;

definicion_funcion: FUNC IDENT '(' lista_parametros_opt ')' tipo_retorno '{'
lista_definicion_variables lista_sentencias '}'
             ;

tipo_retorno: tipo
             |
             ;

lista_parametros_opt: lista_parametros
             |
             ;

lista_parametros: lista_parametros ',' parametro
             | parametro
             ;

parametro: IDENT tipo
```

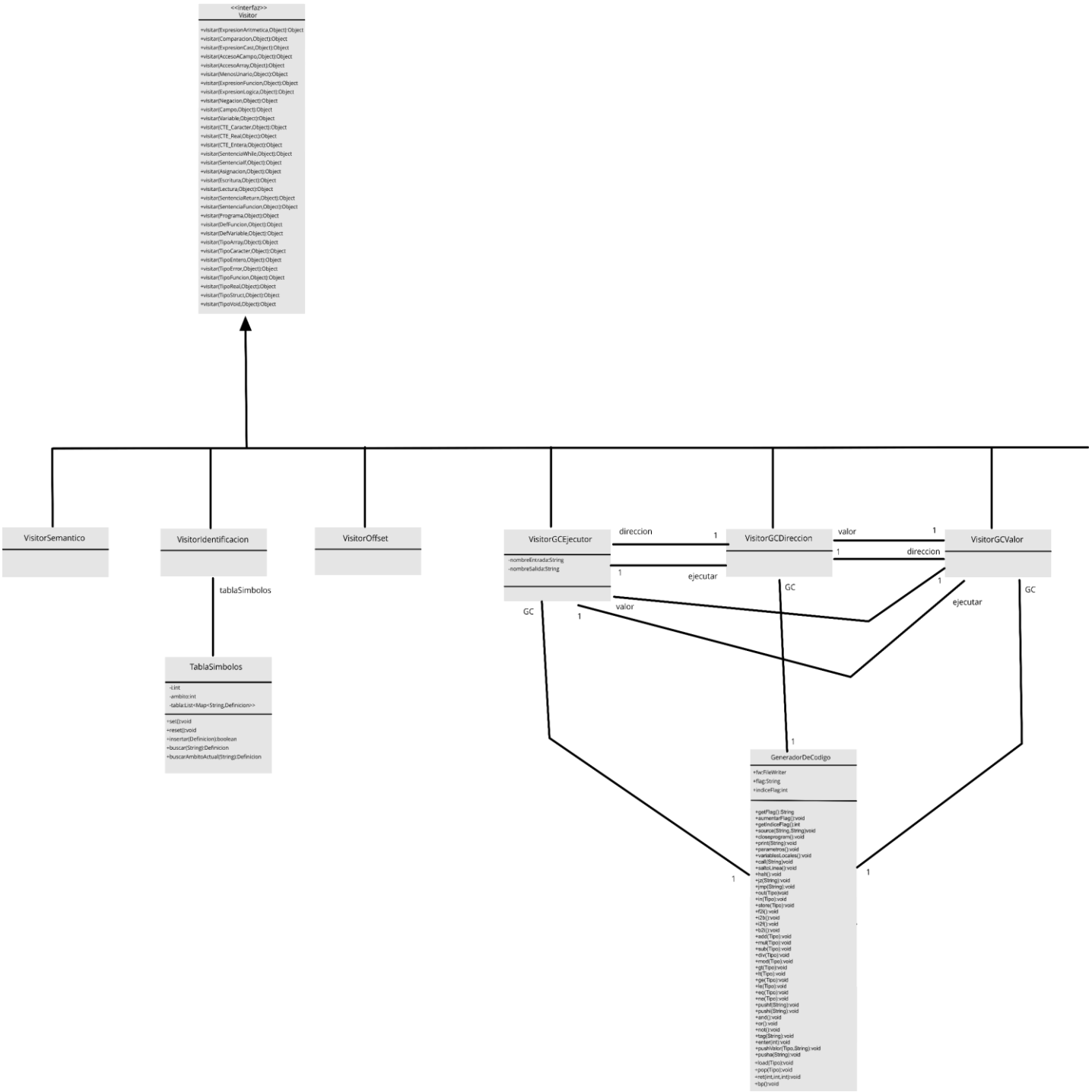
```
            ;

lista_sentencias: lista_sentencias sentencia
             |
             ;

sentencia:   expresion '=' expresion ';'
             | IF expresion '{' lista_sentencias '}'
             | IF expresion '{' lista_sentencias '}' ELSE '{'
lista_sentencias '}'
             | WHILE expresion '{' lista_sentencias '}'
             | IDENT '(' lista_expresiones_opt ')' ';'
             | WRITE '(' lista_expresiones ')' ';'
             | READ '(' lista_expresiones ')' ';'
             | RETURN expresion ';'
             ;

cast: tipo '(' expresion ')'
      ;

lista_expresiones_opt: lista_expresiones
             |
             ;

lista_expresiones: lista_expresiones ',' expresion
             | expresion

             ;

expresion: expresion '+' expresion
         | expresion '*' expresion
         | expresion '/' expresion
         | IDENT '(' lista_expresiones_opt ')'
         | expresion '%' expresion
         | expresion '-' expresion
         | cast
         | '-' expresion %prec MENOS_UNARIO
         | '(' expresion ')'
         | expresion '[' expresion ']'
         | expresion AND expresion
         | expresion OR expresion
         | expresion IGUAL_IGUAL expresion
         | expresion MAYOR_IGUAL expresion
         | expresion MENOR_IGUAL expresion
         | expresion DISTINTO expresion
         | expresion '>' expresion
         | expresion '<' expresion
         | '!' expresion
         | expresion '.' IDENT
         | CTE_ENTERA
         | CTE_REAL
         | CTE_CARACTER
         | IDENT
        ;
```

UML de las clases

AST + tipos



Visitor


<<interfaz>>
Visitor
+visitar(ExpresionAritmetica,Object):Object
+visitar(Comparacion,Object):Object
+visitar(ExpresionCast,Object):Object
+visitar(AccesoACampo,Object):Object
+visitar(AccesoArray,Object):Object
+visitar(MenosUnario,Object):Object
+visitar(ExpresionFuncion,Object):Object
+visitar(ExpresionLogica,Object):Object
+visitar(Negacion,Object):Object
+visitar(Campo,Object):Object
+visitar(Variable,Object):Object
+visitar(CTE_Caracter,Object):Object
+visitar(CTE_Real,Object):Object
+visitar(CTE_Entera,Object):Object
+visitar(SentenciaWhile,Object):Object
+visitar(SentenciaIf,Object):Object
+visitar(Asignacion,Object):Object
+visitar(Escritura,Object):Object
+visitar(Lectura,Object):Object
+visitar(SentenciaReturn,Object):Object
+visitar(SentenciaFuncion,Object):Object
+visitar(Programa,Object):Object
+visitar(DefFuncion,Object):Object
+visitar(DefVariable,Object):Object
+visitar(TipoArray,Object):Object
+visitar(TipoCaracter,Object):Object
+visitar(TipoEntero,Object):Object
+visitar(TipoError,Object):Object
+visitar(TipoFuncion,Object):Object
+visitar(TipoReal,Object):Object
+visitar(TipoStruct,Object):Object
+visitar(TipoVoid,Object):Object
VisitorSemantico
VisitorIdentificacion
VisitorOffset
VisitorGCEjecutor
-nombreEntrada:String
-nombreSalida:String
VisitorGCDireccion
VisitorGCValor
direccion
1
ejecutar
valor
GC
tablaSimbolos
TablaSimbolos
-cint
-ambito:int
-tabla:List<Map<String,Definicion>>
+set():void
+reset():void
+insertar(Definicion):boolean
+buscar(String):Definicion
+buscarAmbitoActual(String):Definicion
GeneradorDeCodigo
+fwcFileWriter
+flag:String
+indiceFlag:int
+getFlag():String
+aumentarFlag():void
+getIndiceFlag():int
+source(String,String):void
+closeprograma():void
+print(String):void
+parametros():void
+variablesLocales():void
+call(String):void
+saltoLinea():void
+halt():void
+jz(String):void
+jmp(String):void
+out(Tipo):void
+in(Tipo):void
+store(Tipo):void
+f2i():void
+c2i():void
+i2f():void
+b2i():void
+add(Tipo):void
+mul(Tipo):void
+sub(Tipo):void
+div(Tipo):void
+mod(Tipo):void
+gt(Tipo):void
+lt(Tipo):void
+ge(Tipo):void
+le(Tipo):void
+eq(Tipo):void
+ne(Tipo):void
+pushf(String):void
+pushi(String):void
+and():void
+or():void
+not():void
+tag(String):void
+enter(int):void
+pushVelor(Tipo,String):void
+pushla(String):void
+load(Tipo):void
+pop(Tipo):void
+ret(int,int,int):void
+bp():void

Plantillas de generación de código

```
ejecutar[[Programa: programa -> definicion*]] =
	<source> programa.file_name

	for(Definicion def definicion*)
		if def instanceof DefVariable
			ejecutar[[def]]

	<call> main
	<halt>


	for(Definicion def definicion*)
		if def instanceof DefFuncion
			ejecutar[[def]]


ejecutar[[Escritura: sentencia -> expresion*]](defFuncion) =

	for(Expresion exp: expresion*)
		valor[[exp]]
		<out>


ejecutar[[Lectura: sentencia -> expresion*]](defFuncion) =

	for(Expresion exp: expresion*)
		direccion[[expresion]]
		<in>
		<store>


ejecutar[[Asignacion: sentencia -> expresion_1 expresion_2]](defFuncion) =


	direccion[[expresion_1]]
	valor[[expresion_2]]
	<store>


valor[[Cast: expresion -> tipo expresionC]] =

		valor[[expresionC]]
		if(expresionC.tipo instanceof TipoReal){
			if(tipo instanceof TipoEntero){
				<f2i>
			}else if(tipo instanceof TipoCaracter){
				<f2i>
				<i2b>
			}

		}else if(expresionC.tipo instanceof TipoEntero){
			if(tipo instanceof TipoReal){
				<i2f>
			}else if(tipo instanceof TipoCaracter){
				<i2b>
```

```
                }
        }else{
                if(tipo instanceof TipoEntero){
                        <b2i>
                }else if(tipo instanceof TipoReal){
                        <b2i>
                        <i2f>
                }
        }


valor[[ExpresionAritmetica: expresion -> expresion_1 operador expresion_2]] =

        valor[[expresion_1]]
        if(expresion_1.tipo instanceof TipoCaracter){
                <b2i>
        }
        valor[[expresion_2]]
        if(expresion_2.tipo instanceof TipoCaracter){
                <b2i>
        }
        if(operador.equals("+")){
                if(expresion_1.tipo instanceof TipoReal){
                        <addf>
                }else{
                        <addi>
                }
        }else if(operador.equals("*")){
                if(expresion_1.tipo instanceof TipoReal){
                        <mulf>
                }else{
                        <muli>
                }
        }else if(operador.equals("-")){
                if(expresion_1.tipo instanceof TipoReal){
                        <subf>
                }else{
                        <subi>
                }
        }else if(operador.equals("/")){
                if(expresion_1.tipo instanceof TipoReal){
                        <divf>
                }else{
                        <divi>
                }
        }else{
                if(expresion_1.tipo instanceof TipoReal){
                        <modf>
                }else{
                        <modi>
                }
        }


valor[[Comparacion: expresion -> expresion_1 operando expresion_2]] =

        valor[[expresion_1]]
        if(expresion_1.tipo instanceof TipoCaracter){
                <b2i>
```

```
	}
	valor[[expresion_2]]
	if(expresion_2.tipo instanceof TipoCaracter){
		<b2i>
	}
	if(operando.equals(">")){
		if(expresion_1.tipo instanceof TipoReal){
			<gtf>
		}else{
			<gti>
		}
	}else if(operando.equals("<")){
		if(expresion_1.tipo instanceof TipoReal){
			<ltf>
		}else{
			<lti>
		}
	}else if(operando.equals(">=")){
		if(expresion_1.tipo instanceof TipoReal){
			<gef>
		}else{
			<gei>
		}
	}else if(operando.equals("<=")){
		if(expresion_1.tipo instanceof TipoReal){
			<lef>
		}else{
			<lei>
		}
	}else if(operando.equals("==")){
		if(expresion_1.tipo instanceof TipoReal){
			<eqf>
		}else{
			<eqi>
		}
	}else{
		if(expresion_1.tipo instanceof TipoReal){
			<nef>
		}else{
			<nei>
		}
	}


valor[[MenosUnario expresion -> expresionM]] =

	valor[[expresionM]]
	if(expresionM.tipo instanceof TipoCaracter){
		<b2i>
	}
	if(expresionM.tipo instanceof TipoReal){
		<pushf> -1.0
		<mulf>
	}else{
		<pushi> -1
		<muli>
	}

valor[[ExpresionLogica expresion -> expresion_1 operador expresion_2]] =
```

```
	valor[[expresion_1]]
	valor[[expresion_2]]
	if(operador.equal("&&")){
		<and>
	}else{
		<or>
	}


valor[[Negacion expresion -> expresionN]] =
	valor[[expresionN]]
	<not>


ejecutar[[DefFuncion definicion -> nombre tipoBase ambito offset defVariable*
sentencia*]] =

	nombre :
	' * Parameters
	for(DefVariable def : tipoBase.argumentos){
		ejecutar[[def]]
	}
	int tamEnter = 0;
	' * Local variables
	for(DefVariable def : defVAriable*){
		ejecutar[[def]]
		tamEnter+= def.offset;
	}
	<enter> tamEnter
	for(Sentencia sent: sentencia*){
		ejecutar[[sent]](DefFuncion)
	}


ejecutar[[DefVariable definicion -> nombre tipoBase ambito offsetVar
parametro]] =


	' * var nombre ejecutar[[tipoBase]] ( offset  offsetVar)


ejecutar[[Tipo tipoEntero]]	=

	int

ejecutar[[Tipo tipoReal]] =

	float32

ejecutar[[Tipo tipoCaracter]]	=

	char

ejecutar[[Tipo tipoStruct -> campo*]]  =

	struct{

	for(Campo cmp: campo*){
```

```
            ejecutar[[cmp]]
      }

      }

ejecutar[[Tipo tipoArray -> tam tipo_de]] =

      [tam] ejecutar[[tipo_de]]


ejecutar[[Campo -> tipo nombre]] =

      nombre ejecutar[[tipo]]

ejecutar[[SenteciaIf sentencia -> exp_condicion sentencia_if*
sentencia_else*]](defFuncion) =

      valor[[exp_condicion]]
      jz {cuerpo_else}
      for(Sentencia sent : sentencia_if*){
            ejecutar[[sent]]
      }
      jmp {fin_if}
      {cuerpo_else}:
      for(Sentencia sent : sentencia_else*){
            ejecutar[[sent]]
      }


      {fin_if}:

ejecutar[[SentenciaWhile sentencia -> expCondicion
sentencia_cuerpo*]](defFuncion) =

      {condicion_while}:
      valor[[expCondicion]]
      jz {fin_while}
      for(Sentencia sent: sentencia_cuerpo*){
            ejecutar[[sent]]
      }
      jmp {condicion_while}
      {fin_while}:

valor[[AccesoACampo expresion -> expresion_izq nombre_string]] =

      dir[[expresion_izq]]
      <push> expresion_izq.getTipo().desplazamiento(nombre_string)
      <addi>
      <loadi>

direccion[[AccesoACampo expresion -> expresion_izq nombre_string]] =

      dir[[expresion_izq]]
      <push> expresion_izq.getTipo().desplazamiento(nombre_string)
      <addi>


direccion[[AccesoArray expresion -> expr_fueraCorchetes
expr_dentroCorchetes]] =
```

```
	dir[[expr_fueraCorchetes]]
	valor[[expr_dentroCorchetes]]
	<pushi> expresion_izq.getTipo().getBits();
	<muli>
	<addi>


valor[[AccesoArray expresion -> expr_fueraCorchetes expr_dentroCorchetes]] =


	dir[[expr_fueraCorchetes]]
	valor[[expr_dentroCorchetes]]
	<pushi> expr_fueraCorchetes.getTipo().getBits();
	<muli>
	<addi>
	<load>




ejecutar[[SentenciaReturn sentencia -> expresion]](defFuncion) =
	<valor> expresion
	<ret>
defFuncion.getTipoRetorno().getBits(),defFuncion.tamVariablesLocales() ,
defFuncion.getTipo().getTamParametros()


ejecutar[[SentenciaFuncion sentencia -> var_identificador
exp_parametro*]](defFuncion) =

	for(Expresion exp: exp_parametro*){
		valor[[exp*]]
	}
	<call> var_identificador.nombre
	if(var_identificador.getTipo().getTipoRetorno() != tipoVoid){
		<pop>
	}

valor[[ExpresionFuncion expresion -> var_identificador exp_parametro* ]] =

	for(Expresion exp: exp_parametro*){
		valor[[exp*]]

	}
	<call> var_identificador.nombre
```