Universidad de Oviedo

DOCUMENTACIÓN DLP PRÁCTICAS

Carlos Sanabria Miranda UO250707

Tabla de contenido

Patrones léxicos

1. Patrones

```
// Saltables y comentarios

Saltables = [ \n\t\r]+

ComentarioUnaLinea = "#" .* (\n)?

ComentarioVariasLineas = \"\"\" ~ \"\"\"

// Enteros

ConstanteEntera = [0-9]+

// Reales

ConstanteRealIzda = {ConstanteEntera}"."

ConstanteRealDcha = "."{ConstanteEntera}

ConstanteRealAmbos = {ConstanteEntera}"."{ConstanteEntera}

ConstanteRealPorPunto = {ConstanteRealIzda} | {ConstanteRealDcha} | {ConstanteRealAmbos}

ConstanteEnteraORealPorPunto = {ConstanteEntera} | {ConstanteRealPorPunto}

ConstanteRealPorExponente = {ConstanteEnteraORealPorPunto}[eE][-+]?{ConstanteEntera}

ConstanteReal = {ConstanteRealPorPunto} | {ConstanteRealPorExponente}

// Caracteres

CaracteresEspeciales = '\\n'|'\\t'

ConstanteCaracterNormal = '.'|{CaracteresEspeciales}

ConstanteCaracterASCII = '\\[0-9]+'

// Identificadores

Identificador = [a-zA-ZñÑáéíóúÁÉÍÓÚ_][a-zA-ZñÑáéíóúÁÉÍÓÚ_0-9]*

// Operadores de más de un carácter

IgualIgual = "=="

MenorOIgual = "<="

MayorOIgual = ">="

Distinto = "!="

And = "&&"

Or = "||"

MasIgual        = "+="

MenosIgual      = "-="

PorIgual        = "*="
```

```
DivIgual        = "/="
ModIgual        = "%="
MasMas          = "++"
MenosMenos      = "--"
Xor             = "^^"
```

2. Acciones

```
// * Saltables y comentarios
{Saltables}                     { }
{ComentarioUnaLinea}            { }
{ComentarioVariasLineas}        { }
// * Palabras reservadas
"input"                         { return Parser.INPUT; }
"print"                         { return Parser.PRINT; }
"def"                           { return Parser.DEF; }
"while"                         { return Parser.WHILE; }
"if"                            { return Parser.IF; }
"else"                          { return Parser.ELSE; }
"int"                           { return Parser.INT; }
"double"                        { return Parser.DOUBLE; }
"char"                          { return Parser.CHAR; }
"struct"                        { return Parser.STRUCT; }
"return"                        { return Parser.RETURN; }
"void"                          { return Parser.VOID; }
"main"                          { return Parser.MAIN; }
"do"                            { return Parser.DO; }
"for"                           { return Parser.FOR; }
"break"                         { return Parser.BREAK; }
// * Operadores, paréntesis, corchetes, llaves, coma, punto, punto y coma, dos puntos, interrogacion
[+\-/*%><=()!\[\]{},;.:?]       { this.yylval = yytext(); return yytext().charAt(0); }
{IgualIgual}                    { this.yylval = yytext(); return Parser.EQUALS; }
{MenorOIgual}                   { this.yylval = yytext(); return Parser.LESS_OR_EQUAL; }
{MayorOIgual}                   { this.yylval = yytext(); return Parser.GREATER_OR_EQUAL; }
```

```
{Distinto}                          { this.yylval = yytext(); return Parser.DISTINCT; }
{And}                               { this.yylval = yytext(); return Parser.AND; }
{Or}                                { this.yylval = yytext(); return Parser.OR; }
{Xor}                               { this.yylval = yytext(); return Parser.XOR; }
// * +=, -=, *=, /= y %=
{MasIgual}          { this.yylval = yytext(); return Parser.PLUS_EQUAL; }
{MenosIgual}        { this.yylval = yytext(); return Parser.MINUS_EQUAL; }
{PorIgual}          { this.yylval = yytext(); return Parser.MUL_EQUAL; }
{DivIgual}          { this.yylval = yytext(); return Parser.DIV_EQUAL; }
{ModIgual}          { this.yylval = yytext(); return Parser.MOD_EQUAL; }
// ++ y --
{MasMas}            { this.yylval = yytext(); return Parser.PLUS_PLUS; }
{MenosMenos}        { this.yylval = yytext(); return Parser.MINUS_MINUS; }
// * Constante Entera
{ConstanteEntera}                   { this.yylval = new Integer(yytext());
                                      return Parser.INT_CONSTANT;  }
// * Constante Real
{ConstanteReal}                     { this.yylval = new Double(yytext());
                                      return Parser.REAL_CONSTANT; }
// * Constantes Caracter
{ConstanteCaracterNormal}           { String s = yytext();
                                      if(s.equals("'\\n'")) this.yylval = '\n';
                                      else if(s.equals("'\\t'")) this.yylval = '\t';
                                      else this.yylval = yytext().charAt(1);
                                      return Parser.CHAR_CONSTANT; }

{ConstanteCaracterASCII}            { String s = yytext();
                                      String ascii_string = s.substring(2,s.length()-1);
                                      Character c = (char) Integer.parseInt(ascii_string);
                                      this.yylval = c;
                                      return Parser.CHAR_CONSTANT; }
```

```
// * Identificador
{Identificador}                 { this.yylval = yytext(); return Parser.ID; }
// * Cualquier otro carácter
.                               { new ErrorType(this.getLine(), this.getColumn(),
                                "Lexical error: Unknow character \'"+ yycharat(0)+"\'."); }
```

Gramática libre de contexto

program: _program main

_program: λ

| _program variable_definition

| _program function_definition

main: DEF MAIN '(' ')' ':' VOID '{' function_body '}'

// Definición de variables

variable_definition: variables ':' type ';'

variables: ID

| variables ',' ID

// Definición de funciones

function_definition: DEF ID '(' parameters ')' ':' return_type '{' function_body '}'

parameters: λ

| _parameters

_parameters: parameter

| _parameters ',' parameter

parameter: ID ':' simple_type

return_type: simple_type

| VOID

function_body: λ

| variable_definitions statements

| variable_definitions

| statements

variable_definitions: variable_definition

| variable_definitions variable_definition

statements: statement

| statements statement

// Tipos de sentencias

statement: statement_without_semicolon ';'
| while
| for
| if

statement_without_semicolon: assignment_as_expression
| function_call_as_expression
| return
| read
| write
| plus_equal
| minus_equal
| mul_equal
| div_equal
| mod_equal
| pre_plus_plus_as_expression
| pre_minus_minus_as_expression
| post_plus_plus_as_expression
| post_minus_minus_as_expression
| do_while
| break

// Tipos

type: simple_type
| array
| struct

simple_type: INT
| DOUBLE
| CHAR

array: '[' INT_CONSTANT ']' type

struct: STRUCT '{' struct_body '}'

struct_body: variable_definition
| struct_body variable_definition

// ########### Sentencias (Statements) ###########

read: INPUT expressions %prec MENOR_QUE_COMA

write: PRINT expressions %prec MENOR_QUE_COMA

| PRINTLN expressions %prec MENOR_QUE_COMA

expressions: expression

| expressions ',' expression

while: WHILE expression ':' '{' statements '}'

| WHILE expression ':' statement

do_while: DO ':' '{' statements '}' WHILE expression

| DO ':' statement WHILE expression

for: FOR '(' for_parenthesis_body ')' ':' '{' statements '}'

| FOR '(' for_parenthesis_body ')' ':' statement

for_parenthesis_body: statement_without_semicolon_1mcs ';' expression ';' statement_without_semicolon_1mcs

statement_without_semicolon_1mcs: statement_without_semicolon

| statement_without_semicolon_1mcs ',' statement_without_semicolon

if: IF expression ':' '{' statements '}' %prec MENOR_QUE_ELSE

| IF expression ':' '{' statements '}' ELSE '{' statements '}'

| IF expression ':' '{' statements '}' ELSE statement

| IF expression ':' statement %prec MENOR_QUE_ELSE

| IF expression ':' statement ELSE '{' statements '}'

| IF expression ':' statement ELSE statement

return: RETURN expression

plusEqual: expression PLUS_EQUAL expression

minusEqual: expression MINUS_EQUAL expression

mulEqual: expression MUL_EQUAL expression

divEqual: expression DIV_EQUAL expression

modEqual: expression MOD_EQUAL expression

break: BREAK

// ########### Expresiones (Expressions) ###########

expression: expression AND expression

| expression OR expression

| xor_expression

| expression '>' expression

| expression GREATER_OR_EQUAL expression

| expression '<' expression

| expression LESS_OR_EQUAL expression

| expression DISTINCT expression

| expression EQUALS expression

| expression '+' expression

| expression '-' expression

| expression '*' expression

| expression '/' expression

| expression '%' expression

| '!' expression

| '-' expression

| '(' simple_type ')'

| expression '.' ID

| expression '[' expression ']'

| '(' expression ')'

| function_call_as_expression

| INT_CONSTANT

| REAL_CONSTANT

| CHAR_CONSTANT

| ID

| pre_plus_plus_as_expression

| pre_minus_minus_as_expression

| post_plus_plus_as_expression

| post_minus_minus_as_expression

| ternaryOperator

| assignment_as_expression

function_call_as_expression: ID '(' parameters_in_function_call ')'

parameters_in_function_call: λ

| expressions

pre_plus_plus_as_expression: PLUS_PLUS expression %prec PRE_ARITHMETIC

pre_minus_minus_as_expression: MINUS_MINUS expression %prec PRE_ARITHMETIC

post_plus_plus_as_expression: expression PLUS_PLUS

post_minus_minus_as_expression: expression MINUS_MINUS

ternaryOperator: expression '?' expression ':' expression %prec TERNARY_OPERATOR

assignment_as_expression: expression '=' expression

xor_expression: expression XOR expression

Gramática abstracta

program → definition*

funDefinition:definition → name:String type statement*

varDefinition:definition, statement → name:String type

functionType:type → param:varDefinition* returnType:type

arrayType:type → size:int of:type

recordType:type → fields:recordField*

realType:type →

intType:type →

charType:type →

voidType:type →

errorType:type → message:String

recordField → name:String type

ifStatement:statement → condition:expression ifBody:statement* elseBody:statement*

while:statement → condition:expression body:statement*

doWhile:statement → condition:expression body:statement*

for:statement → initializationStatements:statement* condition:expression incrementStatements:statement* body:statement*

read:statement → expression

write:statement → expression

return:statement → expression

break:statement →

arithmetic:expression → leftOp:expression operator:String rightOp:expression

logical:expression → leftOp:expression operator:String rightOp:expression

comparison:expression → leftOp:expression operator:String rightOp:expression

cast:expression → castType:type expression

charLiteral:expression → value:char

intLiteral:expression → value:int

realLiteral:expression → value:double

fieldAccess:expression → leftOp:expression name:String

indexing:expression → leftOp:expression rightOp:expression

unaryMinus:expression → expression

unaryNot:expression → expression

variable:expression → name:String

ternaryOperator:expression → condition:expression trueExpression:expression falseExpression:expression

invocation:statement, expression → function:variable arguments:expression*

preArithmetic:statement, expression → expression operator:String

postArithmetic:statement, expression → expression operator:String

assignment:statement, expression → left:expression right:expression

Gramáticas atribuidas

1. Fase de Identificación

Tabla I: Tabla de Atributos

Elemento	Atributo	Dominio	Tipo
VarDefinition	scope	Int	sintetizado
Variable	definition	Definition	sintetizado
Variable	definition	Definition	sintetizado

Conjuntos auxiliares: SymbolTable symbolTable

Tabla II: Gramática atribuida

Gramática Abstracta	Predicados	Reglas semánticas

funDefinition:definition → name:String type statement*	symbolTable.insert(funDefinition)	{
		symbolTable.set();
		visit(type);
		visit(statements$_i$)
		symbolTable.reset();
		}

varDefinition:definition, statement → name:String type	symbolTable.insert(varDefinition)	
variable:expression → name:String	symbolTable.find(name) != null	variable.definition = symbolTable.find(name)

2. Fase de Comprobación de Tipos

Tabla III: Tabla de Atributos

Elemento	Atributo	Dominio	Tipo
Expression	type	Type	sintetizado
Expression	lValue	boolean	sintetizado
Statement	assignsValue	boolean	sintetizado
Statement	isntLoopOrSwitchAndHasBreak	boolean	sintetizado

Tabla IV: Gramática atribuida

Gramática Abstracta	Predicados	Reglas semánticas

funDefinition:definition → name:String type statement*	type.returnType == VoidType ||
	∃ (visit(statements$_i$) == true)
	
	! ∃ (statements$_i$.
	isntLoopOrSwitchAndHasBreak)
		{
		visit(type);
		visit(statements$_i$,
		type.returnType)
		}

varDefinition:definition, statement → name:String type		varDefinition.assignsValue = false varDefinition. isntLoopOrSwitchAndHasBreak = false return false;

assignment:statement, expression → left:expression right:expression	left.lValue == true right.type.promotesTo(left.type) != null	assignment.lValue = false assignment.assignsValue = true assignment.type = right.type.promotesTo(left.type) assignment. isntLoopOrSwitchAndHasBreak = false return false;
ifStatement:statement → condition:expression ifBody:statement* elseBody:statement*	condition.type.isLogical()	ifStatement.assignsValue = false return ∃ (visit($ifBody_i$) == true) && ∃ (visit($elseBody_i$) == true) ifStatement. isntLoopOrSwitchAndHasBreak = ∃ ($ifBody_i$. isntLoopOrSwitchAndHasBreak) || ∃ ($elseBody_i$. isntLoopOrSwitchAndHasBreak)
read:statement → expression	expression.lValue == true	read.assignsValue = true read. isntLoopOrSwitchAndHasBreak = false return false;
return:statement → expression	functionReturnType != VoidType expression.type.promotesTo(functionReturnType) != null	return.assignsValue = false expression.type = expression.type.promotesTo(functionReturnType) return. isntLoopOrSwitchAndHasBreak = false return true;
while:statement → condition:expression body:statement*	condition.type.isLogical()	while.assignsValue = false while. isntLoopOrSwitchAndHasBreak = false return false;
doWhile:statement → condition:expression body:statement*	condition.type.isLogical()	doWhile.assignsValue = false doWhile. isntLoopOrSwitchAndHasBreak = false return false;

write:statement → expression	expression.type.isBuiltIn()	write.assignsValue = false write. isntLoopOrSwitchAndHasBreak = false return false;
for:statement → initializationStatements: statement* condition:expression incrementStatements: statement* body:statement*	condition.type.isLogical() $initializationStatements_i$.assignsValue $incrementStatements_i$.assignsValue	for.assignsValue = false for. isntLoopOrSwitchAndHasBreak = false return false;
break:statement →		break.assignsValue = false return false; break. isntLoopOrSwitchAndHasBreak = true
arithmetic:expression → leftOp:expression operator:String rightOp:expression	leftOp.type.arithmetic(rightOp.type) != null	arithmetic.lValue = false arithmetic.type = leftOp.type.arithmetic(rightOp.type)
cast:expression → castType:type expression	expression.type.canBeCast(castType) != null	cast.lValue = false cast.type = expression.type.canBeCast(castType)
charLiteral:expression → value:char		charLiteral.lValue = false charLiteral.type = CharType
intLiteral:expression → value:int		intLiteral.lValue = false intLiteral.type = IntType
realLiteral:expression → value:double		realLiteral.lValue = false realLiteral.type = RealType
comparison:expression → leftOp:expression operator:String rightOp:expression	leftOp.type.comparison(rightOp.type) != null	comparison.lValue = false comparison.type = leftOp.type.comparison(rightOp.type)
fieldAccess:expression → leftOp:expression name:String	leftOp.type.dot(name) != null	fieldAccess.lValue = true fieldAccess.type = leftOp.type.dot(name)
indexing:expression → leftOp:expression rightOp:expression	leftOp.type.squareBrackets(name) != null	indexing.lValue = true indexing.type = leftOp.type.squareBrackets(name)

logical:expression → leftOp:expression operator:String rightOp:expression	leftOp.type.logical(rightOp.type) != null	logical.lValue = false logical.type = leftOp.type.logical(rightOp.type)
unaryMinus:expression → expression	expression.type.arithmetic() != null	unaryMinus.lValue = false unaryMinus.type = expression.type.arithmetic()
unaryNot:expression → expression	expression.type.logical() != null	unaryNot.lValue = false unaryNot.type = expression.type.logical()
variable:expression → name:String		variable.lValue = true variable.type = variable.definition.type
ternaryOperator:expression → condition:expression trueExpression:expression falseExpression:expression	condition.type.isLogical() trueExpression.type.rightfulSuperType(falseExpression.type) != null	ternaryOperator.lValue = false ternaryOperator.type = trueExpression.type.rightfulSuperType(falseExpression.type)
invocation:statement, expression → function:variable arguments:expression	function.type.parenthesis($arguments_i$.type) != null	invocation.lValue = false invocation.assignsValue = false invocation.type = function.type.parenthesis($arguments_i$.type) invocation. isntLoopOrSwitchAndHasBreak = false return false;
preArithmetic:statement, expression → expression operator:String	expression.lValue == true expression.type.pArithmetic() != null	preArithmetic.lValue = false preArithmetic.assignsValue = true preArithmetic.type = expression.type.pArithmetic() preArithmetic. isntLoopOrSwitchAndHasBreak = false return false;
postArithmetic:statement, expression → expression operator:String	expression.lValue == true expression.type.pArithmetic() != null	postArithmetic.lValue = false postArithmetic.assignsValue = true postArithmetic.type = expression.type.pArithmetic()

		postArithmetic. isntLoopOrSwitchAndHasBreak = false return false;

Plantillas de Generación de Código

Sintaxis utilizada: La plantilla genera el código literalmente, salvo el texto entre { }, que es evaluado.

EXECUTE[[**program** → definition*]]() =

{if definition$_i$ instanceof VarDefinition}

EXECUTE[[definition$_i$]]

CALL main

HALT

{if definition$_i$ instanceof FunDefinition}

EXECUTE[[definition$_i$]]

EXECUTE[[**varDefinition**:definition, statement → name:String type]](scope) =

{cg.varDefinitionDirective(VarDefinition, scope)}

EXECUTE[[**funDefinition**:definition → name:String type statement*]]() =

{name} :

EXECUTE[[type.param$_i$]](“param”)

{if statement$_i$ instanceof VarDefinition}

EXECUTE[[statement$_i$]](“local”)

ENTER {funDefinition.bytesLocalVariables}

{if statement$_i$! instanceof VarDefinition}

EXECUTE[[statement$_i$]](funDefinition)

{if type.returnType instanceof VoidType}

RET 0, {funDefinition.bytesLocalVariables}, {type.bytesParameters}

EXECUTE[[**write**:statement → expression]]() =

VALUE[[expression]]

OUT {expression.type.suffix()}

EXECUTE[[**read**:statement → expression]]() =

ADDRESS[[expression]]

IN {expression.type.suffix()}

STORE {expression.type.suffix()}

EXECUTE[[**assignment**:statement, expression → left:expression right:expression]]() =

ADDRESS[[left]]

VALUE[[right]]

{cg.convert(right.type, left.type)}

STORE {left.type.suffix()}

EXECUTE[[**while**:statement → condition:expression body:statement*]]() =

{int labelNum = cg.getLabelNum();}

while{labelNum} :

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JZ end_while{labelNum}

EXECUTE[[$body_i$]](“end_while{labelNum}”)

JMP while{labelNum}

end_while{labelNum} :

EXECUTE[[**doWhile**:statement → condition:expression body:statement*]]() =

{int labelNum = cg.getLabelNum();}

do_while{labelNum} :

EXECUTE[[$body_i$]](“end_do_while{labelNum}”)

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JNZ do_while{labelNum}

{if ∃ $body_i$.isntLoopOrSwitchAndHasBreak }

end_do_while{labelNum} :

EXECUTE[[**for**:statement → initializationStatements:statement* condition:expression incrementStatements:statement* body:statement*]]() =

{int labelNum = cg.getLabelNum();}

EXECUTE[[$initializationStatements_i$]]

for{labelNum} :

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JZ end_for{labelNum}

EXECUTE[[$body_i$]](“end_for{labelNum}”)

EXECUTE[[$incrementStatements_i$]]

JMP for{labelNum}

end_for{labelNum} :

EXECUTE[[**ifStatement**:statement → condition:expression ifBody:statement* elseBody:statement*]]() =

{int labelNum = cg.getLabelNum();}

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

{if elseBody ≠ ø}

JZ else{labelNum}

{else}

JZ end_if{labelNum}

EXECUTE[[$ifBody_i$]]

{if elseBody ≠ ø}

JMP end_if{labelNum}

else{labelNum} :

EXECUTE[[$elseBody_i$]]

end_if{labelNum} :

```
EXECUTE[[ invocation:statement, expression → function:variable    arguments:expression* ]]() =
        VALUE[[ invocation ]]
        {if function.type.returnType != VoidType}
                POP {function.type.returnType.suffix()}

EXECUTE[[ return:statement → expression ]](funDefinition) =
        VALUE[[ expression ]]
        {cg.convert(expression.type, funDefinition.type.returnType)}
        RET {funDefinition.type.returnType.numBytes()},
                {funDefinition.bytesLocalVariables}, {funDefinition.type.bytesParameters}

EXECUTE[[ preArithmetic:statement, expression → expression    operator:String ]]() =
        VALUE[[ preArithmetic ]]
        POP{preArithmetic.type.suffix()}

EXECUTE[[ postArithmetic:statement, expression → expression    operator:String ]]() =
        VALUE[[ postArithmetic ]]
        POP{postArithmetic.type.suffix()}

EXECUTE[[ break:statement → ]](end_label_name) =
        JMP end_label_name

ADDRESS[[ variable:expression → name:String ]]() =
        {if variable.definition.scope == 0}
                PUSHA {variable.definition.offset}
        {else}
                PUSHA BP
                PUSH {variable.definition.offset}
                ADD
```

```
ADDRESS[[ indexing:expression → leftOp:expression    rightOp:expression ]]() =
        ADDRESS[[ leftOp ]]
        VALUE[[ rightOp ]]
        {cg.convert(rightOp.type, IntType)}
        PUSH {indexing.type.numBytes()}
        MUL
        ADD

ADDRESS[[ fieldAccess:expression → leftOp:expression    name:String ]]() =
        ADDRESS[[ leftOp ]]
        PUSH {leftOp.type.get(name).offset}
        ADD

VALUE[[ charLiteral:expression → value:char ]]() =
        PUSHB {value}
VALUE[[ intLiteral:expression → value:int ]]() =
        PUSH {value}
VALUE[[ realLiteral:expression → value:double ]]() =
        PUSHF {value}

VALUE[[ variable:expression → name:String ]]() =
        ADDRESS[[ variable ]]
        LOAD{expression.type.suffix()}

VALUE[[ arithmetic:expression → leftOp:expression    operator:String    rightOp:expression ]]() =
        VALUE[[ leftOp ]]
        {cg.convert(leftOp.type, arithmetic.type)}
        VALUE[[ rightOp ]]
        {cg.convert(rightOp.type, arithmetic.type)}
        {cg.arithmetic(arithmetic.type, operator)}
```

```
VALUE[[ comparison:expression → leftOp:expression  operator:String  rightOp:expression ]]()=
        VALUE[[ leftOp ]]
        {cg.convert(leftOp.type, leftOp.type.superType(rightOp.type))}
        VALUE[[ rightOp ]]
        {cg.convert(rightOp.type, rightOp.type.superType(leftOp.type))}
        {cg.comparison(comparison.type, operator)}

VALUE[[ logical:expression → leftOp:expression  operator:String  rightOp:expression ]]()=
        {int labelNum = cg.getLabelNum();}
        VALUE[[ leftOp ]]
        {cg.convert(leftOp.type, logical.type)}
        {if operator == "&&"}
                DUP{logical.type.suffix()}
                JZ end_logical{labelNum}
        {else if operator == "||"}
                DUP{logical.type.suffix()}
                JNZ end_logical{labelNum}
        VALUE[[ rightOp ]]
        {cg.convert(rightOp.type, logical.type)}
        {cg.logical(operator)}
end_logical{labelNum} :

VALUE[[ cast:expression → castType:type  expression ]]() =
        VALUE[[ expression ]]
        {cg.convert(expression.type, castType)}

VALUE[[ unaryNot:expression → expression ]]() =
        VALUE[[ expression ]]
        {cg.convert(expression.type, unaryNot.type)}
        NOT
```

VALUE[[**unaryMinus**:expression → expression]]() =

VALUE[[expression]]

{cg.convert(expression.type, unaryMinus.type)}

PUSH -1

MUL{unaryMinus.type.suffix()}

VALUE[[**indexing**:expression → leftOp:expression rightOp:expression]]() =

ADDRESS[[indexing]]

LOAD{indexing.type.suffix()}

VALUE[[**fieldAccess**:expression → leftOp:expression name:String]]() =

ADDRESS[[fieldAccess]]

LOAD{

{fieldAccess.type.suffix()}

VALUE[[**invocation**:statement, expression → function:variable arguments:expression*]]() =

{for arg in arguments}

VALUE[[arg_i]]

{cg.convert(arg_i.type, function.type.param.get(i++).type)}

CALL {function.name}

VALUE[[**preArithmetic**:statement, expression → expression operator:String]]()=

{Type type = expression.type instanceof CharType ? IntType : preArithmetic.type}

ADDRESS[[expression]]

VALUE[[expression]]

{if expression.type instanceof CharType}

{cg.convert(CharType, IntType)}

PUSH 1

{cg.convert(IntType, type)}

{cg.pArithmetic(type, operator)}

{if expression.type instanceof CharType}

{cg.convert(IntType, CharType)}

STORE{preArithmetic.type.suffix()}

VALUE[[expression]]

VALUE[[**postArithmetic**:statement, expression → expression operator:String]]()=

{Type type = expression.type instanceof CharType ? IntType : postArithmetic.type}

VALUE[[expression]]

ADDRESS[[expression]]

VALUE[[expression]]

{if expression.type instanceof CharType}

{cg.convert(CharType, IntType)}

PUSH 1

{cg.convert(IntType, type)}

{cg.pArithmetic(type, operator)}

{if expression.type instanceof CharType}

{cg.convert(IntType, CharType)}

STORE{postArithmetic.type.suffix()}

VALUE[[**ternaryOperator**:expression → condition:expression trueExpression:expression falseExpression:expression]]()=

{int labelNum = cg.getLabelNum();}

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JZ terOp_false_exp{labelNum}

VALUE[[trueExpression]]

{cg.convert(trueExpression.type, ternaryOperator.type)}

JMP end_terOp{labelNum}

terOp_false_exp{labelNum} :

VALUE[[falseExpression]]

{cg.convert(falseExpression.type, ternaryOperator.type)}

end_terOp{labelNum} :

VALUE[[**assignment**:statement, expression → left:expression right:expression]]()=

ADDRESS[[left]]

VALUE[[right]]

{cg.convert(right.type, left.type)}

STORE {left.type.suffix()}

VALUE[[left]]

Ampliaciones

1. Promoción implícita de tipos

2. MasIgual … (+=, -=, …)

Funcionalidad

a: int; a = 0; b: [3]int; b[0] = 2;

a+=3; a-=2; b[0] *= 4; b[0] /= 4; a %= 2;

Implementación

Se añaden los siguientes patrones al JFlex:

```
MasIgual    = "+="
MenosIgual  = "-="
PorIgual    = "*="
DivIgual = "/="
ModIgual    = "%="
```

Se añaden estos tokens en el Yacc:

```
%right '=' PLUS_EQUAL MINUS_EQUAL MUL_EQUAL DIV_EQUAL MOD_EQUAL
```

Se añaden estas acciones en el JFlex:

```
{MasIgual}      { this.yylval = yytext(); return Parser.PLUS_EQUAL; }
{MenosIgual}    { this.yylval = yytext(); return Parser.MINUS_EQUAL; }
{PorIgual}      { this.yylval = yytext(); return Parser.MUL_EQUAL; }
{DivIgual}      { this.yylval = yytext(); return Parser.DIV_EQUAL; }
{ModIgual}      { this.yylval = yytext(); return Parser.MOD_EQUAL; }
```

Se añaden en Yacc en la GLC los siguientes statements:

```
statement_without_semicolon:
                ...
                | plusIqual
                | minusIqual
                | mulIqual
                | divIqual
                | modIqual

plusEqual:      expression PLUS_EQUAL expression
minusEqual:     expression MINUS_EQUAL expression
mulEqual:       expression MUL_EQUAL expression
divEqual:       expression DIV_EQUAL expression
modEqual:       expression MOD_EQUAL expression
```

3. PreArithmetic y PostArithmetic (++, --)

Funcionalidad

a: int; a = 0; b: [3]int; b[0] = 2; c: char; c = 'a'; d: int;

a++; // a = 1;

b[0]++; // b[0] = 3;

c++; // c = 'b'

d = a++; // d = 1 y a = 2

d = ++a; // d = 3 y a = 3

Implementación

Se añaden los siguientes patrones al JFlex:

MasMas = "++"
MenosMenos = "--"

Se añaden estos tokens en el Yacc:

```
%nonassoc '!' UNARY_MINUS PRE_ARITHMETIC // Este es solo para la prioridad
%nonassoc PLUS_PLUS MINUS_MINUS
```

Se añaden estas acciones en el JFlex:

```
{MasMas}        { this.yylval = yytext(); return Parser.PLUS_PLUS; }
{MenosMenos}    { this.yylval = yytext(); return Parser.MINUS_MINUS; }
```

Se añaden en Yacc en la GLC, como expresiones y como sentencias:

```
expression:
                ...
                | pre_plus_plus_as_expression
                | pre_minus_minus_as_expression
                | post_plus_plus_as_expression
                | post_minus_minus_as_expression

statement_without_semicolon:
                ...
                | pre_plus_plus_as_expression
                | pre_minus_minus_as_expression
                | post_plus_plus_as_expression
                | post_minus_minus_as_expression

pre_plus_plus_as_expression: PLUS_PLUS expression          %prec PRE_ARITHMETIC
pre_minus_minus_as_expression: MINUS_MINUS expression %prec PRE_ARITHMETIC
post_plus_plus_as_expression: expression PLUS_PLUS
post_minus_minus_as_expression: expression MINUS_MINUS
```

Se añaden en la gramática abstracta los dos siguientes (y se crean las clases Java correspondientes):

preArithmetic:statement, expression → expression operator:String

postArithmetic:statement, expression → expression operator:String

Se añade lo siguiente en la gramática atribuida de la fase de comprobación de tipos:

Gramática Abstracta	Predicados	Reglas semánticas
preArithmetic:statement, expression → expression operator:String	expression.lValue == true expression.type.pArithmetic() != null	preArithmetic.lValue = false preArithmetic.type = expression.type.pArithmetic()
postArithmetic:statement, expression → expression operator:String	expression.lValue == true expression.type.pArithmetic() != null	postArithmetic.lValue = false postArithmetic.type = expression.type.pArithmetic()

Se añaden las siguientes plantillas de generación de Código:

VALUE[[**preArithmetic**:statement, expression → expression operator:String]]()=

{Type type = expression.type instanceof CharType ? IntType : preArithmetic.type}

ADDRESS[[expression]]

VALUE[[expression]]

{if expression.type instanceof CharType}

{cg.convert(CharType, IntType)}

PUSH 1

{cg.convert(IntType, type)}

{cg.pArithmetic(type, operator)}

{if expression.type instanceof CharType}

{cg.convert(IntType, CharType)}

STORE{preArithmetic.type.suffix()}

VALUE[[expression]]

EXECUTE[[**preArithmetic**:statement, expression → expression operator:String]]() =

VALUE[[preArithmetic]]

POP{preArithmetic.type.suffix()}

```
VALUE[[ postArithmetic:statement, expression → expression    operator:String ]]()=
        {Type type = expression.type instanceof CharType ? IntType : postArithmetic.type}
        VALUE[[ expression ]]
        ADDRESS[[ expression ]]
        VALUE[[ expression ]]
        {if expression.type instanceof CharType}
                {cg.convert(CharType, IntType)}
        PUSH 1
        {cg.convert(IntType, type)}
        {cg.pArithmetic(type, operator)}
        {if expression.type instanceof CharType}
                {cg.convert(IntType, CharType)}
        STORE{postArithmetic.type.suffix()}

EXECUTE[[ postArithmetic:statement, expression → expression    operator:String ]]() =
        VALUE[[ postArithmetic ]]
        POP{postArithmetic.type.suffix()}
```

4. Println

Funcionalidad

println ‘mensaje’; ⟺ print ‘mensaje’, ‘\n’;

Implementación

Se añade lo siguiente a Flex:

"println"{ return Parser.PRINTLN; }

Se añaden estos tokens en el Yacc:

%token PRINTLN

Se añade en Yacc en la GLC:

write: PRINT expressions

| PRINTLN expressions

En la ultima regla, se añade al final de la lista de statements un statement Write cuya expresión es un CharLiteral con valor ‘\n’.

5. Operador ternario (?:)

Funcionalidad

```
i: int;

i = true ? 4 : 'a'; // i = 4

i = false ? 4 : 'a'; // i = 97
```

Implementación

Se añade lo siguiente a Flex:

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

Se añaden estos tokens en el Yacc:

```
%right TERNARY_OPERATOR '?'
```

Se añade en Yacc en la GLC:

```
expression: ...
            | ternaryOperator

ternaryOperator: expression '?' expression ':' expression   %prec TERNARY_OPERATOR
```

Se añade en la gramática abstracta (y se crea la clase Java correspondiente):

ternaryOperator:expression → condition:expression trueExpression:expression falseExpression:expression

Se añade lo siguiente en la gramática atribuida de la fase de comprobación de tipos:

Gramática Abstracta	Predicados	Reglas semánticas
ternaryOperator:expression → condition:expression trueExpression:expression falseExpression:expression	condition.type.isLogical() trueExpression.type.rightfulSuperType(falseExpression.type) != null	ternaryOperator.lValue = false ternaryOperator.type = trueExpression.type.rightfulSuperType(falseExpression.type)

Se añade la siguiente plantilla de generación de Código:

VALUE[[**ternaryOperator**:expression → condition:expression trueExpression:expression falseExpression:expression]]()=

```
    {int labelNum = cg.getLabelNum();}

    VALUE[[ condition ]]

    {cg.convert(condition.type, IntType)}
```

```
        JZ terOp_false_exp{labelNum}
        VALUE[[ trueExpression ]]
        {cg.convert(trueExpression.type, ternaryOperator.type)}
        JMP end_terOp{labelNum}
terOp_false_exp{labelNum} :
        VALUE[[ falseExpression ]]
        {cg.convert(falseExpression.type, ternaryOperator.type)}
end_terOp{labelNum} :
```

6. Asignación múltiple

Funcionalidad

i: int; r: double;

r = i = 4;

Implementación

Se añade en Yacc, en la GLC, la asignación como expresión:

expression: ...
| assignment_as_expression

assignment_as_expression: expression '=' expression

Se añade en la gramática abstracta que la asignación es también una expresión:

assignment:statement, expression → left:expression right:expression

Se modifica la gramática atribuida de la fase de comprobación de tipos:

Gramática Abstracta	Predicados	Reglas semánticas
assignment:statement, expression → left:expression right:expression	left.lValue == true right.type.promotesTo(left.type) != null	assignment.lValue = false assignment.type = right.type.promotesTo(left.type)

Se añade la siguiente plantilla de generación de Código:

VALUE[[**assignment**:statement, expression → left:expression right:expression]]()=

ADDRESS[[left]]

VALUE[[right]]

{cg.convert(right.type, left.type)}

STORE {left.type.suffix()}

VALUE[[left]]

EXECUTE[[**assignment**:statement, expression → left:expression right:expression]]() =

ADDRESS[[left]]

VALUE[[right]]

{cg.convert(right.type, left.type)}

STORE {left.type.suffix()}

7. XOR

Funcionalidad

```
println 1 ^^ 1; // 0

println 1 ^^ 0; // 1

println 0 ^^ 1; // 1

println 0 ^^ 0; // 0
```

Implementación

Se añaden el siguiente patrón al JFlex:

```
Xor        = "^^"
```

Se añaden estos tokens en el Yacc:

```
%left AND OR XOR
```

Se añade esta acción en el JFlex:

```
{Xor}                { this.yylval = yytext(); return Parser.XOR; }
```

Se añaden en Yacc en la GLC una nueva expression:

```
expression: ...
            | xor_expression

xor_expression: expression XOR expression
```

En la ultima regla, se crea una expresión Logical del tipo: a ^^ b ⟺ (a || b) && !(a && b)

8. Do While

Funcionalidad

```
do :
        println 'a';
while condition;
do : {
        println 'b';
        println 'c';
} while condition;
```

Implementación

Se añade lo siguiente a Flex:

```
"do"      { return Parser.DO; }
```

Se añaden estos tokens en el Yacc:

```
%token DO
```

Se añade en Yacc en la GLC:

```
statement_without_semicolon: ...
                                    | do_while

do_while: DO ':' '{' statements '}' WHILE expression
        | DO ':' statement WHILE expression
```

Se añade en la gramática abstracta que la asignación es también una expresión:

doWhile:statement → condition:expression body:statement*

Se añade lo siguiente en la gramática atribuida de la fase de comprobación de tipos:

Gramática Abstracta	Predicados	Reglas semánticas
doWhile:statement → condition:expression body:statement*	condition.type.isLogical()	

Se añade la siguiente plantilla de generación de Código:

EXECUTE[[**doWhile**:statement → condition:expression body:statement*]]() =

{int labelNum = cg.getLabelNum();}

do_while{labelNum} :

EXECUTE[[$statement_i$]]

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JNZ do_while{labelNum}

9. Evaluación de cortocircuito en expresiones lógicas

Funcionalidad

0 **&&** v_i[0] // Al ser la expresión de la izquierda falsa, no se evalúa la expresión de la derecha, y se devuelve falso directamente, ahorrando calcular el valor de la expresión de la derecha.

1 **||** v_i[0] // Al ser la expresión de la izquierda cierta, no se evalúa la expresión de la derecha, y se devuelve cierto directamente, ahorrando calcular el valor de la expresión de la derecha.

Implementación

Se sustituye la anterior plantilla de código de Logical, en la que siempre se calculaban los valores de las dos expresiones, por una nueva, que tenga en cuenta el valor de la expresión de la izquierda para realizar una posible optimización.

Antigua plantilla de código de Logical:

VALUE[[**logical**:expression → leftOp:expression operator:String rightOp:expression]]()=

VALUE[[leftOp]]

{cg.convert(leftOp.type, logical.type)}

VALUE[[rightOp]]

{cg.convert(rightOp.type, logical.type)}

{cg.logical(operator)}

Nueva plantilla de código de Logical:

VALUE[[**logical**:expression → leftOp:expression operator:String rightOp:expression]]()=

{int labelNum = cg.getLabelNum();}

VALUE[[leftOp]]

{cg.convert(leftOp.type, logical.type)}

{if operator == "&&"}

DUP{logical.type.suffix()}

JZ end_logical{labelNum}

{else if operator == "||"}

DUP{logical.type.suffix()}

JNZ end_logical{labelNum}

VALUE[[rightOp]]

{cg.convert(rightOp.type, logical.type)}

{cg.logical(operator)}

end_logical{labelNum} :

10. If optimizado cuando no hay else

Funcionalidad

Cuando un if no tiene parte else, se simplifica el código a bajo nivel generado, eliminando el jmp end_if y la etiqueta del comienzo de la parte else.

Implementación

Se sustituye la anterior plantilla de código de IfStatement, en la que siempre se realizaba el jmp y se ponía la etiqueta del comienzo del else, por una nueva, que tenga en cuenta si hay o no parte else, para realizar una posible optimización.

Antigua plantilla de código de IfStatement:

EXECUTE[[**ifStatement**:statement → condition:expression ifBody:statement* elseBody:statement*]]() =

{int labelNum = cg.getLabelNum();}

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JZ else{labelNum}

EXECUTE[[$ifBody_i$]]

JMP end_if{labelNum}

else{labelNum} :

EXECUTE[[$elseBody_i$]]

end_if{labelNum} :

Nueva plantilla de código de IfStatement:

EXECUTE[[**ifStatement**:statement → condition:expression ifBody:statement* elseBody:statement*]]() =

{int labelNum = cg.getLabelNum();}

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

{if elseBody ≠ ø}

JZ else{labelNum}

{else}

JZ end_if{labelNum}

EXECUTE[[$ifBody_i$]]

{if elseBody ≠ ø}

JMP end_if{labelNum}

else{labelNum} :

EXECUTE[[$elseBody_i$]]

end_if{labelNum} :

11. For

Funcionalidad

```
i, j, a, b: int;

for(i = 1; i < 10; i++) : {          for(input i, j; 1; i -=1) :          for(++i; i && 1; input i, j) :
        println i;                            println ‘a’;                          println ‘b’;
        println i+1;
}

for(b=2, input a ; a < 5 ; a++, b+=2) :
        println a, ' ' ,b;
```

El primer elemento entre llaves del for ha de ser una o varias sentencias que asignen valor a una variable.

El segundo elemento entre llaves del for ha de ser una expresión lógica.

El tercer elemento entre llaves del for ha de ser una o varias sentencias que asignen valor a una variable.

Los paréntesis son obligatorios, para facilitar su visualización.

Implementación

Se añade el siguiente token en el Yacc:

```
%token FOR
```

Se añade esta acción en el JFlex:

```
"for"          { this.yylval = yytext(); return Parser.FOR; }
```

Se añaden en Yacc en la GLC:

```
statement: ...
                | for

for:    FOR '(' for_parenthesis_body ')' ':' '{' statements '}'
        | FOR '(' for_parenthesis_body ')' ':' statement

for_parenthesis_body: statement_without_semicolon_1mcs ';' expression ';'
statement_without_semicolon_1mcs

statement_without_semicolon_1mcs: statement_without_semicolon
| statement_without_semicolon_1mcs ',' statement_without_semicolon
```

Se añade en la gramática abstracta (y se crea la clase Java correspondiente):

for:statement → initializationStatements:statement* condition:expression incrementStatements:statement* body:statement*

Se añade lo siguiente en la gramática atribuida de la fase de comprobación de tipos:

Elemento	Atributo	Dominio	Tipo
Statement	assignsValue	boolean	sintetizado

Gramática Abstracta	Predicados	Reglas semánticas
assignment:statement, expression → left:expression right:expression		assignment.assignsValue = true
ifStatement:statement → condition:expression ifBody:statement* elseBody:statement*		ifStatement.assignsValue = false
read:statement → expression		read.assignsValue = true
return:statement → expression		return.assignsValue = false
while:statement → condition:expression body:statement*		while.assignsValue = false
doWhile:statement → condition:expression body:statement*		doWhile.assignsValue = false
write:statement → expression		write.assignsValue = false
for:statement → initializationStatements: statement* condition:expression incrementStatements: statement* body:statement*	condition.type.isLogical() $initializationStatements_i$.assignsValue $incrementStatements_i$.assignsValue	for.assignsValue = false
invocation:statement, expression → function:variable arguments:expression		invocation.assignsValue = false
preArithmetic:statement, expression → expression operator:String		preArithmetic.assignsValue = true
postArithmetic:statement, expression → expression operator:String		postArithmetic.assignsValue = true

Se añade la siguiente plantilla de generación de Código:

EXECUTE[[**for**:statement → initializationStatements:statement* condition:expression incrementStatements:statement* body:statement*]]() =

{int labelNum = cg.getLabelNum();}

EXECUTE[[$initializationStatements_i$]]

for{labelNum} :

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JZ end_for{labelNum}

EXECUTE[[$body_i$]]

EXECUTE[[$incrementStatements_i$]]

JMP for{labelNum}

end_for{labelNum} :

12. Control de flujo (return)

Funcionalidad

Se comprueba que todas las funciones que no sean void retornen algo. En caso de no ser así, se muestra un error.

Implementación

En la gramática atribuida de la fase de comprobación de tipos se indica, para cada Statement, si el visit que lo llama retorna true o false. True indica que ese statement tiene return, y false que no.

Los únicos Statements que retornan true son el Return y el IfStatement (este último sólo cuando al menos un Statement del ifBody y al menos un Statement del elseBody retornan true).

FunDefinition tiene que comprobar si el tipo de retorno de la función es Void o si hay algún statement de la función que al visitarlo retorne true.

Gramática Abstracta	Predicados	Reglas semánticas
funDefinition:definition → name:String type statement*	type.returnType == VoidType || ∃ (visit($statements_i$) == true)	
varDefinition:definition, statement → name:String type		return false;
assignment:statement, expression → left:expression right:expression		return false;
ifStatement:statement → condition:expression ifBody:statement* elseBody:statement*		return ∃ (visit($ifBody_i$) == true) && ∃ (visit($elseBody_i$) == true)
read:statement → expression		return false;
return:statement → expression		return true;
while:statement → condition:expression body:statement*		return false;
doWhile:statement → condition:expression body:statement*		return false;
write:statement → expression		return false;
for:statement → initializationStatements: statement* condition:expression incrementStatements: statement* body:statement*		return false;

invocation:statement, expression → function:variable arguments:expression		return false;
preArithmetic:statement, expression → expression operator:String		return false;
postArithmetic:statement, expression → expression operator:String		return false;

13. Break

Funcionalidad

for(i = 1; i < 10; i++) : {
    if(cond) :
        break;
}

while 1 :
    if(cond):
        break;

do : {
    if(cond):
        break;
} while 1;

Si hay un break suelto en una función, o dentro de un if/else que no está dentro de un bucle o un siwtch, se da un error.

NO se comprueba que haya más de un break y haya código muerto.

NO se puede poner directamente un break dentro de un bucle, aunque sea la última instrucción del mismo. No se da un error, pero como las instrucciones de salto al inicio del bucle nunca se llegan a ejecutar debido al break, MAPL da un warning.

La sentencia break permite salirse de un bucle y pasar a las sentencias posteriores a dicho bucle.

(También se puede utilizar en los switch, para salirse de ellos.)

Implementación

Se añade el siguiente token en el Yacc:

```
%token BREAK
```

Se añade esta acción en el JFlex:

```
"break"  { return Parser.BREAK; }
```

Se añaden en Yacc en la GLC:

```
statement_without_semicolon: ...
                           | break

break: BREAK
```

Se añade en la gramática abstracta (y se crea la clase Java correspondiente):

break:statement →

Se añade lo siguiente en la gramática atribuida de la fase de comprobación de tipos:

Elemento	Atributo	Dominio	Tipo
Statement	isntLoopOrSwitchAndHasBreak	boolean	sintetizado

Gramática Abstracta	Predicados	Reglas semánticas

break:statement →		break.assignsValue = false return false; break. isntLoopOrSwitchAndHasBreak = **true**
funDefinition:definition → name:String type statement*	! ∃ ($statements_i$. isntLoopOrSwitchAndHasBreak)	
varDefinition:definition, statement → name:String type		varDefinition. isntLoopOrSwitchAndHasBreak = false
assignment:statement, expression → left:expression right:expression		assignment. isntLoopOrSwitchAndHasBreak = false
ifStatement:statement → condition:expression ifBody:statement* elseBody:statement*		ifStatement. isntLoopOrSwitchAndHasBreak = ∃ ($ifBody_i$. isntLoopOrSwitchAndHasBreak) || ∃ ($elseBody_i$. isntLoopOrSwitchAndHasBreak)
read:statement → expression		read. isntLoopOrSwitchAndHasBreak = false
return:statement → expression		return. isntLoopOrSwitchAndHasBreak = false
while:statement → condition:expression body:statement*		while. isntLoopOrSwitchAndHasBreak = false
doWhile:statement → condition:expression body:statement*		doWhile. isntLoopOrSwitchAndHasBreak = false
write:statement → expression		write. isntLoopOrSwitchAndHasBreak = false
for:statement → initializationStatements: statement* condition:expression incrementStatements: statement* body:statement*		for. isntLoopOrSwitchAndHasBreak = false
invocation:statement, expression → function:variable arguments:expression		invocation. isntLoopOrSwitchAndHasBreak = false
preArithmetic:statement, expression → expression operator:String		preArithmetic. isntLoopOrSwitchAndHasBreak = false

postArithmetic:statement, expression → expression operator:String		postArithmetic. isntLoopOrSwitchAndHasBreak = false

Se añade la siguiente plantilla de generación de Código:

EXECUTE[[**break**:statement →]](end_label_name) =

JMP end_label_name

Se modifica la plantilla de generación de Código del doWhile, para que, si alguna de sus sentencias no es un bucle o un switch y tiene break, se añada una etiqueta de final del doWhile.

Los bucles han de pasar como parámetro a todas sus sentencias el nombre de su etiqueta de fin.

EXECUTE[[**doWhile**:statement → condition:expression body:statement*]]() =

{int labelNum = cg.getLabelNum();}

do_while{labelNum} :

EXECUTE[[$body_i$]](“end_do_while{labelNum}”)

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JNZ do_while{labelNum}

{if ∃ $body_i$.isntLoopOrSwitchAndHasBreak }

end_do_while{labelNum} :

EXECUTE[[**while**:statement → condition:expression body:statement*]]() =

{int labelNum = cg.getLabelNum();}

while{labelNum} :

VALUE[[condition]]

{cg.convert(condition.type, IntType)}

JZ end_while{labelNum}

EXECUTE[[$body_i$]](“end_while{labelNum}”)

JMP while{labelNum}

end_while{labelNum} :

EXECUTE[[**for**:statement → initializationStatements:statement* condition:expression incrementStatements:statement* body:statement*]]() =

```
        {int labelNum = cg.getLabelNum();}
        EXECUTE[[ initializationStatements_i ]]
for{labelNum} :
        VALUE[[ condition ]]
        {cg.convert(condition.type, IntType)}
        JZ end_for{labelNum}
        EXECUTE[[ body_i ]]("end_for{labelNum}")
        EXECUTE[[ incrementStatements_i ]]
        JMP for{labelNum}
end_for{labelNum} :
```

14. Switch (No implementado)

Funcionalidad

```
a: int;   a = 1;          c: char;   c = ‘a’;

switch a : {
case 1:
        print ‘A’;
        break;
}
```

```
switch a : {
case 1:
        print ‘A’;
}
```

```
switch a : {
case 1:
        print ‘A’;
case 2:
        print ‘B’;
}
```

```
switch a : {
case ‘a’:
        print ‘A’;
        break;
case 2:
        print ‘B’;
}
```

```
switch c : {
case ‘a’:
        print ‘A’;
case 2:
        print ‘B’;
        break;
default:
        print ‘C’;
}
```

```
switch a : {
        default:
                print ‘A’;
                break;
}
```

La expresión de control del switch debe promover al tipo entero.

Las expresiones de los case sólo pueden ser literales enteros o literales char.

Las expresiones de los case no pueden estar repetidas.

Puede haber o no sentencias break en los case. En caso de entrar por un case y no haber break, se ejecutarán después las sentencias del case siguiente.

Si la expresión de control no coincide con la expresión de ningún case, se ejecutarán las sentencias de default (si lo hay). El default siempre es el ultimo case y es opcional.

Implementación

Se añaden los siguientes tokens en el Yacc:

```
%token SWITCH
%token DEFAULT
```

Se añaden estas acciones en el JFlex:

```
"switch"{ return Parser.SWITCH; }
"default"            { return Parser.DEFAULT; }
```

Se añaden en Yacc en la GLC:

```
statement: ...
            | switch

switch: SWITCH expression ':' '{' switch_body '}'

switch_body: optional_cases optional_default

optional_cases: λ

                    | optional_cases case

case: CASE expression ':' optional_statements

optional_statements: λ

                                | optional_statements statement

optional_default: λ

                            | default

default: DEFAULT ':' optional_statements
```

Se añade en la gramática abstracta (y se crea la clase Java correspondiente):

switch:statement → controlExpression:expression cases:case* defaultCase:case

case → expression statement*

La expression del defaultCase será null.

Se añade lo siguiente en la gramática atribuida de la fase de comprobación de tipos:

Gramática Abstracta	Predicados	Reglas semánticas
switch:statement → controlExpression:expression cases:case* defaultCase:case	controlExpression.type.promotesTo(IntType) $cases_i$.expression instanceof CharLiteral || $cases_i$.expression instanceof IntLiteral ((int) $cases_i$.expression.value) != ((int) $cases_j$.expression.value), con i≠j	switch.assignsValue = false return visit($case_i$) == true && visit(defaultCase) == true switch. isntLoopOrSwitchAndHasBreak = false
case → expression statement*		return ∃ (visit($statement_i$) == true)

Se añaden las siguientes plantillas de generación de Código:

EXECUTE[[**switch**:statement → controlExpression:expression cases:case* defaultCase:case]]() =

{int labelNum = cg.getLabelNum();}

{int i = 0;}

{for(Case case: cases)}

VALUE[[controlExpression]]

{cg.convert(controlExpression.type, IntType)}

VALUE[[case.expression]]

{cg.convert(case.expression.type, IntType)}

EQ

JNZ switch{labelNum}_case{i}

JMP switch{labelNum}_default_case

{int i = 0;}

{for(Case case: cases)}

switch{labelNum}_case{i}:

EXECUTE[[case]]

switch{labelNum}_default_case :

EXECUTE[[defaultCase]]

EXECUTE[[**case** → expression statement*]]() =

EXECUTE[[$statement_i$]]