Documentación

ERIK GABRIEL GONZÁLEZ GARCÍA

UO224164

# Apartado Léxico

## Expresiones regulares:

TOKENS = [%.\-+\*/<>;(){}!=:,&|\[\](&&)(||)?]

JUMPS = [ \n\t\r]\*

ConstanteEntera = [0-9]+

REAL = ({ConstanteEntera}\*['.']{ConstanteEntera}\*)

REAL\_EXPONENTE = ({REAL}|{REAL}{EXPONENTE})

EXPONENTE = ([eE][+\-]?{ConstanteEntera})

CHAR\_VALUE = .

CHAR = '({CHAR\_VALUE}|(\\{CHAR\_VALUE})|{ConstanteEntera}|(\\{ConstanteEntera}))'

NUMBER = ({REAL}|{ConstanteEntera})\*

Word = [a-zA-Z]+

IDENT = ({Word}|{ConstanteEntera}|\_)\*

COMMENT = #.\*

BIG\_COMMENT = \"\"\" ~ \"\"\"

DEF = def

RETURN = return

WHILE = while

IF = if

ELSE = else

PRINT = print

INPUT = input

STRUCT = struct

INT = int

REAL\_TYPE = double

CHAR\_TYPE = char

VOID = void

MAIN = main

SWITCH = "switch"

CASE = "case"

BREAK = "break"

GREATER\_THAN = ">="

LESS\_THAN = "<="

NEQ = "!="

EQ = "=="

AND = "&&"

OR = "||"

RANGE\_LEFT = "<<"

RANGE\_RIGHT = ">>"

## Generación de código de las expresiones regulares

|  |  |
| --- | --- |
| {SWITCH} | { this.yylval = yytext(); return Parser.SWITCH; } |
| {CASE} | { this.yylval = yytext(); return Parser.CASE; } |
| {BREAK} | { this.yylval = yytext(); return Parser.BREAK; } |
| {RANGE\_LEFT} | { this.yylval = yytext(); return Parser.RANGE\_LEFT; } |
| {RANGE\_RIGHT} | { this.yylval = yytext(); return Parser.RANGE\_RIGHT; } |
| {AND} | { this.yylval = yytext(); return Parser.AND; } |
| {OR} | { this.yylval = yytext(); return Parser.OR; } |
| {GREATER\_THAN} | { this.yylval = yytext(); return Parser.GREATER\_THAN; } |
| {LESS\_THAN} | { this.yylval = yytext(); return Parser.LESS\_THAN; } |
| {EQ} | { this.yylval = yytext(); return Parser.EQ; } |
| {NEQ} | { this.yylval = yytext(); return Parser.NEQ; } |
| {VOID} | { this.yylval = new String(yytext()); return Parser.VOID; } |
| {INT} | { this.yylval = new String(yytext()); return Parser.INT; } |
| {REAL\_TYPE} | { this.yylval = new String(yytext()); return Parser.REAL\_TYPE; } |
| {CHAR\_TYPE} | { this.yylval = new String(yytext()); return Parser.CHAR\_TYPE; } |
| {STRUCT} | { this.yylval = new String(yytext()); return Parser.STRUCT; } |
| {WHILE} | { this.yylval = new String(yytext()); return Parser.WHILE; } |
| {MAIN} | { this.yylval = new String(yytext()); return Parser.MAIN; } |
| {IF} | { this.yylval = new String(yytext()); return Parser.IF; } |
| {ELSE} | { this.yylval = new String(yytext()); return Parser.ELSE; } |
| {INPUT} | { this.yylval = new String(yytext()); return Parser.INPUT; } |
| {PRINT} | { this.yylval = new String(yytext()); return Parser.PRINT; } |
| {RETURN} | { this.yylval = new String(yytext()); return Parser.RETURN; } |
| {DEF} | { this.yylval = new String(yytext()); return Parser.def; } |
| {TOKENS} | { return yytext().charAt(0); } |
| {JUMPS} | { } |
| {COMMENT} | { } |
| {BIG\_COMMENT} | { } |
| {CHAR} | { this.yylval = new String(yytext()); return Parser.CHAR\_CONSTANT; } |
| {ConstanteEntera} | { this.yylval = new Integer(yytext()); return Parser.INT\_CONSTANT; } |
| {Word} | { this.yylval = new String(yytext()); return Parser.ID; } |
| {IDENT} | { this.yylval = new String(yytext()); return Parser.ID; } |
| {REAL\_EXPONENTE} | { this.yylval = new Double(yytext()); return Parser.REAL\_CONSTANT; } |

# Apartado Sintáctico

## Gramática libre de contexto

|  |  |
| --- | --- |
| **programa**: definiciones main  ; | {  ast = new **Program**(scanner.getLine(), scanner.getColumn(), (List<Definition>)$1); ((List)$1).add($2);  } |
| **definiciones**: definiciones definicionVariable  | definiciones function  |  ; | { $$ = $1; ((List)$$).addAll((List)$2); }  { $$ = $1; ((List)$$).add($2); }  { $$ = new **ArrayList**(); } |
| **definicion**: definicionVariable  | function  ; |  |
| **statement**: assigment  | if  | while  | call\_function  | return  | print  | input  | switch  ; | { $$ = $1; }  { $$ = $1; }  { $$ = $1; }  { $$ = $1; }  { $$ = $1; }  { $$ = $1; }  { $$ = $1; }  { $$ = $1; } |
| **composedStatement**: statement  | '{' statements '}'  | '{' '}'  ; | { $$ = new **ArrayList**(); ((List)$$).addAll((List) $1);}  { $$ = $2; }  { $$ = new **ArrayList**(); } |
| **expression**: expression '+' expression  | expression '-' expression  | expression '\*' expression  | expression '/' expression  | expression '%' expression  | expression '?' expression ':' expression  | expression RANGE\_LEFT expression RANGE\_LEFT expression  | expression RANGE\_RIGHT expression RANGE\_RIGHT expression  | expression '.' ID  | expression '<' expression  | expression '>' expression  | expression EQ expression  | expression GREATER\_THAN expression  | expression LESS\_THAN expression  | expression NEQ expression  | expression OR expression    | expression AND expression    | '(' basic\_type ')' expression %prec CAST  | ID '(' expressions\_or\_empty ')'  | '(' expression ')'  | '-' expression %prec UNARY\_MINUS  | '!' expression  | expression '['expression']'    | INT\_CONSTANT  | REAL\_ CONSTANT  | CHAR\_CONSTANT  | ID  ; | { $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "+", (Expression)$3); }  { $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "-", (Expression)$3); }  { $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "\*", (Expression)$3); }  { $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "/", (Expression)$3); }  { $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "%", (Expression)$3); }  { $$ = new **TernaryOperator**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (Expression)$3, (Expression)$5); }  {$$ = new **RangeComparator**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (Expression)$3, (Expression)$5, "<<"); }  {$$ = new **RangeComparator**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (Expression)$3, (Expression)$5, ">>"); }  { $$ = new **FieldAccess**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (String)$3); }  { $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "<", (Expression)$3); }  { $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, ">", (Expression)$3); }  { $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "==", (Expression)$3); }  { $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, ">=", (Expression)$3); }  { $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "<=", (Expression)$3); }  { $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "!=", (Expression)$3); }  { $$ = new **Logical**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (String)$2, (Expression)$3); }  { $$ = new **Logical**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (String)$2, (Expression)$3); }  { $$ = new **Cast**(scanner.getLine(), scanner.getColumn(), ((Expression)$4), (Type)$2); }  { $$ = new **Invocation**(scanner.getLine(), scanner.getColumn(), new Variable(scanner.getLine(), scanner.getColumn(), (String)$1), (List<Expression>)$3); }  { $$ = $2; }  { $$ = new **UnaryMinus**(scanner.getLine(), scanner.getColumn(), (Expression)$2); }  { $$ = new **UnaryNot**(scanner.getLine(), scanner.getColumn(), (Expression)$2); }  { $$ = new **Indexing**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "[]", (Expression)$3);}  { $$ = new **IntLiteral**(scanner.getLine(), scanner.getColumn(), (int)$1); }  { $$ = new **RealLiteral**(scanner.getLine(), scanner.getColumn(),(double)$1); }  { $$ = new **CharLiteral**(scanner.getLine(), scanner.getColumn(), (String)$1); }  { $$ = new **Variable**(scanner.getLine(), scanner.getColumn(), (String)$1); } |
| **expressions**: expressions ',' expression  | expression  ; | { $$ = $1; ((List)$$).add($3); }  { $$ = new **ArrayList**(); ((List)$$).add($1); } |
| **expressions\_or\_empty**: expressions  |  ; | { $$ = $1; }  { $$ = new **ArrayList**(); } |
| **definicionVariable**: ids ':' type ';'  ; | { $$ = new **ArrayList**(); for(String id : (List<String>)$1) ((List)$$).add(new **VarDefinition**(scanner.getLine(), scanner.getColumn(), id, (Type)$3)); } |
| **parametrosFuncion**: parametrosFuncion ',' definicionParametro  | definicionParametro  |  ; | { $$ = $1; ((List)$$).add($3); }  { $$ = new **ArrayList**(); ((List)$$).add($1); }  { $$ = new **ArrayList**(); } |
| **definicionParametro**: ID ':' basic\_type  ; | { $$ = new **VarDefinition**(scanner.getLine(), scanner.getColumn(), (String)$1, (Type)$3); } |
| **ids**: ids ',' ID  | ID  ; | { $$ = $1; ((List)$$).add($3); }  { $$ = new **ArrayList**(); ((List)$$).add($1); } |
| **function**: def ident '(' parametrosFuncion ')' ':' return\_type '{' function\_body '}'  ; | { $$ = new **FunctionDefinition**(scanner.getLine(), scanner.getColumn(), (Variable)$2, new **FunctionType**(scanner.getLine(), scanner.getColumn(), (List<VarDefinition>)$4, (Type)$7), (List)((Object[]) $9)[0], (List)((Object[])$9)[1]); } |
| **ident**: ID  ; | { $$ = new **Variable**(scanner.getLine(), scanner.getColumn(), (String)$1); } |
| **main**: def MAIN '(' ')' ':' VOID '{' function\_body '}'  ; | { $$ = new **FunctionDefinition**(scanner.getLine(), scanner.getColumn(), new **Variable**(scanner.getLine(), scanner.getColumn(), "main"), new **FunctionType**(scanner.getLine(), scanner.getColumn(), new ArrayList(), VoidType.getInstance()), (List)((Object[]) $8)[0], (List)((Object[])$8)[1]); } |
| **function\_body**: function\_var\_declaration statements  | function\_var\_declaration  | statements  |  ; | { $$ = new **Object**[] {$1, $2}; }  { $$ = new **Object**[] {$1, new **ArrayList**<Statement>()}; }  { $$ = new **Object**[] {new **ArrayList**<VarDefinition>(), $1}; }  { $$ = new **Object**[] {new **ArrayList**(), new ArrayList()}; } |
| **function\_var\_declaration**: function\_var\_declaration definicionVariable  | definicionVariable  ; | { $$ = $1; ((List)$$).addAll((List)$2); }  { $$ = $1; } |
| **statements**: statements statement  | statement  ; | { $$ = $1; ((List)$$).addAll((List)$2); }  { $$ = new **ArrayList**(); ((List)$$).addAll((List)$1); } |
| **switch**: SWITCH '(' ident ')' ':' '{' cases '}'  ; | { $$ = new **ArrayList**(); ((List)$$).add(new **Switch**(scanner.getLine(), scanner.getColumn(), (Variable)$3, (List)$7)); } |
| **cases**: cases case  | case  ; | { $$ = $1; ((List)$$).add($2); }  { $$ = new **ArrayList**(); ((List)$$).add($1); } |
| **case**: CASE expression ':' statements break  ; | { $$ = new **Case**(scanner.getLine(), scanner.getColumn(), (Expression)$2, (List)$4, (Statement)$5); } |
| **break**: BREAK ';'  |  ; | { $$ = new **Break**(scanner.getLine(), scanner.getColumn()); }  { $$ = null; } |
| **call\_function**: ID '(' expressions\_or\_empty ')' ';'  ; | { $$ = new **ArrayList**(); ((List)$$).add(new **Invocation**(scanner.getLine(), scanner.getColumn(), new **Variable**(scanner.getLine(), scanner.getColumn(), (String)$1), (List)$3)); } |
| **assigment**: expression '=' expression ';'  ; | { $$ = new **ArrayList**(); ((List)$$).add(new **Assignment**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (Expression)$3)); } |
| **while**: WHILE expression ':' composedStatement  ; | { $$ = new **ArrayList**(); ((List)$$).add(new **While**(scanner.getLine(), scanner.getColumn(), (Expression)$2, (List)$4)); } |
| **return**: RETURN expression ';'  ; | { $$ = new **ArrayList**(); ((List)$$).add(new **Return**(scanner.getLine(), scanner.getColumn(), (Expression)$2)); } |
| **print**: PRINT expressions ';'  ; | { $$ = new ArrayList(); for(Expression: (List<Expression>)$2) ((List<Write>)$$).add(new Write(scanner.getLine(), scanner.getColumn(), expression));} |
| **if**: IF expression ':' composedStatement ELSE composedStatement  | IF expression ':' composedStatement %prec MENORQUEELSE  ; | { $$ = new **ArrayList**(); ((List)$$).add(new **IfStatement**(scanner.getLine(), scanner.getColumn(), (List)$6, (List)$4, (Expression)$2)); }  {$$ = new **ArrayList**(); ((List)$$).add(new **IfStatement**(scanner.getLine(), scanner.getColumn(), new **ArrayList**(), (List)$4, (Expression)$2)); } |
| **input**: INPUT expressions ';'  ; | { $$ = new **ArrayList**(); for(Expression expression: (List<Expression>)$2) ((List<Read>)$$).add(new **Read**(scanner.getLine(), scanner.getColumn(), expression)); } |
| **struct\_body**: struct\_body definicionStruct  | definicionStruct  ; | { $$ = $1; ((List)$$).addAll((List)$2); }  { $$ = $1; } |
| **definicionStruct**: ids ':' type ';'  ; | { $$ = new **ArrayList**(); for(String id : (List<String>)$1) ((List<RecordField>)$$).add(new **RecordField**(scanner.getLine(), scanner.getColumn(), id, (Type)$3)); } |
| **type**: basic\_type  | VOID  | '[' INT\_CONSTANT ']' type  | STRUCT '{' struct\_body '}'  ; | { $$ = $1; }  { $$ = **VoidType**.getInstance(); }  { $$ = new **ArrayType**(scanner.getLine(), scanner.getColumn(), (int)$2, (Type)$4); }  { $$ = new **RecordType**(scanner.getLine(), scanner.getColumn(), (List)$3); } |
| **basic\_type**: INT  | REAL\_TYPE  | CHAR\_TYPE  ; | { $$ = **IntType**.getInstance(); }  { $$ = **RealType**.getInstance(); }  { $$ = **CharType**.getInstance(); } |
| **return\_type**: basic\_type  | VOID  ; | { $$ = $1; }  { $$ = **VoidType**.getInstance(); } |

## Gramática abstracta

Separamos los diferentes nodos del lenguaje en cuatro categorías:

* Definición
* Sentencia
* Expresión
* Tipo

Los nodos de la gramática de nuestro programa son los siguientes:

### Main:

program -> definitions: definition\*;

functionDefinition: definition -> name: String, type: Type, varDefinition: definition\*, statements:

varDefinition: definition -> name: String, type: Type;

statement\*

### Statements:

assignment: statement -> left: expression, right: expression;

break: statement;

case: statement -> condition: expression, body: statement\*, break: statement;

ifStatement: statement -> elsebody: statement\*, ifbody: statement\*, expression: expression;

invocation: statement -> function: variable, expressions: expression\*;

read: statement -> expression: expression;

return: statement -> expression: expression;

switch: statement -> param: variable, cases: case\*;

while: statement -> condition: expression, statements: statements\*;

write: statement -> expression: expression;

### Expressions:

arithmetic: expression -> left: expression, operator: String, right: expression;

cast: expression -> expression: expression, type: Type;

charLiteral: expression -> value: String;

comparison: expression -> left: expression, op: String, right: expression;

fieldAccess: expression -> leftop: expression, name: String;

indexing: expression -> left: expression, op: String, right: expression;

intLiteral: expression -> value: int;

logical: expression -> left: expression, op: String, right: expression;

rangeComparator: expression -> left: expression, value: expression, right: expression, operator: String;

realLiteral: expression -> value: double;

ternaryOperator: expression -> condition: expression, left: expression, right: expression;

unaryMinus: expression -> expression: expression;

unaryNot: expression -> expression: expression;

variable: expression -> name: String;

### Types:

arrayType: type -> size: int, type: type;

charType: type -> ;

errorType: type -> message: String, node: ASTnode;

functionType: type -> params: varDefinition\*, returnType: type;

intType: type -> ;

realType: type -> ;

recordField: type -> name: String, type: type;

recordType: type -> body: recordField\* ;

voidType: type -> ;

# Generación de código

## Plantillas:

### Execute:

EXECUTE[[assigment: statement => left: expression, right: expression]]() =

ADDRESS[left]()

VALUE[right]()

<store> left.type.suffix()

EXECUTE[[functionDefinition: definition => name: variable, type: type, vars: varDefinition\*, body: statement\*]]() =

<label> funtionDefinition.name

For (varDefinition var: vars) EXECUTE[[var]]()

For (varDefinition var: vars) EXECUTE[[var]]()

<enter>

For (statement st: body)

EXECUTE[[st]]()

If (type.returnType == voidType)

<ret> 0

EXECUTE[[ternaryOperator: expression => condition: expression, left: expression, right: expression]]() =

VALUE[[condition]]()

<jz> right

EXECUTE[[left]]()

<jmp> end

<label> right

EXECUTE[[right]]()

<label> end

EXECUTE[[ifStatement: statement => elsebody: statement\*, ifbody: statement\*, expression: expression]]() =

VALUE[[expression]]()

<jz> else

For (statement st: ifbody)

EXECUTE[[st]]()

<jmp> end

<label> else

If (elseBody != null)

For (statement st: elsebody)

EXECUTE[[st]]()

<label> end

EXECUTE[[invocation: statement => function: variable, expressions: expressions\*]]() =

For (expression exp: expressions)

VALUE[[exp]]()

<call> function

If (variable.type != voidType)

<pop> variable.type.suffix

EXECUTE[[program => definitions: definition\*]]() =

For (definition d: definitions)

If (d instanceof varDefinition)

EXECUTE[[d]]()

<call> main

For (definition d: definitions)

If (d instanceof functionDefinition)

EXECUTE[[d]]()

EXECUTE[[read: statement => expression: expression]]() =

ADDRESS[[expression]]()

<in> expression.type.suffix

<store> expression.type.suffix

EXECUTE[[return: statement => expression: expression]]() =

VALUE[[expression]]()

<ret> expression.type.bytes, fundefinition.bytes, fundefinitions.params

EXECUTE[[varDefinition: definition => name: string, type: type]]() =

EXECUTE[[while: statement => condition: expression, statements: statement\*]]() =

<label> startWhile

VALUE[[condition]]()

<jz> endWhile

For (statement st: statements)

EXECUTE[[st]]()

<jmp> startWhile

<lable> endWhile

EXECUTE[[write: statement => expression: expression]]() =

VALUE[[expression]]()

<out> expression.type.suffix

EXECUTE[[switch: statement => param: variable, cases: case\*]]() =

VALUE[[param]]()

For (case c: cases)

VALUE[[new Comparison(param, ‘==’, c.condition)]]()

<jz> nextCase

For (statement st: body)

EXECUTE[[st]]()

If (case.hasBreak())

<jmp> endSwitch

<label> nextCase

<jmp> endSwitch

<label> endSwitch

### Value:

VALUE[[arithmetic: expression => left: expression, operator: string, right: expression]]() =

VALUE[[left]]()

VALUE[[right]]()

Operation (this.operator, type.suffix)

VALUE[[cast: expression => expression: expression, type: type]]() =

VALUE[[expression]]()

<cast> expression.type, this.castType

VALUE[[charLiteral: expression => value: string]]() =

<pushb> value

VALUE[[comparison: expression => left: expression, op: string, right: expression]]() =

VALUE[[left]]()

VALUE[[right]]()

Operation (this.operator, type.suffix)

VALUE[[fieldAccess: expression => leftop: expression, name: string]]() =

ADDRESS[[expression]]()

<load> expression.type.suffix

VALUE[[indexing: expression => left: expression, op: string, right: expression]]() =

ADDRESS[[this]]()

<load> this.type.suffix

VALUE[[intLiteral: expression => value: string]]() =

<pushi> value

VALUE[[invocation: expression => function: variable, expressions: expression\*]]() =

For (expression e: expressions)

<call> function.name

VALUE[[logical: expression => left: expression, op: string, right: expression]]() =

VALUE[[left]]()

VALUE[[right]]()

Operation (this.operator, type.suffix)

VALUE[[realLiteral: expression => value: double]]() =

<pushf> value

VALUE[[unaryNot: expression => expression: expression]]() =

VALUE[[expression]]()

<not> this.type.suffix

VALUE[[unaryMinus: expression => expression: expression]]() =

<push> expression.type.suffix

VALUE[[expression]]()

<sub> expression.type.suffix

VALUE[[variable: expression => name: string]]() =

ADDRESS[[this]]()

<load> this.type.suffix

VALUE[[ternaryOperator: expression => condition: expression, left: expression, right: expression]]() =

VALUE[[condition]]()

<jz> else

VALUE[[left]]()

<jmp> end

<label> else

VALUE[[right]]()

<label> end

VALUE[[rangeComparator: expression => left: expression, value: expression, right: expression, operator: string]]() =

VALUE[[left]]()

VALUE[[right]]()

Operation (this.operator, type.suffix)

### Address:

ADDRESS[[fieldAccess: expression => leftop: expression, name: string]]() =

ADDRESS[[leftop]]()

<pushi> leftop.type.field(this.name.offset)

<add> i

ADDRESS[[indexing: expression => left: expression, op: string, right: expression]]() =

ADDRESS[[left]]()

VALUE[[right]]()

<pushi> this.type.bytes

<mul> i

<add> i

ADDRESS[[variable: expression => name: string]]() =

If (this.scope == 0)

<pusha> this.offset

Else

<pushbp>

<pushi> this.offset

<add> i