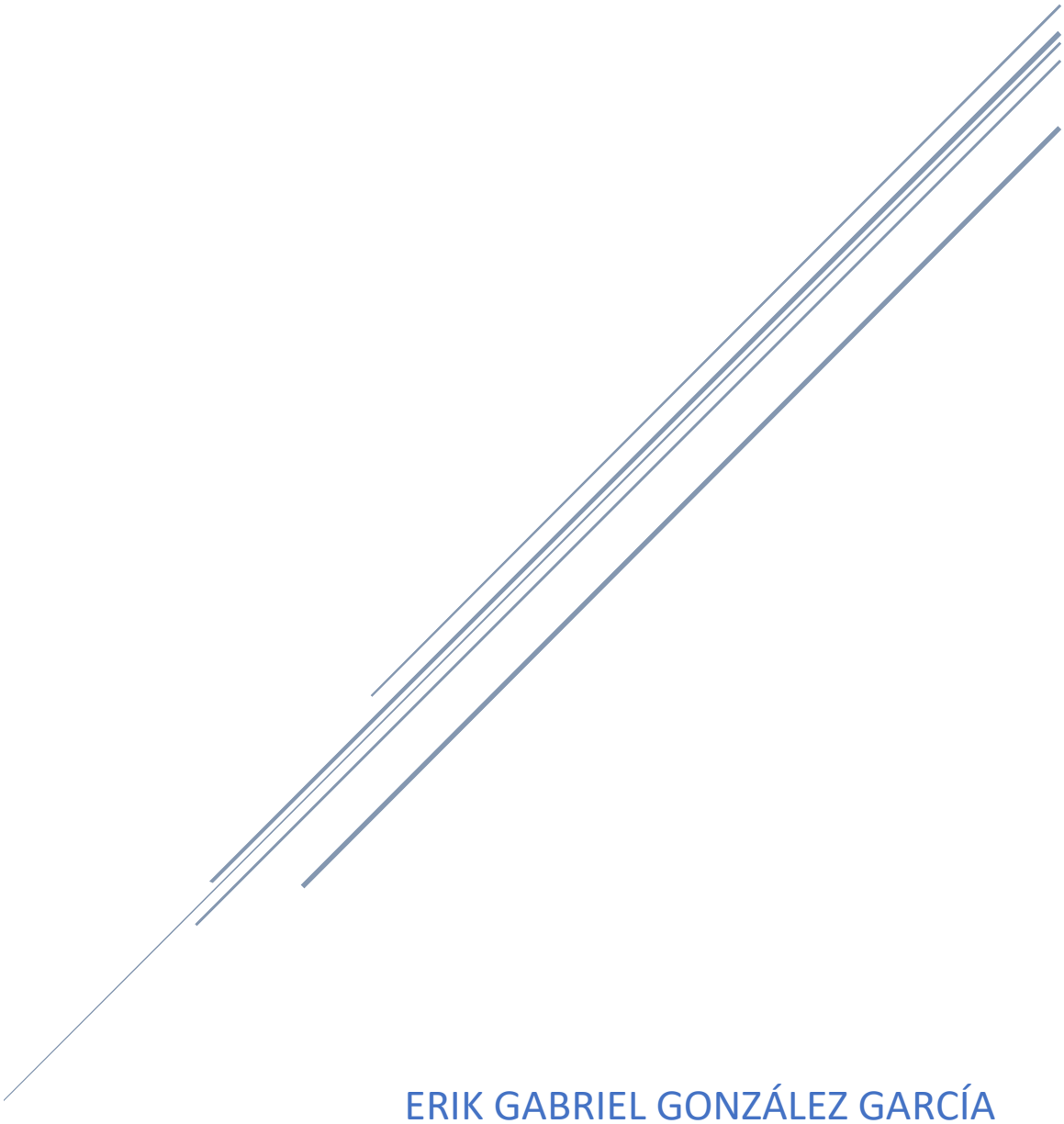
DOCUMENTACIÓN



ERIK GABRIEL GONZÁLEZ GARCÍA
UO224164

Índice

Apartado léxico

Expresiones regulares:

```
TOKENS = [%.\-+*/<>;(){}!=:,&|\[\](&&)(||)?]
JUMPS = [ \n\t\r]*
ConstanteEntera = [0-9]+
REAL = ({ConstanteEntera}*['.']{ConstanteEntera}*)
REAL_EXPONENTE = ({REAL}|({REAL}|{ConstanteEntera}){EXPONENTE})
EXPONENTE = ([eE][+\-]?{ConstanteEntera})
CHAR_VALUE = .
CHAR = '({CHAR_VALUE}|(\\{CHAR_VALUE})|{ConstanteEntera}|(\\{ConstanteEntera}))'
NUMBER = ({REAL}|{ConstanteEntera})*
Word = [a-zA-Z]+
IDENT = ({Word}|_)({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	{ $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "+", (Expression)$3); }
| expression '-' expression	{ $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "-", (Expression)$3); }
| expression '*' expression	{ $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "*", (Expression)$3); }
| expression '/' expression	{ $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "/", (Expression)$3); }
| expression '%' expression	{ $$ = new **Arithmetic**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "%", (Expression)$3); }
| expression '?' expression ':' expression	{ $$ = new **TernaryOperator**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (Expression)$3, (Expression)$5); }
| expression RANGE_LEFT expression RANGE_LEFT expression	{$$ = new **RangeComparator**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (Expression)$3, (Expression)$5, "<<"); }
| expression RANGE_RIGHT expression RANGE_RIGHT expression	{$$ = new **RangeComparator**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (Expression)$3, (Expression)$5, ">>"); }
| expression '.' ID	{ $$ = new **FieldAccess**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (String)$3); }
| expression '<' expression	{ $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "<", (Expression)$3); }
| expression '>' expression	{ $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, ">", (Expression)$3); }
| expression EQ expression	{ $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "==", (Expression)$3); }
| expression GREATER_THAN expression	{ $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, ">=", (Expression)$3); }
| expression LESS_THAN expression	{ $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "<=", (Expression)$3); }
| expression NEQ expression	{ $$ = new **Comparison**(scanner.getLine(), scanner.getColumn(), (Expression)$1, "!=", (Expression)$3); }
| expression OR expression	{ $$ = new **Logical**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (String)$2, (Expression)$3); }

| 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 ;	{ $$ = new **Logical**(scanner.getLine(), scanner.getColumn(), (Expression)$1, (String)$2, (Expression)$3); } { $$ = 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
```

Ampliaciones

Se han realizado un total de cuatro ampliaciones siendo documentadas tres de ellas (punteros y referencias se finalizó en las últimas horas y por eso no hay documentación)

Operador Ternario

El operador ternario fue realizado siguiente el siguiente planteamiento: condition ? return1 : return2 donde condition, return1 y return2 son expresiones.

Siguiendo este planteamiento lo primero que se realiza es considerar condition como una comparación. Su tipo no nos importa, solo nos importa que los tipos de la comparación sean compatibles para realizar dicha operación. Los tipos que realmente nos importan son los de las expresiones de retorno dado que deben ser compatibles con el tipo de la sentencia que acompañan, si no, se produce un error.

Al final, un operador ternario es una especie if, que siempre tiene un else. Por tanto, la plantilla de ejecución es similar.

```
EXECUTE[[ternaryOperator: expression => condition: expression, left: expression, right: expression]]() =
        VALUE[[condition]]()
        <jz> right
        EXECUTE[[left]]()
        <jmp> end
        <label> right
        EXECUTE[[right]]()
        <label> end
```

Comparador de rango

La idea del comparador de rango es comprobar si el valor que se introduce está comprendido entre los dos parámetros extremos: parameter1 << value << parameter2 ó parameter1 >> value >> parameter2. No se permite combinar los operadores << y >> en la misma expresión.

El comparador de rango es sencillo si se ve como dos comparaciones simples unidas por una operación lógica and: parameter1 << value && value << paramter2.

De esta manera la plantilla de código sería básicamente la de una operación binaria.

```
VALUE[[rangeComparator: expression => left: expression, value: expression, right: expression, operator:
string]]() =
        VALUE[[left]]()
        VALUE[[right]]()
        Operation (this.operator, type.suffix)
```

Switch con break

La ampliación de switch usando breaks fué implementada en este caso para tener en cuenta la siguiente lógica. Si se cumple la condición de un case, se ejecutará dicho código y se procederá a comprobar el resto de cases. Sin embargo, si uno de los cases que cumpla la condición, posee dentro de su cuerpo un break, se saltará automáticamente, al finalizar dicho case, al final de switch.

Es necesario especificar que el break solo puede estar al final de un case. Si no se produce un error sintáctico.

El planteamiento para generar el código del switch fue el siguiente. Partiendo de que un switch puede tener uno o más cases y cada case tiene una condición que ha de ser comprobada. Se iterará sobre cada case creando una expresión de comparación entre la condición de dicho case y la condición del switch. Si se cumple, se realizarán las sentencias de dicho case de forma normal y, si tiene break, se creará una etiqueta jump al final de switch.

```
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
```