Escuela de Ingeniería Informática de Oviedo

C. de Valdes Salas, 11
Tel. 98 510 27 96
eii@uniovi.es
33007 Oviedo

- PROGRAMMING LANGUAGE DESIGN -

Guillermo Facundo Colunga.

PROFESORADO:
Miguel García Rodríguez - garciarmiguel@uniovi.es

SUMARIO

LEXICAL PATTERNS

<table>
<tr><th>TOKEN</th><th>PATTERN</th></tr>
<tr><td>WHITE_SPACE</td><td><code>[\t\r\n\fEOF]+</code></td></tr>
<tr><td>ONE_LINE_COMMENT</td><td><code>#' ~[\r\n\f]*</code></td></tr>
<tr><td>MULTI_LINE_COMMENT</td><td><code>"""'.*?'"""'</code></td></tr>
<tr><td>DIGIT</td><td><code>[0-9]</code></td></tr>
<tr><td>NON_ZERO_DIGIT</td><td><code>[1-9]</code></td></tr>
<tr><td>EXPONENT</td><td><code>[eE] [+-]? DIGIT+</code></td></tr>
<tr><td>FRACTION</td><td><code>.' DIGIT+</code></td></tr>
<tr><td>INT_CONSTANT</td><td><code>‘0’ | [1-9]DIGIT*</code></td></tr>
<tr><td>REAL_CONSTANT</td><td><code>INT_CONSTANT? FRACTION | INT_CONSTANT '.' | INT_CONSTANT EXPONENT |
INT_CONSTANT '.' DIGIT* EXPONENT</code></td></tr>
<tr><td>STRUCT</td><td><code>‘struct'</code></td></tr>
<tr><td>CHAR_CONSTANT</td><td><code>\'' . '\'' | '\'' '\\' INT_CONSTANT INT_CONSTANT INT_CONSTANT'\'' |
'\'' '\\n' '\'' | '\'' '\\t' '\'' | '\'' '-' '\''</code></td></tr>
<tr><td>ID</td><td><code>[a-zA-Z_][a-zA-Z0-9_]*</code></td></tr>
</table>

GRAMMAR

```
program
 : (var_def';'|func_def)* main_def EOF
 ;

// -------------------------
// BUILT-IN
// -------------------------

// Built-in types are int, double and char.
build_in_type
 : 'int'
 | 'double'
 | 'char'
 | STRUCT '{' (fields) '}'
 | '['size=INT_CONSTANT']' build_in_type
 ;


// -------------------------
// VARIABLE DEFINITIONS
// -------------------------

// Non-empty enumeration of comma-separated identifiers followed by a ':' and a type.
var_def
 : single_var_def
 | multi_var_def
 | ID ':' 'struct' '{' fields '}'
 ;

// When we only have one variable definition.
single_var_def
 : ID':' build_in_type
 | 'ref' ID ':' build_in_type
 ;

// When we have an enumeration of variable and a definition.
multi_var_def
 : ID(','ID})+ ':' build_in_type
 ;

// Declared as a variable and ending with ';' always.
fields
 : (var_def';')+
 ;


// -------------------------
// FUNCTION DEFINITIONS
// -------------------------

// It's defined by the def keyword, the function identifier and a list of
// comma-separated parameters between ( and ) followed by a ':' and a return
```

```
// type or the void keyword. The return type and parameter types must be build-in.
// The function body goes between { and }.
func_def
 : 'def' ID parameters ':' (build_in_type|'void') '{' func_body '}'
 ;

// Is the last mandatory function in every program.
// Always no parameters and void return type;
main_def
 :'def' 'main' '('')' ':' 'void' '{' func_body '}'
 ;

//
parameters
 : '('')'
 | '(' single_var_def (',' single_var_def )* ')'
 ;

// Sequence of variable definitions followed by sequences of statements.
// Both must end with the ';' character.
func_body
 : (var_def';'?)*(statement';'?)*
 ;


// --------------------------
// EXPRESSIONS
// --------------------------

expression
 : '('expression')'
 | expression '['expression']'
 | expression '.' ID
 | cast
 | '-' expression
 | '!' expression
 | expression ('*'|'/'|'%') expression
 | expression ('+'|'-') expression
 | expression ('>'|'>='|'<'|'<='|'!='|'==') expression
 | expression ('&&'|'||') expression
 | expression '^' expression
 | expression ('++'|'--')
 | var_invocation
 | func_invocation
 | INT_CONSTANT
 | REAL_CONSTANT
 | CHAR_CONSTANT
 ;


// --------------------------
// STATEMENTS
// --------------------------
```

```
statement
 : if_st
 | ternary
 | while_st
 | dowhile_st
 | write_st
 | read_st
 | func_invocation
 | assigment
 | return_st
 ;

ternary
 : '(' expression ')' '?' elif_simple_body ':' elif_simple_body ';'?
 ;

// The word print followed by a non-empty comma separated list of expressions.
write_st
 : 'print' expression (','expression)* ';'?
 ;

// The word input followed by a non-empty comma separated list of expressions.
read_st
 : 'input' expression (','expression)* ';'?
 ;

// Built from and expression, a '=' operator and another expression.
assigment
 : expression '=' expression ';'?
 | expression ( '++' | '--' )
 | expression ( '+=' | '-=' | '*=' | '/=' ) expression
 | expression ('&&=' | '||=' ) expression
 | expression '^=' expression
 ;

// An expression, the conditional operator and the expression to compare.
condition
 : expression ('>'|'>='|'<'|'<='|'!='|'==') expression
 | expression ('&&'|'||') expression
 ;

// If statement can have a complex body associated or not. Same with the else.
if_st
 : 'if' expression ':' (elif_simple_body | elif_body) else_st?
 ;

// Else cal have a complex body.
else_st
 : 'else' (elif_simple_body | elif_body)
 ;

// Simple body for the else and if blocks. Only one expression.
// the expression ends in ';'.
elif_simple_body
```

```
 : statement ';'?
 ;

// Multiple expressions in the else if body.
elif_body
 : '{' (statement';'?)+ '}'
 ;

// While statement. before + after condition!
while_st
 : 'while' expression ':' while_body
 ;

// Body of the while statement, must have at least one expression.
// Every expression ends in ';'
while_body
 : '{' (statement ';'?)+ '}'
 ;

// DoWhile statement.
dowhile_st
 : 'do' ':' while_body 'while' expression ';'?
 ;

// the word return and the expression, that is mandatory.
return_st
 : 'return' expression ';'?
 ;

cast
 : '('build_in_type')' expression ';'?
 ;


// -------------------------
// INVOCATIONS
// -------------------------

func_invocation
 : ID argument ';'?
 ;

argument
 : '('')'
 | '('expression (',' expression)*')'
 ;

proc_invocation
 : ';'
 ;

var_invocation
 : ID ';'?
 ;
```

CODE TEMPLATES

```
EXECUTE [[Program: Program -> Definition*]]
      for(Definition d:Definition)
            if(d instanceof VarDefinition)
                  EXECUTE[[d]]()

      <CALL MAIN>
      <HALT>

      for(Definition d:Definition)
            if(d instanceof FunDefinition)
                  EXECUTE[[d]]()

EXECUTE[[FunDefinition: Definition -> Type Statement*]]
      Definition.Name <:>
      <ENTER> Definition.LocalBytes
      for(Statement s:Statement*)
            if(!s instanceof VarDefinition)
                  EXECUTE[[s]]()
      if(Type.ReturnType instanceof VoidType)
            <RET> 0 <,> Definition.LocalBytes <,> Definition.ParamBytes

EXECUTE[[Write: Statement -> Exp]]
      VALUE[[Exp]]()
      <OUT> Exp.Type.Suffix()

EXECUTE[[Read: Statement -> Exp]]
      ADDRESS[[Exp]]()
      <IN> Exp.Type.Suffix()
      <STORE> Exp.Type.Suffix()

EXECUTE[[Assigment: Statement -> Exp1 Exp2]]
      ADDRESS[[Exp1]]()
      VALUE[[Exp2]]()
      cg.convert(Exp2.Type,Exp1.Type)
      <STORE> Exp1.Type.Suffix()

EXECUTE[[IfStatement: Statement -> Exp if:Statement* else:Statement*]]
      int label =  cg.getLabels(2);
      VALUE[[Exp]]()
      <JZ><LABEL> label

      for(Statement s:if)
            EXECUTE[[s]]()

      <JMP><LABEL> label+1
      <LABEL> label <:>

      for(Statement s:else)
            EXECUTE[[s]]()
```

```
      <LABEL> label+1 <:>

EXECUTE[[WhileStatement: Statement -> Exp Statement*]]
      int label =  cg.getLabels(2);
      <LABEL> label <:>
      VALUE[[Exp]]
      <JZ><LABEL> label+1

      for(Statement s:Statement*)
            EXECUTE[[s]]()

      <JMP><LABEL> label
      <LABEL> label+1 <:>

EXECUTE[[DoWhileStatement: Statement -> Exp Statement*]]
      int label =  cg.getLabels(1);
      <LABEL> label <:>
      for(Statement s:Statement*)
            EXECUTE[[s]]()

      VALUE[[Exp]]
      <JNZ><LABEL> label
      <LABEL> label+1 <:>

EXECUTE[[ Invocation: Statement -> Variable Exp*]]
      VALUE[[ (Expression) Statement]]()
      if(Variable.Type.ReturnType != IO.VoidType)
            <POP> Variable.Type.ReturnType.Suffix();

EXECUTE[[Return: Statement -> Exp]] Param -> (FunDefinition)]]
      VALUE[[Exp]]()
      cg.convert(Exp.Type,FunDefinition.Type.ReturnType);
      <RET> FunDefinition.ReturnType.NumberBytes
      <,> FunDefinition.LocalBytes
      <,> FunDefinition.ParamBytes

VALUE[[IntLiteral: Exp -> IntConstant]]
      <PUSHI> Exp.VALUE

VALUE[[ChaLiteral: Exp -> CharConstant]]
      <PUSHB> Exp.VALUE

VALUE[[RealLiteral: Exp -> RealConstant]]
      <PUSHF> Exp.VALUE

VALUE[[Variable: Exp -> ID]]
      ADDRESS[[EXP]]()
      <LOAD> Exp.Type.Suffix()

VALUE[[Arithmetic: Exp1 -> Exp2 Exp3 ]]
```

```
      VALUE[[Exp2]]()
      cg.convert(Exp2.Type,Exp1.Type)
      VALUE[[Exp3]]()
      cg.convert(Exp3.Type,Exp1.Type)
      cg.arithmetic(Exp1.operator,Exp1.Type)
```

VALUE[[Comparison: Exp1 -> Exp2 Exp3]]

```
      supertype = Exp2.Type.SuperType(Exp3.Type)
      VALUE[[Exp2]]()
      cg.convert(Exp2.Type,supertype)
      VALUE[[Exp3]]()
      cg.convert(Exp3.Type,supertype)
      cg.comparison(Exp1.operator,supertype)
```

VALUE[[Cast: Exp1 -> CastType Exp2]]

```
      VALUE[[Exp2]]()
      cg.cast(Exp2.Type, CastType)
```

VALUE[[Logical: Exp1 -> Exp2 Exp3]]

```
      VALUE[[Exp2]]()
      VALUE[[Exp3]]()
      cg.logig(Exp1.operator)
```

VALUE[[UnaryNot: Exp1 -> Exp2]]

```
      VALUE[[Exp2]]()
      <NOT>
```

VALUE[[FieldAcces: Exp1 -> Exp2 ID]]

```
      ADDRESS[[Exp1]]()
      <LOAD>Exp1.Type.Suffix()
```

VALUE[[Indexing: Exp1 -> Exp2 Exp3]]

```
      ADDRESS[[EXP1]]()
      <LOAD>Exp1.Type.Suffix()
```

VALUE[[Invocation: Exp -> Variable Exp*]]

```
      int i=0;
      for(Expression e:Exp*)
            VALUE[[e]]()
            cg.convert(e.Type,Variable.Type.parameters[i++].Type)
      <CALL> Variable.Name
```

ADDRESS[[Variable: Exp -> ID]]

```
      if(Exp.Definition.scope == 0)
            <PUSHA> Exp.Definition.Offset
      else
            <PUSH BP>
            <PUSHI> Exp.Definition.Offset
            <ADDI>
```

```
ADDRESS[[ Indexing: Exp1 -> Exp2 Exp3 ]]
      ADDRESS[[Exp2]]()
      VALUE[[Exp3]]()
      <PUSH> Exp1.Type.NumberBytes
      <MUL>
      <ADD>

ADDRESS[[FieldAcces: Exp1 -> Exp2 ID]]
      ADDRESS[[Exp2]]
      <PUSH>Exp2.Type.get(ID).Offset
      <ADD>
```

EXTRA FEATURES

IMPLICIT PROMOTION

It's been developed the implicit promotion for: **assignment, arithmetic, comparison, function return type and parameters.**

OPERATORS

The following new operators have been developed: `++`, `--`, `+=`, `-=`, `*=`, `/=`, `&&=`, `||=`, `^=`. The ones that end with '=' act as statements. Others as both statements and expressions.

LOGICAL FUNCTIONS

The following new logical functions have been developed: `^ (XOR)`.

TERNARY OPERATOR

The new syntax is now supported: `( condition ) ? statement : statement ;` . Where if the condition is true the first statement is executed; the second otherwise.

DO WHILE STATEMENT

Now the grammar and the compiler support do while blocks as following: `do : { … } while ( condition ) ;` .

CODE OPTIMIZATION

The next mechanisms have been developed for code optimization: **Simple Dead Code Removal**. *In case that there is code after a return statement the compiler wont generate those instructions. This is done by checking if the current statement promotes to a return (Return | ifStatement with a return on the if and the else), and then skipping the following instructions within the scope of the return.*