Diseño de Lenguajes de Programación

Informe prácticas

David Covián Gómez

UO295168@uniovi.es

Ingeniería Informática del Software

Universidad de Oviedo

2024/2025

Contenido

Léxico del lenguaje expresado mediante expresiones regulares

1. INT_CONSTANT

(0|[1-9][0-9]*)

---

2. ID

[a-zA-Z_][a-zA-Z0-9_]*

---

3. SINGLE_COMMENT

//.*?(?:\n|$)

---

4. MULTI_COMMENT

/*.*?*/

---

5. CHAR_CONSTANT

'(\\n|\\t|\\[0-9]+|.)'

---

6. REAL_CONSTANT

((?:[1-9][0-9]*|\b)?\.[0-9]+|[1-9][0-9]*\.[0-9]*)([eE][+-]?[0-9]+)? |[1-9][0-9]*([eE][+-]?[0-9]+)

---

7. WHITE_SPACE

[ ]+

---

8. SALTO_LINEA

\n+

---

9. TABULADOR

\t+

---

10. RETORNO_CARRO

\r+

Formato de ANTLR:

```
INT_CONSTANT: [1-9][0-9]*
            | '0'
            ;
ID: [a-zA-Z_][a-zA-Z0-9_]*
;
SINGLE_COMMENT: '//'.*?('\n'|EOF) -> skip
    ;
MULTI_COMMENT: '/*'.*?'*/' -> skip
    ;
CHAR_CONSTANT: '\''.'\''
            | '\'\\'INT_CONSTANT'\''
            | '\'\\n\''
            | '\'\\t\''
            ;
fragment
DIGIT: [0-9]
    ;
fragment
MANTISA: INT_CONSTANT*'.'DIGIT+
        | INT_CONSTANT+'.'DIGIT*
        ;
fragment
EXPONENTE: [eE][+-]?INT_CONSTANT
        ;
REAL_CONSTANT: MANTISA EXPONENTE?
            | INT_CONSTANT EXPONENTE
            ;
WHITE_SPACE: ' '+ -> skip
            ;
SALTO_LINEA:'\n'+ -> skip
            ;
TABULADOR:'\t'+ -> skip
            ;
RETORNO_CARRO:'\r'+ -> skip
```

Sintaxis del lenguaje mediante una Gramática Libre de Contexto

CFG = (V_T, V_N, P, S) siendo:

V_T:

```
{
    'let', 'function', 'main', 'int', 'number', 'char', 'void',
    'log', 'input', 'if', 'else', 'while', 'return', 'as',
    '(', ')', '{', '}', '[', ']', ':', ';', ',', '.', '=',
    '+', '-', '*', '/', '%', '>', '>=', '<', '<=', '!=', '==',
    '&&', '||', '!', '-',
    ID, INT_CONSTANT, REAL_CONSTANT, CHAR_CONSTANT
},
```

V_N:

```
{
    Program,
    VariableDefinitionList, VariableDefinition, VariableDefinitionTail,
    FunctionDefinitionList, FunctionDefinition,
    ParameterListOptional, ParameterList, ParameterTail,
    ReturnType,
    MainFunction,
    Type,
    RecordFieldDefinitionList,
    SimpleType,
    StatementList, Statement,
    ExpressionListTail, InputExpressionListTail,
    ElseBlockOptional,
    Block,
    FunctionInvocation,
    ArgumentListOptional, ArgumentList, ArgumentTail,
    Expression
},
```

S: { Program },

P:

```
Program → VariableDefinitionList FunctionDefinitionList MainFunction

VariableDefinitionList → VariableDefinition VariableDefinitionList | ε
VariableDefinition → 'let' ID VariableDefinitionTail ':' Type ';'
VariableDefinitionTail → ',' ID VariableDefinitionTail | ε

FunctionDefinitionList → FunctionDefinition FunctionDefinitionList | ε
FunctionDefinition → 'function' ID '(' ParameterListOptional ')' ':' ReturnType
'{' VariableDefinitionList StatementList '}'

ParameterListOptional → ParameterList | ε
ParameterList → ID ':' SimpleType ParameterTail
ParameterTail → ',' ID ':' SimpleType ParameterTail | ε

ReturnType → SimpleType | 'void'
```

```
MainFunction → 'function' 'main' '(' ')' ':' 'void' '{' VariableDefinitionList
StatementList '}'

Type → SimpleType
     | '[' INT_CONSTANT ']' Type
     | '[' RecordFieldDefinitionList ']'

RecordFieldDefinitionList → VariableDefinition RecordFieldDefinitionList | ε

SimpleType → 'int' | 'number' | 'char'

StatementList → Statement StatementList | ε
Statement → 'log' Expression ExpressionListTail ';'
          | 'input' Expression InputExpressionListTail ';'
          | Expression '=' Expression ';'
          | 'if' '(' Expression ')' Block ElseBlockOptional
          | 'while' '(' Expression ')' Block
          | 'return' Expression ';'
          | FunctionInvocation ';'

ExpressionListTail → ',' Expression ExpressionListTail | ε
InputExpressionListTail → ',' Expression InputExpressionListTail | ε
ElseBlockOptional → 'else' Block | ε

Block → Statement
      | '{' StatementList '}'

FunctionInvocation → ID '(' ArgumentListOptional ')'

ArgumentListOptional → ArgumentList | ε
ArgumentList → Expression ArgumentTail
ArgumentTail → ',' Expression ArgumentTail | ε
Expression → '(' Expression ')'
           | Expression '[' Expression ']'
           | Expression '.' ID
           | '(' Expression 'as' Type ')'
           | '-' Expression
           | '!' Expression
           | Expression ('*' | '/' | '%') Expression
           | Expression ('+' | '-') Expression
           | Expression ('>' | '>=' | '<' | '<=' | '!=' | '==') Expression
           | Expression ('&&' | '||') Expression
           | FunctionInvocation
           | INT_CONSTANT
           | REAL_CONSTANT
           | CHAR_CONSTANT
           | ID
```

Formato de ANTLR:

```
program:
    (varDefinition | functionDefinition)* mainFunction EOF
;

varDefinition:
    'let' ID (',' ID)* ':' type ';'
;

functionDefinition:
    'function' ID '(' (ID ':' simpleType (',' ID ':' simpleType)*)? ')' ':'
(simpleType | 'void') '{' varDefinition* statement* '}'
;

mainFunction:
    'function' 'main' '(' ')' ':' 'void' '{' varDefinition* statement* '}'
;

type:
      simpleType
    | '[' INT_CONSTANT ']' type
    | '[' varDefinition+ ']'
;

simpleType:
    'int'
    | 'number'
    | 'char'
;

statement:
      'log' expression (',' expression)* ';'
    | 'input' expression (',' expression)* ';'
    | expression '=' expression ';'
    | 'if' '(' expression ')' block ('else' block)?
    | 'while' '(' expression ')' block
    | 'return' expression ';'
    | functionInvocation ';'
;

block:
      statement
    | '{' statement* '}'
;

functionInvocation:
    ID '(' (expression (',' expression)*)? ')'
;

expression:
      '(' expression ')'
    | expression '[' expression ']'
    | expression '.' ID
    | '(' expression 'as' type ')'
    | '-' expression
    | '!' expression
    | expression ('*' | '/' | '%') expression
    | expression ('+' | '-') expression
    | expression ('>' | '>=' | '<' | '<=' | '!=' | '==') expression
    | expression ('&&' | '||') expression
    | functionInvocation
    | INT_CONSTANT
    | REAL_CONSTANT
    | CHAR_CONSTANT
    | ID
;
```

```
// LEXICAL RULES
INT_CONSTANT: [1-9][0-9]* | '0';
ID: [a-zA-Z_][a-zA-Z0-9_]*;
SINGLE_COMMENT: '//' .*? ('\n' | EOF) -> skip;
MULTI_COMMENT: '/*' .*? '*/' -> skip;
CHAR_CONSTANT: '\'' . '\''
             | '\'\\' INT_CONSTANT '\''
             | '\'\\n\''
             | '\'\\t\'';

fragment DIGIT: [0-9];
fragment MANTISA: INT_CONSTANT* '.' DIGIT+ | INT_CONSTANT '.' DIGIT*;
fragment EXPONENTE: [eE][+-]? INT_CONSTANT;
REAL_CONSTANT: MANTISA EXPONENTE? | INT_CONSTANT EXPONENTE;

WHITE_SPACE: ' '+ -> skip;
SALTO_LINEA: '\n'+ -> skip;
TABULADOR: '\t'+ -> skip;
RETORNO_CARRO: '\r'+ -> skip;
```

Descripción de los nodos del Árbol Abstracto (AST) mediante una Gramática Abstracta.

```
IntLiteral: expression -> INT_CONSTANT
CharLiteral: expression -> CHAR_CONSTANT
NumberLiteral: expression -> NUMBER_CONSTANT
Variable: expression -> ID
Cast: expression1 -> expression2 type
FieldAccess: expression1 -> expression2 ID
Arithmetic: expression1 -> expression2 (+|-|*|/) expression3
Comparison: expression1 -> expression2 (>=|<=|>|<) expression3
ArrayAccess: expression1 -> expression2 expression3
Logic: expression1 -> expression2 (&& | ||) expression3
UnaryMinus: expression1 -> expression2
UnaryNot: expression1 -> expression2
Write: statement -> expression
Read: statement -> expression
While: statement1 -> expression statement2*
If: statement1 -> expression statement2*
Assignment: statement1 -> expression1 expression2
Invocation: expression1 -> expression2 expression3*
Return: statement -> expression
FunctionDefinition: definition -> ID type statement*
```

Descripción de la fase de Comprobación de Tipos del análisis semántico mediante una Gramática Atribuida

```
AG = (G,A,R) siendo:

G:
    (1) IntLiteral: expression -> INT_CONSTANT
    (2) CharLiteral: expression -> CHAR_CONSTANT
    (3) NumberLiteral: expression -> NUMBER_CONSTANT
    (4) Variable: expression -> ID
    (5) Cast: expression1 -> expression2 type
    (6) FieldAccess: expression1 -> expression2 ID
    (7) Arithmetic: expression1 -> expression2 (+|-|*|/) expression3
    (8) Comparison: expression1 -> expression2 (>=|<=|>|<) expression3
    (9) ArrayAccess: expression1 -> expression2 expression3
    (10) Logic: expression1 -> expression2 (&& | ||) expression3
    (11) UnaryMinus: expression1 -> expression2
    (12) UnaryNot: expression1 -> expression2
    (13) Write: statement -> expression
    (14) Read: statement -> expression
    (15) While: statement1 -> expression statement2*
    (16) If: statement1 -> expression statement2*
    (17) Assignment: statement1 -> expression1 expression2
    (18) Invocation: expression1 -> expression2 expression3*
    (19) Return: statement -> expression
    (20) FunctionDefinition: definition -> ID type statement*
A:
    {expression.type, statement.returnType} ambos de dominio Type
R:
    (1) expression.type = IntType
    (2) expression.type = CharType
    (3) expression.type = NumberType
    (4) expression.type = expression.definition.type
    (5) expression1.type = expression2.type.canBeCastTo(type)
    (6) expression1.type = expression2.type.dot(ID)
    (7) expression1.type = expression2.arithmetic(expression3.type)
    (8) expression1.type = expression2.comparison(expression3.type)
    (9) expression1.type = expression2.type.squareBrackets(expression3.type)
    (10) expression1.type = expression2.type.logic(expression3.type)
    (11) expression1.type = expression2.type.arithmetic()
    (12) expression1.type = expression2.type.logic()
    (13) expression.type.mustBeBuiltIn()
    (14) expression.type.mustBeBuiltIn()
    (15) expression.type.mustBeLogical()
    (16) expression.type.mustBeLogical()
    (17) expression1.type.mustBePromotes(expression2.type)
    (18) expression1.type = expression2.type.parenthesis(expression3*)
    (19) expression.type.mustBePromotes(statement.returnType)
    (20) statement*.forEach(s => s.returnType = type.returnType)
```

Descripción de la fase de Selección de Código mediante una Especificación de Código

Execute:

```
execute [[Program: program -> definition*]]():
    for (Definition def: definition*)
       if(def instanceof VarDefinition)
           execute[[def]]()
    <' * Invocation to the main function>
    <call main>
    <halt>
    for (Definition def: definition*)
       if(def instanceof FunctionDefinition)
           execute[[def]]()
```

```
execute [[VarDefinition: definition -> type ID]]():
   <' *> type ID <( offset > definition.offset <)>
```

```
execute [[FunctionDefinition: definition -> ID type statement*]]():
   ID<:>
       <enter> definition.localBytesSum
       <' * Parameters>
       type.getArguments().forEach(p -> execute[[p]]())
       <' * Local Variables>
       statement*.forEach(s -> execute[[s]]())
```

```
execute[[Write: statement -> expression]]():
   value[[expression]]()
   <out> expression.getType.suffix()
```

```
execute[[Read: statement -> expression]]():
   address[[expression]]()
   <in> expression.getType.suffix()
   <store> expression.getType.suffix()
```

```
execute[[Assignment: statement -> expression1 expression2]]():
   <#line  > statement.getLine()
   <'  * Assignment>
   address[[expression1]]()
   value[[expression2]]()
   <store> expression1.getType().getSuffix()
```

```
execute [[If: statement1 -> expression statement2* statement3*]]():
  <#line  > statement1.getLine()
  <'  * If>
  String elsePart = cg.getLabel();
  String end = cg.getLabel();
  value[[condition]]();
  <jz> elsePart
  statement2*.forEach(s -> execute[[s]]());
  <jpm> end
  elsePart<:>
```

```
  statement3*.forEach(s -> execute[[s]]());
  end<:>
```

```
execute[[Invocation: statement -> expression1 expression2*]]():
  <#line  > statement.getLine()
  <'  * Invocation>
  value[[(Expression) statement]]()
       if (!statement.getType() instanceof VoidType) {
           <pop> statement.getType().getSuffix();
       }
```

```
execute[[Return: statement -> expression]](FunctionDefinition fd):
    <#line> statement.getLine()
    <' * Return>
    value[[expression]]()
    int bReturn = ((FunctionType)
fd.getType()).getReturnType().getSize();
    int bLocals = fd.getLocalBytesSum();
    int bParams = 0;
    for (VarDefinition v : ((FunctionType)
fd.getType()).getArguments())
       bParams += v.getType().getSize()
    <ret> bReturn <,> bLocals <,> bParams
```

```
execute[[While: statement1 -> expression statement2*]]():
    String condition = cg.nextLabel();
    String end = cg.nextLabel();
    <label> condition <:>
    value[[expression]]()
    <jz> end
    statement2*.forEach(stmnt -> execute[[stmnt]])
    <jmp label> condition
    <label> end <:>
```

Value:

```
value[[Variable: expression -> ID]]():
   address[[expression]]
   <load> expression.getType().suffix()
```

```
value[[Arithmetic: expression1 -> expression2 (+|-|*|/|%)
expression3]]():
    value[[expression2]]()
    cg.convertTo(expression2.getType(), expression1.getType())
    value[[expression3]]()
    cg.convertTo(expression3.getType(), expression1.getType())
    cg.arithmetic(expression1.getOperation(), expression1.getType())
```

```
value[[ArrayAccess: expression1 -> expression2 expression3]]():
    address[[expression1]]
    <load> expression1.type.suffix()
```

```
value[[Cast: expression1-> type expression2]]():
    value[[expression2]]
    cg.convertTo(expression2.getType(), type);
```

```
value[[CharLiteral: expression -> CHAR_CONSTANT]]():
    <pushb> CHAR_CONSTANT
```

```
value[[Comparison: expression1 -> expression2 (==|!=|>=|<=|>|<)
expression3]]():
    value[[expression2]]()
    cg.convertTo(expression2.getType(), expression1.getType())
    value[[expression3]]()
    cg.convertTo(expression3.getType(), expression1.getType())
    cg.comparison(expression1.getOperator(), expression1.getType())
```

```
value[[Invocation: expression1 -> expression2 expression3*]]():
    for (int i = 0; i < expression3*.size(); i++) {
        expression3*.get(i).accept(this, p);
        cg.convertTo(expression3*.get(i).getType(),
             ((FunctionType)
expression2.getDefinition().getType()).getArguments().get(i).getType()
);
    }
    <call> expression2
```

```
value[[IntLiteral: expression -> INT_CONSTANT]]():
    <pushi> INT_CONSTANT
```

```
value[[Logical: expression1 -> expression2 (&&|||) expression3]]() =
    value[[expression2]]()
    value[[expression3]]()
    cg.logical(expression1.getOperation())
```

```
value[[NumberLiteral: expression -> NUMBER_CONSTANT]]():
    <pushf> NUMBER_CONSTANT
```

```
value[[StructAccess: expression1 -> expression2 ID]]():
    address[[expression1]]
    <load> expression1.getType().getSuffix()
```

```
value[[UnaryMinus: expression1 -> expression2]]():
    value[[expression2]]()
    cg.convertTo(expression2.getType(), expression1.getType())
    <pushi> -1
    cg.convertTo(IntType.type, expression1.getType())
    <mul> expression1.getType().getSuffix()
```

```
value[[UnaryNot: expression1 -> expression2]]():
    value[[expression2]]()
    cg.logical(expression1.getOperand())
```

Address:

```
address[[Variable: expression -> ID]]():
   if(expression.getDefinition().getScope()==0)
       <pusha> expression.getDefinition().getOffset()
   else
       <push bp>
       <pusha> expression.getDefinition().getOffset()
       <addi>
```

```
address[[ArrayAccess: expression1 -> expression2 expression2]]():
    address[[expression2]]
    value[[expression3]]
    <pushi> expression1.getType().getSize()
    <muli>
    <addi>
```

```
address[[FieldAccess: expression1 -> expression2 ID]]():
    address[[expression2]]
    <pushi>
expression2.getType().getField(expression1.getName()).getOffset()
    <addi>
```