Language and Traslator - Parser Report

Francesca Daniele
Carmelo Gugliotta
Marco Leto

A.Y. 2023/24

1 Parser Report

Our Parser implementation respect and satisfy all the "project statement" requirments. The main class is "Parser.java" that cointain the method `public Program getAST()` and is responsible the creation of the **Abstract Syntax Tree**, composed by ASTNode. For this reason has been created an abstract class `ASTNode` that is extended by all the class that codify the program constructs.
The parser produce it processing the symbols obtained from the Lexer.
The method is characterized by 3 while loop:

- the first one to parse the Constant
- the second one to parse the Struct
- the third one, always true, that permise to parse Global Variable or Procedure indipendently from the statement order

A pseudocode is shown:

```
public Program getAST() throws IOException, ParserException {

        while("Final" is the next symbol){
            //Parse a Constant
        }

        while("Struct" is the next symbol){
            //Parse a Struct
        }

        while(A Type or an Identifier || Def is the next symbol){
            //Parse
        }

        return program;
```

The next symbol is contained on the `private Symbol lookahead` and the method `private Symbol consume(Token token)` is used in all the class to consume the symbol if and just if it respect the token passed as parameter, otherwise an exception will be called. Furthermore the `nextSymbol` is putted into `lookahead`.

2 ExpressionStatement

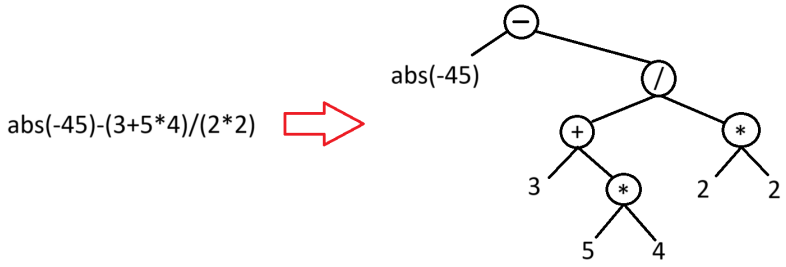
The entire code is characterized by the presence of expressions, for this reason the method `parseExpression()` has been created into the Parser class, and codify the following rules:

```
Exp -> AndOrExp

AndOrExp ->  ComparaisonExp |AndOrExp AndOrOp ComparaisonExp

ComparisonExp ->  AdditiveExp | ComparasionExp ComparisonOp AdditiveExp

AdditiveExp -> MultiplicativeExp | AdditiveExp AdditiveOp MultiplicativeExp

MultiplicativeExpression -> StructAccess | MultiplicativeExp MultiplyOp

StructAccess -> Factor StructAccess' |
                StructAccess [ Exp ] . StructAccess StructAccess'

StructAccess' -> . Factor StructAccess' | epsilon
```

This rules permises to well parse the Expression respecting the priority of the operators requested in the statement, and are traduced into a series of nested methods:

```
parseExpression()
    ↓
    parseAndOrExpression()
        ↓
        parseComparisonExpression()
            ↓
            parseAdditiveExpression()
                ↓
                parseMultiplyExpression()
                    ↓
                    parseArrayAccessOrStructAccess()
```

the return is a Binary Expression Tree, where the leaf nodes are the values and the inner nodes are object that have to be evaluated(operators, function,ecc):



Defined the expression, then we create different class that utilize it for the code implementation:

```
VariableAssigment -> IdentifierRef = Exp; | IdentifierRef IncrementOperator;

IdentifierRef -> VariableRef | VariableRef . IdentifierRef | VariableRef [ Exp ]

VariableInstantiation -> Type VariableRef = Expr;

VariableDeclaration -> Type VariableRef ; | VariableInstantiation ;
```

Note that a `VariableRef . IdentifierReference` is a structAccess and `VariableRef [ Exp ]` is a StructAccess, where `VariableReference` represent an identifier String (ex. in `x[3]` x is a variable reference).

3 Data Type

In the following report section we are gonna discuss about main parser implementation, showing the grammar about our code. While the global constant could just have type java base, couse no struct has not yet been declared, i the others code section the Type is defined by the rule:

```
Type -> BasedType | IdentifierType | Void
```

where:

```
IdentifierType -> StructType

IdentifierType[] -> ArrayStructType

BasedType -> BaseType

BasedType[] -> ArrayType
```

4 Struct, Global Variable and Procedures

Obtained an essential view of expressions and the data types, we has been able to design all the others structurd:

```
Struct -> struct Identifier { VariableDeclarationInStruct}

VariableDeclarationInStruct -> Type Identifier ; |
                               VariableDeclarationInStruct |
                               epsilon

GlobalVariable -> Type Identifier = Expression ;

Procedure -> Type Identifier ( VariableDeclarationInProcedure ) { Block }

VariableDeclarationInProc -> Type Identifier |
                             Type Identifier, VariableDeclarationInProc|
                             epsilon
```

5 Block

The block is one of the most complex structure in our implementation, as the basic structure of the program it admits all the Statement previusly discussed in any order, to which the loops and if-else constructet is added. The method `parseBlock()` utilize a while loop that terminate when a "} is finded and follow the rules:

```
Block -> {Statements; }

Statements -> Statement | Statement Statements

Statement -> VariableDeclaration; |
             VariableAssigment; |
             FunctionCall; |
             ForStatement |
             WhileStatement |
             IfStatement |
             ReturnStatement
```

where:

```
ForStatement -> for (condition_1; condition_2; condition_3 ) {Block}

condition_1 -> VariableAssignment | epsilon

condition_2 -> Expression | epsilon

condition_3 -> Expression | epsilon

WhileStatement -> while (Expression) {Block}

IfStatement -> if (Expression) {Block} | if (Expression) {Block} else {Block}

ReturnStatement -> return Expression; | return;
```

Is possible to notice that the cicle statements calls inside them other Block, that makes it a deep structure.