DEPARTMENT OF COMPUTER SCIENCE, TECHNISCHE UNIVERSITÄT KAISERSLAUTERN
COMPILERS AND LANGUAGE PROCESSING TOOLS - SS17

A Parser and Lexer for Minijava

Exercise 2

Group 03
Daniele Gadler, Gopal Praveen
Alka Scaria, Stephen Banin Payin

May 13, 2017

OVERVIEW

In the present report, we describe the parser we built for Minijava. More specifically, in the 'Features' section we explain what our parser is capable of doing, whereas in the 'Technical Description', we provide a deeper description of key components of the Parser.
The constructed Parser passes all tests provided in the template as well as test cases laid out by the group members. It is also thoroughly documented by Javadoc.

FEATURES

- **Abstract Syntax Tree Construction**: The parser is capable of parsing java files according to the grammar given as part the exercise and representing a Java file's structure as an AST. Picture 0.1 shows how the parader is able to parse nested structures and represent them as an AST.

```
Program(MainClass(ArrayOk, args, Block(VarDecl(TypeIntArray, arr))), ClassDeclList())
```

Figure 0.1: Resulting AST for a java class containing an array declaration in the main method.

- **Invalid statements recognition**: The parser disallows the parsing of expressions that are not allowed in Java within a statement. This is carried out for stray expressions (Statement –> Exp;) and assignment expressions (Statement –> Exp = Expr;).
  Upon encountering invalid statements, the parser stops the parsing process and raises appropriate syntax errors.
  For example, when parsing the following expression contained in a valid method "2 = id", the following syntax error will be raised:

  The left-hand side of an assignment expression cannot be a number.

TECHNICAL DESCRIPTION

TASK 1: GRAMMAR IMPLEMENTATION, OPERATOR PRECEDENCE AND ASSOCIATIVITY

We implemented the grammar for parsing Minijava as described in the exercise's grammar specifications: particular attention needed to be paid to the implementation of following elements:

- **Kleene Star (*)**: We implemented Minijava elements that may appear zero or more times (e.g: ClassDecl, MemberDecl, BlockStatement, ExprRest) as lists.
  We implemented lists' wrapping of elements in a recursive manner: firstly, we check if an element of the list is followed by a list of elements and parse the single element found. After the element is parsed, it is added to the list of elements parsed so far and the other elements in the list are parsed in the same recursive manner. If a list appears to empty, an empty list is returned.

- **Optional (?)**: We implemented optional Minijava elements that may appear zero or one time (e.g: ExprList, ParaList) as lists. If such a list appears to contain valid element(s), then it is parsed. Alternatively, if such a list is recognized to be empty, then an empty list is returned. The main difference between 'optional' lists and 'Kleene star' lists lies in the fact that optional lists are parsed just one time, no matter they are empty or contain something.

We implemented operators' precedence and associativity by grouping expressions' operations' declarations into the same grammar declaration. By doing so, operators' precedence declarations can actually have an effect on the associativity and the order of operations. Consequently, we declared the following precedence rules for Java operators, as described by [1].

```
precedence left EQ;
precedence left AND;
precedence left EQUALS;
precedence left LESS;
precedence left  PLUS, MINUS;
precedence left  TIMES, DIV;
precedence left LBRACKET, RBRACKET, DOT, LPAREN, RPAREN;
```

TASK 2 - INVALID STATEMENTS RECOGNITION

We implemented invalid statements' recognition by making use of the visitor pattern in the MJInvalidStatement class. By passing the AST constructed in Task 1 to the "acceptProgram" method, the whole tree is traversed with the purpose of detecting statements that are invalid in Java.
By overriding the visit method for **assignment statements** (e.g: Statement –> Expr = Expr;), we raise appropriate syntax errors if the left-hand side of assignment operations contains: a number, a binary expression, 'this', 'null', an array length expression, a unary expression (!Expr or -Expr) or a boolean constant.
We also overrode the visit method for **'stray' expressions** (e.g: Statement –> Expr;). The only stray expressions allowed are 'MethodCall' and 'New Object Creation'. In all other cases, a syntax error is raised.

REFERENCES

[1] Precedence and associativity of Java operators. *http://introcs.cs.princeton.edu/java/11precedence/*. Accessed on 10th May 2017.