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

Type and Name Analysis for Minijava

Exercise 3

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

May 28, 2017

OVERVIEW

In the present report, we describe the type and name analyzer built by our group for Minijava. More specifically, in the 'Features' section we explain what our analyzer is capable of doing, whereas in the 'Technical Description', we provide a deeper description of key components of the Analyzer.
The constructed type and name analyzer is thoroughly commented with Javadoc and passes all tests provided for the 'Name' analysis and passes most of the tests for the 'Type' analysis . Due to time constraints, the group members did not manage to make the analyzer pass all test cases.

FEATURES

- **Class Extension checking**: The name analyzer checks

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 (*)**: The Minijava elements that may appear zero or more times (e.g: ClassDecl, MemberDecl, BlockStatement, ExprRest) as lists.
  Lists' wrapping of elements was implemented in a recursive manner: firstly, it was checked 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 (?)**: The 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.

The operators' precedence and associativity were implemented 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.