apl. Prof. Dr. Thomas Noll
Sebastian Junges, Christoph Matheja, Matthias Volk

Compiler Construction 2017
— Programming Exercise 2 —

Upload in L2P until May 23rd before the exercise class.

General Remarks

- Implement the methods indicated by `TODO` but do not modify the signatures of the provided methods. You are however allowed to add your own methods, data structures and classes in the code.
- Please document essential parts of your code properly such that it is possible to grasp your ideas. Although the code will be graded mostly by functionality, your comments will help us to clarify whether a bug is a conceptual mistake or just a small error.
- The practical part will be implemented in Java 8. You may use the standard library to solve the programming tasks. Other libraries are not allowed.
- Submitted code which does not execute results in 0 points.
- Your solutions to the practical programming exercise should be uploaded via L2P as a zip file.
- If you have questions regarding the exercises and/or lecture, feel free to post in the L2P forum, write us an email at cb2017@i2.informatik.rwth-aachen.de or visit us at our office.

Programming Exercise 1 (5 Points)

The goal of this exercise is to build our own lexer which transforms an input string into a list of symbols.

Hint: as before we provide a framework which can be downloaded from the course webpage or the L2P.

- Implement `lexer.BacktrackingDFA.doStep(char)`, the method that performs a step in the product automaton of all DFAs and returns the recognized token.
- Implement `lexer.BacktrackingDFA.run(String)`, the method that, given an input string, performs the steps of the backtracking automaton as discussed in the lecture and returns a list of symbols. In case of a lexer error throw the corresponding `lexer.LexerException`.

Test your implementation! For example, given the following input

```
1  /* A random walk */
2  int x = 10;
3  int s = 0;
4  while ( x > 0 ) {
5      int b = read() % 2; // randomness by user input
6      if (b == 1) {
7          x = x + 1;
8      } else {
9          x = x - 1;
10     }
11     s++;
12 }
13 write("I stopped walking after: ");
14 write(s);
15 write(" steps");
```

your implementation should generate a list of symbols like this:

```
(INT, int), (ID, x), (ASSIGN, =), (NUMBER, 10), (SEMICOLON, ;),
(INT, int), (ID, s), (ASSIGN, =), (NUMBER, 0), (SEMICOLON, ;),
(WHILE, while), (LPAR, (), (ID, x), (GT, >), (NUMBER, 0), (RPAR, )), (LBRACE, {),
(INT, int), (ID, b), (ASSIGN, =), (READ, read), (LPAR, (), (RPAR, )), (MOD, %),
    (NUMBER, 2), (SEMICOLON, ;),
(IF, if), (LPAR, (), (ID, b), (EQ, ==), (NUMBER, 1), (RPAR, )), (LBRACE, {),
(ID, x), (ASSIGN, =), (ID, x), (PLUS, +), (NUMBER, 1), (SEMICOLON, ;),
(RBRACE, }), (ELSE, else), (LBRACE, {),
(ID, x), (ASSIGN, =), (ID, x), (MINUS, -), (NUMBER, 1), (SEMICOLON, ;),
(RBRACE, }),
(ID, s), (INC, ++), (SEMICOLON, ;),
(RBRACE, }),
(WRITE, write), (LPAR, (), (STRING, "I stopped walking after: "), (RPAR, )),
    (SEMICOLON, ;),
(WRITE, write), (LPAR, (), (ID, s), (RPAR, )), (SEMICOLON, ;),
(WRITE, write), (LPAR, (), (STRING, " steps"), (RPAR, )), (SEMICOLON, ;), (EOF, $)
```