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

Compiler Construction 2016

— Series 1 —

Hand in until April 26th before the exercise class.

General Remarks This course will be accompanied by a series of practical assignments with the goal to build our own compiler for the language *WHILE*. The *WHILE* language captures (integer) variable declarations, assignments, arithmetic operations, conditional branches, loops, basic I/O (read and write) and Java-style comments. Please consider the following remarks regarding implementation assignments:

- All implementation tasks must be done in Java.
- Please mail your code as a ZIP-file to cb2016@i2.informatik.rwth-aachen.de.
- Exercise sheets will be accompanied by a small framework including predefined classes and method declarations. Your task usually is to implement these methods. Please do not modify the signatures of the provided methods.
- Submitted code which does not execute results in 0 points. Therefore make sure you submit everything that you have used to run your code.
- You may use the Java standard library. However, other third party libraries are not allowed.
- 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.

Exercise 1 (2 Points)

As an example of the language *WHILE*, consider the following program.

```
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");
```

(a) Give a complete list of the symbol classes and corresponding tokens needed for the lexical analysis of our programming language *WHILE*. Recall that *WHILE* captures (integer) variable declarations, assignments, arithmetic operations, conditional branches, loops, basic I/O (read and write) and Java-style comments.

(b) Decompose the if-statement (lines 6-10) into a sequence of lexemes and translate each lexeme into a symbol.

Exercise 2 (3 Points)

(a) Provide regular expressions for comments (`//` and `/* ... */`) in *WHILE*. Please keep in mind that `*` as well as `/` may also occur inside of comments. Thus `/*foo * bar / */` is a valid comment. Note that single-line comments are terminated by a newline symbol `\n` (Linux), carriage return `\r` (Mac OS) or `\r\n` (Windows).

(b) Provide a regular expression capturing numbers in scientific notation, e.g. `-17.42e+23`. To be more precise, a number in scientific notation consists of a floating point number with an optional sign followed by `e` followed by an integer number which may be preceded by an optional sign. In case the floating point number is an integer, the dot may be omitted. Furthermore, if it is less than one, an initial zero may be omitted. Thus, `.3e-8` and `+42e0` are valid numbers in scientific notation.

(c) Derive *one* NFA that accepts all words which are either comments or numbers in scientific notation as defined in the previous tasks.

(d) Solve the simple matching problem on the NFA from above for the input string `-17.42e+23`.

(e) For a regular expression r and natural numbers n, m with $n \leq m$ we define a *repetition operator* $r^{[n,m]}$ to denote n to m occurrences of r. For instance, $L(a^{[1,3]}) = \{a, aa, aaa\}$. Prove or disprove that for each regular expression with repetition operators, there exists a regular expression without repetition operators.

Exercise 3 (5 Points)

In this exercise we make the first steps towards building a lexer. The task of a lexer is to read an input string and return a sequence of symbols. For now, we start by building deterministic finite automata that recognise particular tokens.

On the course webpage (https://moves.rwth-aachen.de/teaching/ss-16/cc/), we provide you with a framework which contains the essential class definitions and method declarations for this task. The easiest way to work with that framework is to simply import the files as an existing project into Eclipse. Parts which need implementation are marked with `TODO`.

(a) Implement the class `AbstractDFA.java` representing an arbitrary DFA with its states and transitions. You may use `Pair.java` in the `helper` module.

(b) Implement the class `DFA.java` which is instantiated with a string. This class should represent a DFA that recognises exactly the given string.

(c) Implement the class `CommentDFA.java` that recognises single-line and multi-line comments.

For testing, you find several small examples in the `test` directory. You can compile the program with

```
$javac -d bin -sourcepath src src/Main.java
```

and then run the examples with

```
$java -cp bin Main tests/test1.txt
```

The results on the examples should be:

```
$java -cp bin Main tests/test1.txt
input: while
WHILE: true
COMMENT: false

$java -cp bin Main tests/test2.txt
input: While
WHILE: false
COMMENT: false
```

```
$java -cp bin Main tests/test3.txt
input: /**while*/
WHILE: false
COMMENT: true

$java -cp bin Main tests/test4.txt
input: /* */*/
WHILE: false
COMMENT: false

$java -cp bin Main tests/test5.txt
input: //foo

WHILE: false
COMMENT: true
```