apl. Prof. Dr. Thomas Noll
Philipp Berger, Matthias Volk

Compiler Construction 2018/19
— Programming Exercise 3 —

Upload in L2P until November 26th before the exercise class.

General Remarks

You have **2 weeks** to solve this exercise.

The aim of this exercise sheet is to make you familiar with the ANTLR (`http://www.antlr.org`) parser builder library. It is a widespread library to write parsers for a range of programming languages. Please consider the following remarks regarding implementation assignments:

- 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 ANTLR exercise will be implemented in Java 8 **and ANTLR 4.7.1**. You may use the standard library **(but not java.util.regex)** to solve the programming tasks. Other libraries are not allowed.
- Submitted code which does not execute results in 0 points. Therefore make sure you submit everything that you have used to run your code.
- Your solutions to the practical programming exercise should be uploaded via L2P as a zip file.
- **Important:** Make sure that your zip file contains the "RegexGrammar" folder, with that folder containing at least the "src" and "lib" folders with all necessary files for compiling your code. Example: "/RegexGrammar/src/RegexGrammarRunner.java" should be a valid path in your archive.
- There are several possible solutions, and this is a way to get to know ANTLR. **Feel free to discuss in the L2P forum.** There, we also post a couple of resources.

Programming Exercise 1 (20 Points)

You are asked to build a parser for regular expressions extended by sets of symbols. In this exercise sheet, you have to design a parser for basic regular expressions, consisting of alternatives ($+$), concatenation (two words written together) and the Kleene star ($*$). An ε is denoted by the underscore _. We assume Latin non-capital characters (a–z) as alphabet. Furthermore, you can write concrete sets in the expressions: $\{a, b, c\}$ which means $a + b + c$. On sets, we allow set union ($|$), set intersection ($\&$) and set minus ($-$).

The following is a list of example regex with a word which is either in or not in the language.

Regex	word	member
$c(de + ed)^*$	$cdeeded$	`true`
$ab^*c(de + ed)^*$	$acdeeded$	`true`
$ab^*c(de + ed)^*$	$aabbcdeeded$	`false`
$c(\{d\}\|\{e\})^*$	$cddd$	`true`
$c(\{d\} - \{d, e\})^*$	ce	`false`

The input for this table and more can be found in the file `RegexGrammar/tests/test.txt`.

The operator precedence should be $* \succ$ concatenation $\succ + \succ | \succ \& \succ -$, where $*$ binds strongest.

Your task is to write a tool which **decides membership automatically**. We prepared a main method and an automata class for NFA. Parts which need implementation are marked with `TODO`.

In particular, we suggest to:

- Install the *antlr4ide* (`https://github.com/antlr4ide/antlr4ide`) for Eclipse, and follow the instructions to create a project for ANTLR 4 in Eclipse, if you want to use an IDE. Otherwise, see the supplied `compile.{sh,bat}` scripts for compiling by hand. When not using the IDE, do *not* forget to recompile your grammar after changing it!
- If you use Eclipse you can import the provided sources as a project into Eclipse. Afterwards, you have to set the correct path to `antlr-4.7.1-complete.jar` in the project `Properties > ANTLR 4 > Tool`.
- Design a parser, and realise a print-function for the abstract syntax tree. This helps debugging, and you will get to know ANTLR. You can implement the print function in the `getAutomaton()` method of class `RegexGrammarRunner`. You can create additional files and classes if necessary.
- Write a regex-to-NFA method in the `getAutomaton()` method of class `RegexGrammarRunner`. Again, you can create additional files and classes if necessary.
- Implement the `accept()` method in the class `Automaton`.
- Test your implementation. You can either run the program from Eclipse or from the commandline. For compiling, see the scripts `compile.sh` (Linux) and `compile.bat` (Windows, needs path to your `javac.exe`, see top of file).
  Execute with:

  ```
  $ java -cp build/classes:lib/antlr-4.7.1-complete.jar RegexGrammarRunner
  ```

  If no argument is given the regex must be given on the commandline. End the regex with a linebreak and press Ctrl+D to end the input:

  ```
  Input regex:
  ab*
  (CTRL+D)
  ```

  Then the corresponding automaton is constructed and printed.

  You can also provide a list of regex/word pairs and check for the membership:

  ```
  $ java ... RegexGrammarRunner "a*" "aa" "b*" "ab"
  'aa' is a member of 'a*'? true
  'ab' is a member of 'b*'? false
  ```

  The third option is to provide a file as argument containing a list of regex/word pairs:

  ```
  $ java ... RegexGrammarRunner tests/test.txt
  'cdeeded' is a member of 'c(de+ed)*'? true
  'acdeeded' is a member of 'ab*c(de+ed)*'? true
  ...
  ```