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

# Compiler Construction 2016

# — Series 4 —

Hand in until May 31st before the exercise class.

---

## General Remarks

- Follow the naming convention for the zip file: `ex4_MATRNO1_MATRNO2_MATRNO3` and include the complete framework provided to you via our webpage.
- It is allowed to hand in your solutions for the theoretical part via email **as a separately attached PDF file**.
- Please hand in your solutions in groups of 3 or 4.

## Exercise 1 (3 Points)

Show that every regular language can be generated by a $LL(1)$-grammar.

## Exercise 2 (4 Points)

Consider the grammar $G = (N, \Sigma, P, start)$ covering some boolean expressions:

- $N := \{start, guard, rel\}$
- $\Sigma := \{AND, OR, ID, EQ, LEQ\}$
- $$\begin{array}{lcl} start & \to & guard \\ guard & \to & rel \mid guard\ AND\ guard \mid guard\ OR\ guard \\ rel & \to & ID\ EQ\ ID \mid ID\ LEQ\ ID \end{array}$$

(a) Construct $NTA(G)$.
(Either give a transition table or depict the automaton and specify what the edge labelling means. Do not forget to give a numbering to the grammar rules.)

(b) Provide a run of $NTA(G)$ on the input $ID\ EQ\ ID\ AND\ ID\ LEQ\ ID$.

(c) Construct an equivalent grammar $G'$ with $G' \in LL(1)$.

(d) Specify the deterministic top-down parsing automaton $DTA(G')$.
(Again, either give a transition table as in the lecture or depict the automaton and specify what the edge labelling means. As before, do not forget to give a numbering to the grammar rules of $G'$.)

(e) Provide a run of $DTA(G')$ on the input $ID\ EQ\ ID\ AND\ ID\ LEQ\ ID$.

# Exercise 3 (3 Points)

After finishing the lexer, the next step is to implement a parser. The goal of this exercise is to build a recursive-descent parser which transforms the list of symbols (returned from the lexer) into a list of grammar rules. The grammar is as follows and covers assignments:

1. $start \rightarrow assignment$ `SEMICOLON EOF`
2. $assignment \rightarrow$ `INT ID ASSIGN` $expr$
3. $expr \rightarrow$ `ID` $subexpr$
4. $expr \rightarrow$ `NUMBER` $subexpr$
5. $expr \rightarrow$ `LPAR` $expr$ `RPAR`
6. $expr \rightarrow$ `READ LPAR RPAR` $subexpr$
7. $subexpr \rightarrow$ `PLUS` $expr$
8. $subexpr \rightarrow$ `MINUS` $expr$
9. $subexpr \rightarrow$ `TIMES` $expr$
10. $subexpr \rightarrow$ `DIV` $expr$
11. $subexpr \rightarrow$ `MOD` $expr$
12. $subexpr \rightarrow \varepsilon$

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

Implement the methods in `parser.RecursiveDescentParserAssignment` for the given grammar. The superclass `parser.RecursiveDescentParser` offers useful methods for getting the next token (`next()`), printing a grammar rule (`print(id)`) and throwing an error (`printError(msg)`).

Test your implementation! For example, given the following input

```
int b = read() % 2;
```

your implementation should generate a list of grammar rules like this:

```
1, 2, 6, 11, 4, 12
```