apl. Prof. Dr. Thomas Noll

Friedrich Gretz, Souymodip Chakraborty

Exercise 1 (CFL): (3 Points)

Give context-free grammars to generate the following sets:

a) The set of strings over alphabet $\{a, b\}$ such that each prefix of a string has at least as many a's as b's.

b) The set of strings over alphabet $\{a, b\}$ such that the number of a's is twice the number of b's.

c) The set of strings over alphabet $\{a, b\}$ of even positive length such that the string is NOT of the form ww.

Exercise 2 (NTA): (2 Points)

Prove soundness of the top down analysis automaton $NTA(G)$ for a grammar $G = \langle N, \Sigma, P, S\rangle$, i.e. show that for all $w \in \Sigma^*$ and all $z \in \{1, \ldots, |P|\}^*$:

$$(w, S, \varepsilon) \vdash^* (\varepsilon, \varepsilon, z) \quad \text{implies} \quad S \underset{l}{\overset{z}{\Rightarrow}} w$$

Exercise 3 (Ambiguous CFG): (2 Points)

Consider the grammar $G = (N, \Sigma, P, E)$ for arithmetic expressions.

- $N := \{E\}$
- $\Sigma := \{+, *, (,), \mathbf{id}\}$
- $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

a) Show that the grammar is ambiguous.

b) Construct an equivalent unambiguous grammar for all arithmetic expressions with no redundant parentheses. A set of parentheses is redundant if its removal does not change the expression, e.g., the parentheses are redundant in $\mathbf{id} + (\mathbf{id} * \mathbf{id})$.

Exercise 4 (Lexer Implementation): (3 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!

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.
Test your implementation! For example, given the following input

```
1   /* GCD-Computation of x and y
2       w/ WHILE */
3   int x; int y;
4   x = read();
5   y = read();
6   while ( x != y ) {
7       if (x <= y) {
8           y = y - x;
9       } else {
10          x = x - y;
```

```
11              }
12          }
13          // Output result
14          write("GCD: ");
15          write(x);
```

your implementation should generate a list of symbols like this:

```
[(INT, int), (ID, x), (SEMICOLON, ;), (INT, int), (ID, y), (SEMICOLON, ;), (ID, x),
 (ASSIGN, =), (READ, read), (LPAR, (), (RPAR, )), (SEMICOLON, ;), (ID, y), (ASSIGN, =),
 (READ, read), (LPAR, (), (RPAR, )), (SEMICOLON, ;), (WHILE, while), (LPAR, (),
 (ID, x), (NEQ, !=), (ID, y), (RPAR, )), (LBRACE, {), (IF, if), (LPAR, (), (ID, x),
 (LEQ, <=), (ID, y), (RPAR, )), (LBRACE, {), (ID, y), (ASSIGN, =), (ID, y), (MINUS, -),
 (ID, x), (SEMICOLON, ;), (RBRACE, }), (ELSE, else), (LBRACE, {), (ID, x), (ASSIGN, =),
 (ID, x), (MINUS, -), (ID, y), (SEMICOLON, ;), (RBRACE, }), (RBRACE, }),
 (WRITE, write), (LPAR, (), (STRING, "GCD: "), (RPAR, )), (SEMICOLON, ;),
 (WRITE, write), (LPAR, (), (ID, x), (RPAR, )), (SEMICOLON, ;)]
```