Seminar 3

You are expected to present solutions to these problems at the seminar in week 3.

1 Consider the grammar

$$\begin{array}{llll} S \rightarrow T\ \$ & T \rightarrow ID & T \rightarrow \text{'('}\ L\ \text{')'} \\ L \rightarrow T\ R & R \rightarrow \text{','}\ L & R \rightarrow \epsilon \end{array}$$

Construct a NULLABLE/FIRST/FOLLOW table and an LL(1) table for this grammar.

2 In the following small grammar for arithmetic and logical expressions INT is a string of digits and BOOL is either true or false. How much look ahead may be required to do predictive syntax analysis?

$$\begin{array}{llll} E \rightarrow N & N \rightarrow INT & & B \rightarrow BOOL \\ E \rightarrow B & N \rightarrow \text{'('}\ N\ \text{'+'}\ N\ \text{')'} & & B \rightarrow \text{'('}\ B\ \text{'\&'}\ B\text{')'} \end{array}$$

3 A variable declaration in a small programming language has the grammar

$$\begin{array}{lll} \text{declaration} & \rightarrow & \text{type ID ';'} \\ \text{type} & \rightarrow & \text{'int'} \mid \text{'boolean'} \end{array}$$

a. Construct one Java class to represent a declaration and one to represent a type. Implement constructors to initialize the attributes of the classes using parameters. No methods are required.

b. Instead, introduce one class for each type.

4 The following concrete grammar

$$\begin{array}{lll} \text{stmt} & \rightarrow & \text{whileStmt} \mid \text{assignment} \mid \text{block} \\ \text{whileStmt} & \rightarrow & \text{'while' '(' expr ')' block} \\ \text{assignment} & \rightarrow & \text{ID '=' expr} \\ \text{block} & \rightarrow & \text{'\{' stmtList '\}'} \\ \text{stmtList} & \rightarrow & \text{stmt (';' stmt)*} \end{array}$$

is part of a grammar for a programming language. Construct four Java classes to represent the three kinds of statements as subclasses to an abstract superclass. Show the constructors. No methods are required. You may assume that there is a class `Expr` to represent expressions. You may use the Collection class `java.util.ArrayList` to represent a list of statements.

5 Implement `public String toString()` for the three non-abstract classes in problem 4 so that they return `String` representations of the statements according to the concrete grammar. You may assume that there is an analogous `toString` method in `Expr`.

6 In a recursive descent parser for the grammar of problem 4 there is a method

```
public static void whileStmt() {
  accept(WHILE);
  accept(LEFTPAR);
  expr();
  accept(RIGHTPAR);
  block();
}
```

Assume that `expr` returns an abstract representation of the expression with type `Expr` and that `block` returns a representation of the block with type `Block`. Modify `whileStmt` so that it returns a representation of the statement.

7 The grammar rule

expr → term ('-' term)*

has the following parse method

```
void expr() {
  term();
  while (token==MINUS) {
    accept(MINUS);
    term();
  }
}
```

Change the method so that it returns an AST of type `Expr` when "-" is assumed to associate to the left. The `term` method is assumed to return an AST of type `Expr` representing the term. `Sub` is a subtype of `Expr` with constructor `Sub(Expr expr1, Expr expr2)`.

8 The grammar rules

primary → factor
primary → factor '^' primary

describe simple expressions with exponentiation. It can be parsed by

```
void primary() {
  factor();
  if(token==POWER) {
   accept(POWER);
   primary();
  }
}
```

By convention, the exponentiation operator associates to the right so that `a^b^c` should be evaluated as `a^(b^c)`.

Change the method so that it returns an AST of type `Expr`. The `factor` method is assumed to return an AST of type `Expr`. `Power` is a subtype of `Expr` with constructor `Power(Expr expr1, Expr expr2)`.