Software-Modellierung und Verifikation
Informatik 2
Prof. J.-P. Katoen
RWTH Aachen

Priv.-Doz. T. Noll noll@cs.rwth-aachen.de
D. Klink klink@cs.rwth-aachen.de

# 7. Exercise sheet *Compiler Construction 2008*

Due to Wed., 25 June 2008, *before* the exercise course begins.

**Exercise 7.1:**

Consider the following grammar:

$$
\begin{array}{lcll}
S' & \rightarrow & S & w.1 = 1 - b.1 \\
S & \rightarrow & AB & w.1 = w.0, \\
 & & & w.2 = b.1, \\
 & & & b.0 = b.2 \\
A & \rightarrow & AA & w.1 = w.0, \\
 & & & b.0 = b.1 \\
A & \rightarrow & a & b.0 = w.0 \\
B & \rightarrow & BB & w.2 = w.0, \\
 & & & b.0 = b.2 \\
B & \rightarrow & b & b.0 = w.0
\end{array}
$$

a) Prove that the grammar is circular by graphically representing the dependencies in an appropriate derivation tree.

b) Construct the equation system for the derivation tree chosen in a) and solve it.

**Exercise 7.2:**

Consider the following grammar:

$$
\begin{array}{lcll@{\qquad}lcll}
S & \rightarrow & A & i_1.1 = 1 & A & \rightarrow & Aa & i_1.1 = s_1.1 \\
 & & & i_2.1 = 2 & & & & i_2.1 = i_1.0 \\
 & & & s_1.0 = s_2.1 & & & & s_1.0 = s_2.1 \\
 & & & & & & & s_2.0 = 0 \\
A & \rightarrow & a & s_1.0 = 0 & A & \rightarrow & b & s_1.0 = i_2.0 \\
 & & & s_2.0 = i_1.0 & & & & s_2.0 = 0
\end{array}
$$

a) Show that the grammar is not circular using the method from the lecture.

b) Check whether it is strongly non-circular.

**Exercise 7.3:**

Consider the following attributed version of the parameterised grammar from Exercise 4.2 where all attributes are of type boolean.

$$
\begin{array}{lcl rcl}
S & \rightarrow & 1\texttt{,}S_1 \mid \ldots \mid n\texttt{,}S_n & \text{attrSyn.0} & = & \text{attrSyn.3} \\
S_i & \rightarrow & R_i \overset{i\times}{\ldots} R_i & \text{odd.1} & = & \text{true} \\
 & & & \text{odd.}k & = & \neg\text{oddSyn.}(k-1) \qquad \text{for } k > 1 \\
 & & & \text{attrSyn.0} & = & \text{attrSyn.1 XOR } \ldots \text{ XOR attrSyn.}i \\
R_i & \rightarrow & N \overset{i\times}{\ldots} N\texttt{,} & \text{attrSyn.0} & = & \text{attrSyn.1 XOR } \ldots \text{ XOR attrSyn.}i \text{ XOR odd.0} \\
 & & & \text{oddSyn.0} & = & \text{odd.0} \\
N & \rightarrow & \texttt{a} & \text{attrSyn.0} & = & \text{true} \\
N & \rightarrow & \texttt{b} & \text{attrSyn.0} & = & \text{false}
\end{array}
$$

(for all $i > 0$)

a) Extend the recursive descent parser from Exercise 4 such that the attibutes of the grammar are evaluated on-the-fly and the value of attrSyn.0 in the start rule is returned by $s()$.

b) Give an acyclic attribution for the grammar that cannot be dealt with using a recursive descent parser. Explain, e.g. by graphically representing the information flow in an appropriate derivation tree!

**Exercise 7.4: (optional)**

Consider the following ANTLR grammar for terms of binary `OP`erations `+`, `*` and `^` over variable `ID`entifiers and `NUM`bers in prefix notation. E.g. `* + x 2 1` is a word recognised by the grammar:

```
grammar AstEx;

options{output=AST;}
tokens{SID;}

@header{package ast;}
@lexer::header{package ast;}

spec:   NUM
    |   x=ID -> SID[$x.text]
    |   op
    ;

op  :   OP spec spec -> ^(OP spec spec);

ID  :   ('a'..'z')('a'..'z' | 'A'..'Z' | '0'..'9')*;
NUM :   ('0'..'9')+;
OP  :   '+' | '*' | '^';
WS  :   (' ' | '\r' | '\t' | '\u000C' | '\n' | '(' | ')') {$channel=HIDDEN;};
```

Note that the `output` is set to `AST` (abstract syntax tree) and a token `SID` is specified that is not implicitly given by the rules of the grammar. In rules `spec` and `op`, rewrite rules have been used. In `spec` instead of an `ID` token, an `SID` token (with the same text information) will be created. In

`op`, rewriting has been used to specify that the structure added to the AST is not a set containing `OP` and two structures given by `spec`, but a tree with root `OP` and two `spec` children.
Use the debugger of ANTLRWorks to examine the `AST` created for `* + x 2 1` and play with the following Java code to see how the parser output, the abstract syntax tree, can be accessed.

```
package ast;

import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

public class AstTest {

public static void main(String args[]) throws Exception {
        AstLexer lex = new AstLexer(new ANTLRFileStream("/pathTo/astTest.txt"));
        CommonTokenStream tokens = new CommonTokenStream(lex);

        AstParser g = new AstParser(tokens);

        try {

            Tree ast = (Tree) g.spec().getTree();
            System.out.println(ast.toStringTree());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

a) Extend the grammar such that it recognises functions. I.e. `f x y := * + x 2 y`, the term describing function $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ with $f(x, y) = (x + 2) * y$, has to be recognised and an appropriate abstract syntax tree has to be generated.

b) Extend the Java code such that an exception is thrown, if

   – a variable identifier is the same as the function identifier,
   – a variable identifier is declared twice,
   – a variable used on the right hand side has not been declared.

Send your code (the ANTLR and Java files) to `klink@cs.rwth-aachen.de`. Use `Exercise 7.3` as subject and add your student ID numbers (Mat.nr.).