Seminar 4

You are expected to present solutions to these problems at the seminar in week 5.

1 This is grammar for a very small language.

$$\begin{aligned} S &\rightarrow E\ \$ \\ E &\rightarrow (\ E\ ) \\ E &\rightarrow \epsilon \end{aligned}$$

a. Construct an NFA with the states labeled by LR(0) items for the grammar

b. Construct an equivalent DFA. Is the grammar LR(0)?

2 The following abstract grammar describes simple integer and string expressions. The addition operator is overloaded, i.e. it is defined for both integer (addition) and string (concatenation) operands. Adding an integer and a string causes a type error.

```
abstract Expr;
AddExpr: Expr ::= Left:Expr Right:Expr;
IntConst: Expr ::= <INT>;
StringConst: Expr ::= <STRING>;
IdUse: Expr ::= <ID>;
```

Types are represented by static objects of the type `SemType` from

```
interface SemType {
  public final static SemType INT = ...;
  public final static SemType STRING = ...;
  public final static SemType UNKNOWN = ...;
}
```

The type `UNKNOWN` is used to represent the type of an undeclared identifier or an expression that cannot be typed. Introduce the attribute `SemType type` for the `Expr` class and a method `void computeTypes()` which computes the type attributes for the expression. You may assume that the attribute `IdUse.type` is available and no implementation of `IdUse.computeTypes()` is required. The solution should be given as a jadd specification.

3 Extend the type checking of the previous problem with a method `typeCorrect()` in the class `Expr` which decides if the current `Expr` node is type correct. The method should be defined so that errors don't propagate to enclosing expressions. The solution should be given as a jadd specification.

4 With the following abstract grammar fragment variables and constants may be declared and assigned values.

```
Start :: Block:
Block ::= Decls: Decl* Stmts: Stmt*;
abstract Decl ::= IdDecl;
VarDecl: Decl;
ConstDecl: Decl ::= Exp;
IdDecl ::= <ID>;
abstract Stmt;
Assignment: Stmt ::= IdUse Exp;
abstract Exp;
IdUse: Exp ::= <ID>;
IntExp: Exp ::= <INT>;
```

The semantics for the language specifies that variables and constants must be declared in order to be used in assignment statements. Constants receive their values in the declaration and may not appear in the left part of assignment statements. Implement the checking of these rules. You may assume that the `IdUse` class has an attribute `Decl decl` referring to the corresponding declaration and which is `null` if the identifier has not been declared. The solution should be given as a jadd specification.

5 Compute the `decl` attribute in the previous problem by introducing a method `void computeDecl()` in the `IdUse` class. You may assume that there is a superclass, `ASTNode`, to all the grammar classes and that the method `Block enclosingBlock()` has been implemented returning the enclosing `Block` node. The solution should be given as a jadd specification.

6 The following abstract grammar for declarations is given.

```
Decls ::= IdDecl*;
IdDecl ::= <ID>;
```

To declare the same identifier several times in one declaration list is an error. To detect this an attribute boolean multiple should be added to the `IdDecl` class telling if the identifier has been declared several times. Implement a method `void computeMultiples()` in the `Decls` class that computes the multiple attribute of the children. You may introduce auxiliary methods making the solution clearer. The solution should be given as a jadd specification.

7 In a Java method local variables may only be used after the point of declaration. Implement a name analysis checking this. You should use some Java library class to represent the symbol table. The following grammar is assumed.

```
Block ::= DeclOrStmt*;
abstract DeclOrStmt;
abstract Decl: DeclOrStmt;
abstract Stmt: DeclOrStmt;
VarDecl: Decl ::= IdDecl;
IdDecl ::= <ID>;
Assignment: Stmt ::= IdUse Exp;
abstract Exp;
IdUse: Exp ::= <ID>;
```

The implementation should use the *Visitor* pattern. You may assume that there is a class `TraversingVisitor` with a visit method for each grammar class.

```
class TraversingVisitor implements Visitor {
  Object visit(Block node, Object data) {
  ...
  }
  ...
}
```

The implementations of the visit methods in `TraversingVisitor` traverse the abstract syntax tree in preorder. You may assume that every `AST` class has an `accept` method.

The symbol table should be an instance variable in your visitor. Implement a method `boolean namesCorrect(Block)` in your visitor that makes the name analysis for all the program and returns true if all identifiers are used correctly.

8 Use JastAdd reference attributes to define an attribute `Decl IdUse.decl()` for the grammar of problem 4.