Third Laboratory Assignment

1. **Overview**

The goal of this lab is to understand how a semantic analyser for the programming language PL/3007 works. Different from the previous lab, this lab provides you with a complete set of the .ast file and the .jrag files (downloadable as a zip file from the course website) which the JastAdd tool uses to generate the semantic analyser. You are also provided with a unit test suite with 60 test cases, as an example of doing unit test of the semantic analyser.

There is no submission to be made for this lab. You should read this document and the given code in `grammar.ast` and the `*.jrag` files. Complete a simple short individual quiz in NTULearn (Lecture site) which has 2% weightage on the final grade. The deadline for completing the individual quiz is **11.59pm 19th March.**

2. Code Files Provided

Download the file lab3.zip from the course webpage. The folder **src** contains the following sub-folders:

- folder **frontend** contains an abstract grammar file defining the abstract syntax of PL/3007 (`grammar.ast`), and several `.jrag` files containing attribute definitions and inter-type field and method declarations for the semantic analyser, as detailed below;
- folder **test** contains the unit test suite in file `SemanticAnalyserTests.java`;

These files are

- `grammar.ast` – defines all classes in abstract grammar for building the AST and doing the semantic checking
- `namecheck.jrag` – all the attributes/methods for the various AST classes/nodes to do scope checking
- `names.jrag` – all the attributes for name analysis (search for the declarations of various names) to help scope checking
- `typecheck.jrag` - all the attributes/methods for the various AST classes/nodes to do type checking
- `types.jrag` – all the attributes for type analysis (compute/infer the types of the AST nodes) to help type checking
- `SemanticAnalyserTests.java` – the unit test suite which contains 60 test cases to test the semantic analyser.

Note that since you are provided with the source code for the semantic analyser and the test module only, you are not able to run the semantic analyser.

3. A Walkthrough of the Source Code of the Semantic Analysis

We describe how the semantic analyser of PL/3007 works by going through the source code provided. Not everything is described. To have a more thorough understanding, you should read the relevant codes as well.

3.1 grammar.ast

This file defines the abstract syntax of PL/3007. In grammar.ast, we see:

```
Program ::= Module*;
```

Each source program to be compiled is an instance of the class `Program`. Class `Program` has a field which is a list of the instances of Class `Module`. The general structure of an AST is illustrated in Figure 1. Of course each source program will have its own AST representing its structure.

With respect to object-oriented programming design, many good systems have class hierarchies which include abstract superclasses. In some cases, abstract superclasses form a few levels of the hierarchy. We see this approach adopted whenever an inheritance situation arises in the design of the AST structure.

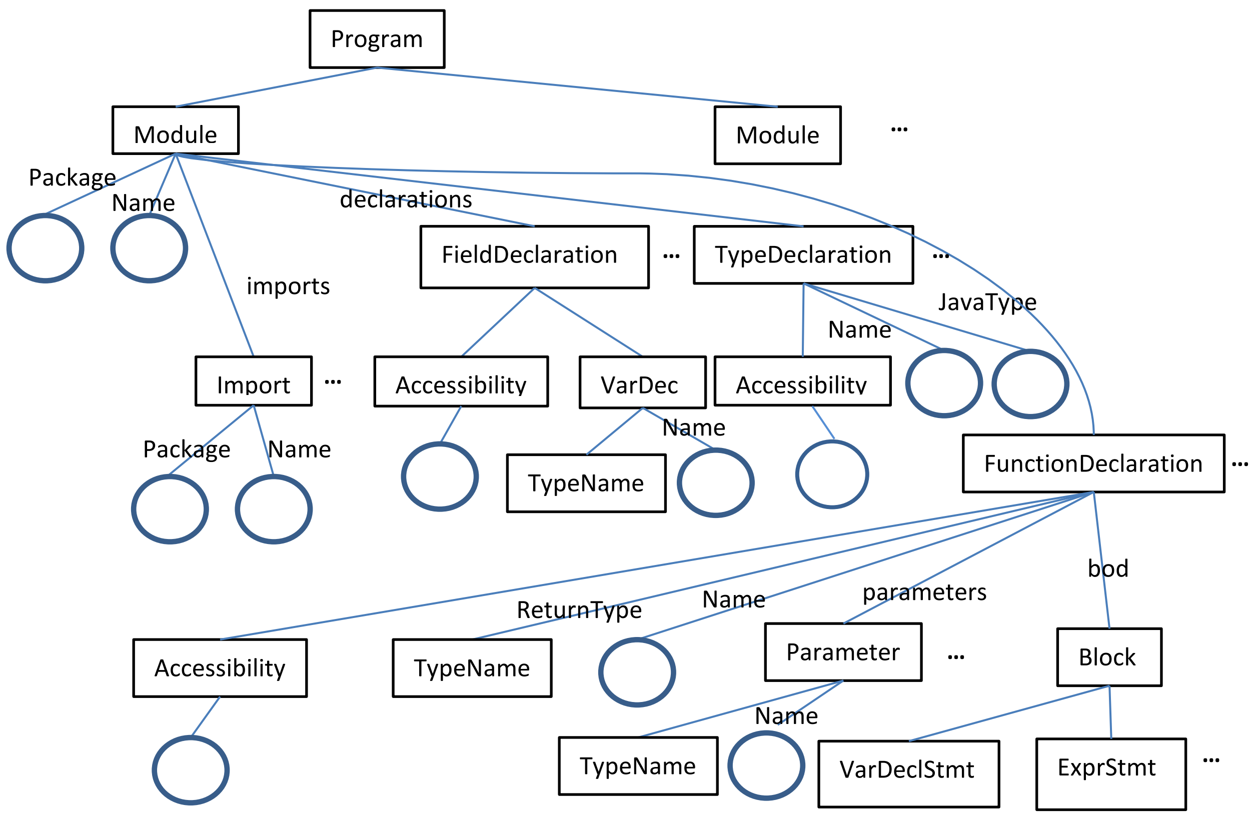

Figure 1. **A general structure of the ASTs for programs in PL/3007**.

Most of the definitions in the abstract grammar should be fairly straightforward: they correspond to the concrete grammar, except that alternative production rules are replaced by object-oriented inheritance, and inessential terminals are omitted. One difference between the abstract grammar and concrete grammar in our lab is that the abstract grammar defines the AST nodes for a complete program where the concrete grammar only defines the syntax for one module (which is what you do in Lab 2).

For example, in concrete grammar, i.e. context-free-grammar (CFG), we have

```
Stmt = Expr SEMICOLON
      | TypeName ID SEMICOLON
      | LCURLY Stmts RCURLY
      | IF LPAREN Expr RPAREN Stmt ELSE Stmt
      | IF LPAREN Expr RPAREN Stmt
      | WHILE LPAREN Expr RPAREN Stmt
      | RETURN Expr SEMICOLON
      | RETURN SEMICOLON
      | BREAK SEMICOLON
      ;
```

In abstract grammar, we have

```
abstract Stmt;

ExprStmt : Stmt ::= Expr;
VarDeclStmt : Stmt ::= VarDecl;
Block : Stmt ::= Stmt*;
IfStmt : Stmt ::= Expr Then:Stmt [Else:Stmt];
WhileStmt : Stmt ::= Expr Body:Stmt;
ReturnStmt : Stmt ::= [Expr];
BreakStmt : Stmt;
```

The individual abstract grammar rule for the various types of statements corresponds to the respective production rule in CFG, with terminal symbols (tokens) omitted. The alternative production rules for Stmt are replaced by class inheritance: various types of statement classes are subclasses of the abstract super class Stmt.

Having alternative production rules for a nonterminal is one possible scenario for a class inheritance situation. There are other inheritance situations. Notice that in our abstract grammar BinaryExpr is the superclass of CompExpr (all comparison expressions), which is the superclass of ArithCompExpr (arithmetic comparison expressions <, >, <= and <=).

```
abstract CompExpr : BinaryExpr;
EqExpr : CompExpr;
NeqExpr : CompExpr;
abstract ArithCompExpr : CompExpr;
LtExpr : ArithCompExpr;
GtExpr : ArithCompExpr;
LeqExpr : ArithCompExpr;
GeqExpr : ArithCompExpr;
```

Why do we design such a class hierarchy? Read on. You will find the answer in Sections 3.4 and 3.5.

There are also node types/classes for type descriptors in the abstract syntax, which do not have concrete syntax. These nodes are not created by the parser when it builds the AST during parsing. They are created during semantic checking when the types of the various relevant AST nodes need to be computed (see types. jrag). Type descriptors are different from type names: while an AST of a source program may have several different nodes of type IntTypeName, their getDescriptor() attributes all refer to the same (and the single instance of) type descriptor IntType. Similarly, there is only a single BooleanType node representing the type boolean, and a VoidType node representing void. ArrayType represents an array type, with its only child ElementType representing the element type of the array. Finally, JavaType wraps Java types for use in PL/3007 programs.

3.2 names.jrag

This file defines attributes for name analysis (name lookup), similar to those discussed in the lecture. Attributes are translated to methods and added automatically into the respective class files by JastAdd (refer to Lecture slide 89). In particular, there are inherited attributes lookupVar, lookupFunction, and lookupType for searching the declaration nodes of variable names (including field names), function names and type names, respectively.

Variable lookup in PL/3007 is very similar to variable lookup in C or in Java. When looking up a variable from inside a function, the lookup attributes first traverse the enclosing blocks, looking for local variable declaration statements that declare a variable of the name we are looking up. In the AST, such a variable declaration statement is represented as a node of type VarDeclStmt, which contains a VarDecl child node. As discussed in the lecture, once a variable has been found, we do not continue exploring surrounding blocks.

If no local variable with the appropriate name is found, name lookup will eventually reach the function node itself. At this point, we check whether there is any parameter with the name we are looking for.

If the name to be looked up is neither a local variable nor a parameter, we finally try to look it up as a field. In PL/3007, a field can either be declared locally in a module, or imported from another module. The helper attribute lookupLocalField performs local lookup within a module, stepping through all the declarations trying to find a FieldDeclaration node with the right name. If such a field does not exist, it should walk over all import statements. The method Import. resolve() is to find out which module an import refers to, and then use lookupLocalField on the imported module to try and find the field inside that module. Note, however, that a field can only be imported if its accessibility is public (Declaration. isPublic() is used to find out). Finally, if the field cannot be found among the imported modules, null is returned. The details at the code level are discussed in tutorial Question 3 of T3_part2.

Lookup for types (lookupType) and functions (lookupFunction) works exactly the same as field lookup.

3.3 namecheck.jrag

This file defines the inter-type methods namecheck() for the various AST nodes/classes to check for scoping errors (refer to Lecture slide 3). These methods are added automatically into the respective class files by JastAdd (refer to Lecture slide 89).

The scope check of a source program, i.e. the AST of the source program, starts with Program. namecheck() shown below. Scope check traverses the AST top-down, performing checks along the way. The functionality in names. jrag is used to perform name lookups.

```
public void Program.namecheck() {
   // check for name clashes on modules
   Set<String> module_names = new HashSet<String>();
   for(Module module : getModules()) {
      if(!module_names.add(module.getQualifiedName()))
          error("Multiple modules with name " + module.getQualifiedName());
      module.namecheck();
   }
}
```

In each iteration of the for loop over the list of Modules: check whether the module name has appeared in the program before (issue error message if it has); then call namecheck() of the Module class to perform scope check for the module.

```
public void Module.namecheck() {
      Set<String> imports = new HashSet<String>();
      for(Import imp : getImports()) {
            imp.namecheck();
            if(!imports.add(imp.getQualifiedName()))
            error("Multiple imports of module " + imp.getQualifiedName());
      }
      for(Declaration decl : getDeclarations())
            decl.namecheck();
}
```

The 1st for loop iterates over the list of import nodes in the module. Each iteration calls namecheck() method of the Import class; then check whether this is a duplicated import statement (if the module name appeared in an earlier import node).

The 2nd for loop iterates over the list of Declarations. Each iteration calls the namecheck() method of the Declaration class. Note that Declaration is an abstract class, so the namecheck() of FunctionDeclaration, FieldDeclaration or TypeDeclaration is called, depending on what subclass this member of the Declaration list is in each iteration.

So scope check starts from the root (the Program node) of the AST performed by the namecheck() method of the Program class. The namecheck() method of each internal AST node does two things: (1) calls the namecheck() method(s) of its children node(s) – scope check continues down the AST towards the leaf nodes. (2) if necessary, check scope error in the current node, e.g. Module node needs to check whether an import module's name has appeared already, i.e. a module is imported more than once. Two more examples of the namecheck() method of an internal node of AST are discussed below (Assignment node and VarDecl node).

```
public void Assignment.namecheck() {
      getLHS().namecheck();
      getRHS().namecheck();
}
```

Do name check for the left subtree and the right subtree. For example, if the left subtree is a VarName, VarName.namecheck() will be called. If the left subtree is an ArrayIndex, ArrayIndex.namecheck() will be called. Similarly, if the right subtree is a VarName, VarName.namecheck() will be called. If the right subtree is a Call, Call.namecheck() will be called. If the right subtree is a MulExpr, MulExpr.namecheck() will be called (actually, MulExpr class inherits the namecheck() method from superclass BinaryExpr so BinaryExpr.namecheck() is called).

```
public void VarDecl.namecheck() {
     getTypeName().namecheck();

     // check that there aren't two variables with the same name in the
same scope
     VarDecl decl = lookupVar(getName());
     if(decl != null && decl != this && decl.getScope() == this.getScope())
          error("Multiple declarations for " + getName() + " in same
scope");
}
```

This method is called from FieldDeclaration. namecheck() and VarDeclStmt. namcheck(). A VarDecl node has one subtree TypeName and one field Name of string type (see abstract grammar for VarDecl in grammar. ast). The method therefore

1. Do name check for the subtree node TypeName.
2. Search for the declaration of Name in this VarDecl node by calling lookupVar() method.
3. If the name in the VarDecl node has been declared in the same scope, issue an error message of multiple declarations.

When scope check reaches a leaf node of the AST: e.g. VarName node

```
public void VarName.namecheck() {
      if(decl() == null)
             error("Variable " + getName() + " cannot be resolved.");
}
```

Call decl() to search for the declaration of the variable name. If it cannot be found, it means this name has not been declared so an error message for the undeclared variable name is issued.

3.4 types.jrag

The file defines several attributes related to type checking. First, it defines singleton instances of IntType, BooleanType and VoidType that are shared throughout the entire program. There is also an attribute arrayType for creating an ArrayType corresponding to some existing type descriptor. Note that this attribute is lazy, so for any given type descriptor the ArrayType descriptor will only be created once, and then cached. To obtain the type descriptor for integer arrays, for instance, TypeDescriptor. INT. arrayType() is invoked. We also maintain a cache of type descriptors for Java types in field javaTypeDescriptors. These type descriptor nodes will be attached to the AST nodes to describe their types.

We have an attribute getDescriptor for the super class TypeName. Then a set of equations follows, each of which computes the type descriptor for a subclass of TypeName. This attribute is also lazy so the cached field in each (subclass of) TypeName references the type descriptor node of this type, i.e. a descriptor node is attached to the AST node to indicate its type. Thus the type descriptor node IntType is attached to an IntTypeName node in an AST. The type descriptor node BooleanType is attached to an BooleanTypeName node in an AST and so on.

At the very end of the file, there is an attribute type for super class Expr. A set of equations follows, each of which computes the type for a subclass of Expr. Note that for our simple language PL/3007,

```
eq BinaryExpr.type() = TypeDescriptor.INT;
```

we simply assume that a binary expression is of type IntType. This defines the type of all subclasses of BinaryExpr unless a subclass defines its own type() method. For comparison expressions (==, !=, <, <=, >, >=)

```
eq CompExpr.type() = TypeDescriptor.BOOLEAN;
```

This defines the type of all subclasses of CompExpr and over-writes the type() method of its super class BinaryExpr.

As a result, all comparison expression nodes in the AST have a reference to the singleton instance of the type descriptor node BooleanType. Other AST nodes of the remaining subclasses of BinaryExpr have a reference to the singleton instance of the type descriptor node IntType.

3.5 typecheck.jrag

The file defines the inter-type methods typecheck analogous to namecheck() that performs type checking. They are added automatically into the respective class files by JastAdd (refer to Lecture slide 89). The functionality in types. jrag is used to perform type analysis. Furthermore, for simplicity, we assume that an array literal is a one dimensional array.

The type check of a source program, i.e. the AST of the source program, starts with Program. typecheck().

```
public void Program.typecheck() {
      for(Module module : getModules())
            module.typecheck();
}
```

In each iteration of the for loop over the list of Modules: call typecheck() of the Module class to perform type check of the module.

```
public void Module.typecheck() {
      for(Declaration decl : getDeclarations())
            decl.typecheck();
}
```

In each iteration of the for loop over the list of Declarations: call typecheck() of the Declaration class. Note that Declaration is an abstract class, so the typecheck() of FunctionDeclaration, FieldDeclaration or TypeDeclaration is called, depending on what subclass this member of the Declaration list is in each iteration.

Similar to scope check, type check starts from the root (the Program node) of the AST - performed by the typecheck() method of the Program class. The typecheck() method of each internal AST node does two things: (1) calls the typecheck() method(s) of its children node(s) – type check continues down the AST towards the leaf nodes. (2) if necessary, check type error in the current node. Two examples of the typecheck() method of an internal node of AST are discussed below (Assignment and IfStmt).

```
public void Assignment.typecheck() {
      getLHS().typecheck();
      getRHS().typecheck();
      if(getLHS().type() != getRHS().type())
            error("The two sides of the assignment have different types.");
}
```

Do type check for the left subtree and the right subtree. This is to check for type inconsistencies in the subtrees.

If the left subtree is a VarName, VarName. typecheck() will be called. If the left subtree is an ArrayIndex, ArrayIndex. typecheck() will be called.

If the right subtree is a VarName, VarName. typecheck() will be called. If the right subtree is a Call, Call. typecheck() will be called. If the right subtree is a MulExpr, MulExpr. typecheck() will be called (actually, MulExpr class inherits the typecheck() method from superclass BinaryExpr).

For Assignment node, it is necessary to check the type error in Assignment node after checking the children nodes: left subtree and the right subtree should have the same type. Otherwise it is an error. Note that in PL/3007 there is no subtyping. So, the types of both sides of an assignment (or the types of an argument and its corresponding parameter) have to be exactly the same. This is unlike in Java or C, where implicit conversions are sometimes performed.

```
public void IfStmt.typecheck() {
      getExpr().typecheck();
      getThen().typecheck();
      if(hasElse())
            getElse().typecheck();

      // check that if condition is 'boolean'
      if(!getExpr().type().isBoolean())
            error("If condition is not of type boolean.");
}
```

Do type check for the Expr subtree, the Then subtree and the Else subtree if there is Else. This is to check for type inconsistencies in these subtrees of the IfStmt node.

After the above, it is necessary to check if the Expr node is of boolean type. If not, issue an error message. For example, if the Expr is a binary expression 2+3, i.e. it is an AddExpr node, then it is not a boolean typed expression and this is an error.

The typecheck() methods of some superclasses, E.g. Declaration or Stmt, are empty and their respective subclasses provide the code for their typecheck() methods. Sometimes a typecheck() method for a superclass provides the code to define the actions to take. For example, BinaryExpr.

```
public void BinaryExpr.typecheck() {
      getLeft().typecheck();
      getRight().typecheck();
      if(!getLeft().type().isNumeric() || !getRight().type().isNumeric())
            error("Both operands of a binary arithmetic operator must have
numeric type.");
}
```

This defines the typecheck() method for all the subclasses of BinaryExpr unless there is a typecheck() method for a particular subclass that overwrites it (CompExpr has its own typecheck() method). So there is no need to have individual typecheck() method in a number of subclasses of BinaryExpr.

The method will do type check for the left and right subtrees. This is to check for type inconsistencies in these subtrees. After checking the subtrees, it is necessary to ensure both subtree nodes are of numeric type.

```
public void CompExpr.typecheck() {
      getLeft().typecheck();
      getRight().typecheck();
      if(getLeft().type() != getRight().type())
            error("Both operands of a comparison operator must be of the
same type.");
      if(getLeft().type().isVoid())
            error("Operands in comparison may not be of type void.");
}
```

This defines the typecheck() method for all the subclasses of CompExpr and overwrites the typecheck() method of BinaryExpr. This is because for comparison operations, we just need to make sure the left and right operands are of the same type and it is not void type. This method is overwritten by the typecheck() method of ArithCompExpr because this class has more specific requirement than that for CompExpr.

The typecheck() methods of leaf nodes VarName and Literal are empty because there is no type inconsistencies in a variable name or a constant.

3.6 astutil.jrag, errors.jrag and runtime.jrag

astutil. jrag defines several utility attributes for navigating the AST. Certain parts of the semantic analyser use these definitions. Similarly, errors. jrag contains utility methods for reporting semantic errors. Read the definitions and understand them. You do not need to read or understand file runtime. jrag.

3.7 SemanticAnalyserTests.java

This is the class which conducts the tests for the semantic analyser. The main function is the method runtest(). Even though this is the tester of the semantic analyser, it shows the main steps in the analysis phase of the compiler (this compiler does not do code generation and code optimization). The main steps in runtest() are:

```
1. Parser parser = new Parser();
2. List<Module> modules = new List<Module>();
3. for(String src : srcs)
4.    modules.add((Module)parser.parse(new Lexer(new StringReader(src))));
5. Program prog = new Program(modules);
6. prog.namecheck();
7. if(!prog.hasErrors()) prog.typecheck();
```

Line 1: creates the parser class instance.
Line 2: creates an empty list of Modules.
Line 3: iterates line 4 over the input source
Line 4: "((Module)parser.parse(new Lexer(new StringReader(src))))" calls the parse() method of the parser to do syntax analysis of one module, creates the AST for that module and returns the reference to the AST (root of the AST is an instance of the Module class). Then modules.add() adds the Module to the list of modules. Note that a Lexer is created then passed to the parse() method as a parameter.
Line 5: creates a `Program` class node where the list of Modules is the children of this node. The `Program` node is the root of the AST for the whole source program to be compiled.
Line 6: call `namecheck`() method of the `Program` class to do name checking starting from the root of the AST.
Line 7: If the name checking of the program has no error, call `typecheck`() method of the `Program` class to do type checking starting from the root of the AST.

This completes our walkthrough of the source code of the semantic analysis.

4. Unit Tests

Class `SemanticAnalyserTests` contains a utility method `runtest` that makes it very easy to prepare tests: this method accepts as its first argument a Boolean flag, and as its remaining arguments one or more strings. The strings are parsed as module definitions, and their ASTs are assembled into a whole program AST. Then the semantic checks are performed. If the Boolean flag is `true` (the default), there should not be any semantic errors. If the flag is `false`, on the other hand, there *should* be at least one error. If this is not the case, the unit test is considered to have failed.

As an example, here is the invocation of `runtest` in the first unit test in `SemanticAnalyserTests`:

```
runtest(false,
        "module M { }",
        "module M { }");
```

This states that the program consisting of two empty modules both named M should fail semantic analysis.

Sixty tests in the `SemanticAnalyserTests` are provided for testing the semantic analyser. The file just serves as an example of the kind of tests we may conduct to eliminate bugs in our semantic analyser (there are many more).

The goal of unit testing is to test each smallest piece of testable code in a class/method of the semantic analyser individually and independently, and determine whether it behaves exactly as you expect. For example, to test `Import.namecheck`(), the behaviors of the method under three situations are tested: (1) the first if condition is true; (2) the first if condition is false and the second if condition is false; (3) the first if condition is false and the second if condition is true. These are done by the following three tests.

```
    @Test
    public void testUnresolvedImport() {
          runtest(false,
                      "module M { import N; }");
    }
    @Test
    public void testResolvedImport() {
          runtest("module M { import N; }",
                      "module N { }");
    }
```

```
	@Test
	public void testSelfImport() {
		runtest(false, "module M { import M; }");
	}
```

Another example, test the return statement of a function. The following code shows the tests that test (1) the function is of void type and returns nothing; (2) the function is of a non-void type and returns an expression of the correct type; (3) the function is of void type but returns an expression of some other type; (4) the function is of a non-void type but returns nothing; (5) the function is of a non-void type but returns an expression of a type different from the function type.

```
	@Test
	public void testVoidReturn() {
		runtest("module M {" +
		            "  void foo() {" +
		            "    return;" +
		            "  }" +
		            "}");
	}
	@Test
	public void testExprReturn() {
		runtest("module M {" +
		            "  int foo() {" +
		            "    return 0;" +
		            "  }" +
		            "}");
	}
	@Test
	public void testVoidTypeReturnWrong() {
		runtest(false,
		            "module M {" +
		            "  void foo() {" +
		            "    return 0;" +
		            "  }" +
		            "}");
	}
	@Test
	public void testExprReturnVoid() {
		runtest(false,
		            "module M {" +
		            "  boolean foo() {" +
		            "    return;" +
		            "  }" +
		            "}");
	}
	@Test
	public void testReturnTypeWrong() {
		runtest(false,
		            "module M {" +
		            "  int foo() {" +
		            "    return true;" +
		            "  }" +
		            "}");
	}
```

5. Resources

The JastAdd homepage http://jastadd.org contains quite a bit of documentation about JastAdd. However, the materials introduced in the lectures, this document and the set of source code should be enough to understand this lab.

6. Administrativa

If you have any doubts, you are strongly encouraged to ask questions. Remember to complete your short online test by **11.59pm 19th March**.