# Fourth Laboratory Assignment

## 1. Overview

The goal of this lab is to implement a code generator for the programming language PL/3007. In the first two labs, you implemented a lexer and a parser. A semantic analyser is provided in Lab 3. Hence, by the end of this lab you will have created a full compiler from PL/3007 to Java bytecode.

Like the previous labs, this lab provides you with a preconfigured Eclipse workspace (downloadable as a zip file from the course website) to get you started. The workspace contains several supporting files, which you do not need to touch, and several Java classes containing a partial definition of a code generator. The workspace also contains a rudimentary test suite, which you are encouraged to use and extend to test your implementation.

Your assignment will be graded by the instructors using a test suite, so it is imperative that the code you hand in compiles and behaves correctly.

## 2. Setup Instructions

The basic instructions are the same as in the previous lab: Download the file lab4.zip from the course webpage. Unzip it somewhere on your hard drive to reveal a folder lab4. **Start Eclipse and import lab4 as a Java project into your Lab2 workspace.** It contains a single project Lab4, which contains all the code you need to complete the project.

The code contained in this project is structured as follows:

- folder `src` contains the source code:
  - package `backend` contains several Java classes comprising the code generator, as explained below;
  - package `ast` contains a visitor class explained below;
  - package `runtime` contains a single class `Runtime` which is the runtime environment of PL/3007 programs;
  - package `test` contains the test suite in file `CompilerTests.java`;
- folder `lib` contains third-party libraries as well as a lexer (`lexer.jar`), a parser (`parser.jar`), and the AST classes with embedded semantic analyser methods (`ast.jar`).

In this lab, no program generator is used, hence there should not be any build errors. Try running the test suite by right-clicking on Lab4 and choosing "Run As -> JUnit Test". This will bring up the JUnit pane on the left, where the results of the tests are displayed. Currently, there are only a few tests, and they fail: not too surprising, since we have not implemented any code generator functionality yet.

## 3. Problem Description

The code generator is structured into five main classes `ProgramCodeGenerator`, `ModuleCodeGenerator`, `FunctionCodeGenerator`, `StmtCodeGenerator`, `ExprCodeGenerator`, and a utility class `SootTypeUtil`.

### 3.1. Visitors

Many of these classes make heavy use of the Visitor Pattern, so this may be a good time to brush up on this important design pattern. In brief, a visitor is an object that performs an action on an object of some class C, where this action is implemented differently for different subclasses of C. For this lab, you are provided with an implementation of a visitor class for AST nodes in `ast/Visitor.java`: this class defines *visitor methods* for each AST node type, where the visitor method for node type `N` is called `visitN` and takes a node of type `N` as argument. The visitor class is generic with a type parameter `A`, which stipulates the return type of the visitor methods.

In the default implementation, the visitor method for node type `ASTNode` (which is the root AST node type that all other AST node types inherit from) returns `null` (which in Java is an acceptable return value, regardless of which actual type A is instantiated to). All other visitor methods invoke the visitor method for their super node type. The

code generator visitors each override only a part of the visitor methods, and leave the other ones to return `null`.

To apply a visitor object `v` to an AST node `n`, use the method `ASTNode.accept` like this: `n.accept(v)`. At runtime, this will invoke the appropriate method `visitN`, where `n` is the *runtime* type of `n`, and pass `n` itself as the only argument.

### 3.2. Code Generation

The `ProgramCodeGenerator` class is in charge of driving code generation for the entire program. You do not need to change anything in its implementation, but it is important to understand how it works. When compiling PL/3007 to Java bytecode, each module is compiled into a Soot class. The program code generator keeps a mapping `module2class` that maps every module to the corresponding Soot class. For a given module, you can use `ProgramCodeGenerator.getSootClass` to find the appropriate class; if no class has been generated yet, an empty one will be created at this point. The most important method in the program code generator is `generate`, which accepts a `Program` AST node (which is the root node of a program's AST), initialises Soot, and then iterates over all the modules in the program and hands them off to the module code generator to generate the appropriate Soot class for it.

The `ModuleCodeGenerator` also does not need to be changed, but its code again should be understood. The module code generator simply walks over all declarations in the module and generates equivalent Soot declarations for them. In particular, it creates, for every field, a static field declaration with the appropriate type, accessibility, and name. The utility class `SootTypeUtil` is used to map from PL/3007 type descriptors to Soot types: note how a visitor is used to avoid lots of **instanceof** checks on the type descriptor. (Needless to say, you should not change anything in class `SootTypeUtil` either.)

Going back to the module code generator, code generation for functions is handed off to the `FunctionCodeGenerator` class, which generates a static method for each function declaration. Note that no code is generated for type declarations: such declarations only introduce new names for existing types and do not really define anything new.

The function code generator performs two main tasks: it generates a Soot method with all the code for the function in it, and it maintains the mapping from PL/3007 parameters and local variables to Soot local variables. The latter is done by the map `sootLocalMap`, which is managed by method `getSootLocal`. The function code generator also keeps a set of the names of all Soot variables that have been declared so far (`generatedNames`): this is important because in PL/3007 there can be several local variables with the same name in the same function as long as they are declared in distinct blocks. In Soot (as in Java), on the other hand, this is not allowed. For this reason, `getSootLocal` checks whether a local of the same name has already been declared, and if so generates a fresh name (by appending a number to the end of the variable name). Similarly, method `mkTemp` can be used to generate a fresh local variable with a unique name; this will be needed during code generation for expressions.

Finally, method `generate` performs the actual task of generating code for a function declaration. The details of setting up the required Soot data structures are tedious but not particularly complicated; you should study the code carefully, although you do not need to change anything. Code generation for the function body is delegated to class `StmtCodeGenerator`.

The statement code generator is a visitor class, with different methods performing code generation for different kinds of statements. Note that the visitor type parameter is `Void`, representing the fact that the statement code generator returns no values; it simply appends the statements it generates to the statement list in field `units`. (For somewhat silly reasons, the visitor methods still need to return `null`, however.) You are asked to supply the missing implementations of the visitor methods for `BreakStmt` and `WhileStmt`; the `TODO` comments will get you started, and Figure 1 below lists all the API methods you might need.

Finally, the expression code generator `ExprCodeGenerator` is again a visitor class, with different methods performing code generation for different kinds of expressions, each returning a Soot Value. It defines a utility method wrap that checks whether some value can be used as an operand (i.e., it is a literal or a variable); if not, it generates a new assignment statement storing the argument value into a fresh temporary variable, and returns that temporary variable. Hence, whenever you need to use the value returned by a recursive invocation of the expression code generator as an operand to another operator, you should use wrap to ensure that the result is a valid Jimple expression. Several visitor methods have already been defined, study them carefully. The TODO comments show you which ones you need to implement yourself. Again, refer to Figure 1 for a complete list of all Soot API methods you will need.

| API Method Signature | Description |
|---|---|
| `Jimple.newGotoStmt(Unit)` | create new **goto** statement branching to the statement given as argument. Return type of newGotoStmt(): GotoStmt which is a subinterface of Unit. |
| `Jimple.newNopStmt()` | create a new **nop** statement. Return type of newNopStmt(): NopStmt which is a subinterface of Unit. |
| `All the following Jimple.new*Expr()` | The (two) argument(s) needs to be a constant or a local. |
| `Jimple.newEqExpr(Value,Value)` | create a new "==" expression. Return type of newEqExpr(): EqExpr which is a subinterface of Value. |
| `Jimple.newAddExpr(Value,Value)` | create a new "+" expression. Return type of newAddExpr(): AddExpr which is a subinterface of Value. |
| `Jimple.newSubExpr(Value,Value)` | create a new "-" expression. Return type of newSubExpr(): SubExpr which is a subinterface of Value. |
| `Jimple.newMulExpr(Value,Value)` | create a new "*" expression. Return type of newMulExpr(): MulExpr which is a subinterface of Value. |
| `Jimple.newDivExpr(Value,Value)` | create a new "/" expression. Return type of newDivExpr(): DivExpr which is a subinterface of Value. |
| `Jimple.newRemExpr(Value,Value)` | create a new "%" expression. Return type of newRemExpr(): RemExpr which is a subinterface of Value. |
| `Jimple.newNegExpr(Value)` | create a new unary minus (-) expression. Return type of newNegExpr(): NegExpr which is a subinterface of Value. |
| `Jimple.newArrayRef(Value,Value)` | create a new array index expression of the form a[i]. The two arguments need to be locals. Return type of newArrayRef(): ArrayRef which is a subinterface of Value. |
| `IntConstant.v(int)` | create a new integer literal. IntConstant is a class implementing the Value interface. |
| `StringConstant.v(String)` | create a new string literal. StringConstant is a class implementing the Value interface. |

Figure 1: Important Soot API Methods

### 3.3. Testing

As always, it is very important that you write your own tests to test your code generator. Unlike in previous assignments, however, for this assignment you will no longer be writing unit tests (which test one component of the compiler), but system tests (which test the entire compiler).

Class `CompilerTests` contains a utility method `runtest` that makes it very easy to do so. This method accepts six arguments:

1. `modules_src`: an array of strings, each being the source code of one module of the program;
2. `main_module`: the name of the program's main module;
3. `main_function`: the name of the main function (which must be declared in the main module);
4. `parm_types`: an array of Java class objects representing the main function's parameter types[1];
5. `args`: an array of Java objects to provide as arguments to the main function;
6. `expected`: the expected result.

[1] If, for instance, the main function takes two arguments of type **int** and String, then this argument should be **new** Class<?>[] { **int**.class, String.class }.

The method will compile the given modules into JVM classes using your code generator, then load the class corresponding to the main module, and invoke the main function with the provided arguments. If the function runs without throwing an exception and its return type is either **void** or it returns an object that is equal to expected, the test passes. Anything else leads to test failure. There is also an overloaded version of `runtest` where the first argument is a single string, for the case of programs that consist of a single module.

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

```
runtest("module Test {" +
    "     public int f() {" +
    "           return 23+19;" +
    "     }" +
    "}",
    "Test",
    "f",
    new Class<?>[0],
    new Object[0],
    42);
```

This compiles the given module into a JVM class `Test`, loads it, invokes the method `f()` of that class without arguments (note that both parm_types and args are empty arrays), and checks that the result is the number 42.

Another example is listed below in which the method `f()` is invoked with appropriate arguments:

```
runtest("module Test {" +
    "  public int f(int x, int y) {" +
    "    return y;" +
    "  }" +
    "}",
    "Test",
    "f",
    new Class<?>[] { int.class, int.class },
    new Object[] { 23, 42 },
    42);
```

## 4. Resources

The Soot homepage `https://sable.github.io/soot/` contains plentiful documentation. Among others, there is a complete API listing at `https://ssebuild.cased.de/nightly/soot/javadoc/`, and a list of tutorials at `https://github.com/Sable/soot/wiki/Tutorials`.

For your convenience, Figure 1 lists all the Soot API methods you will need to complete this assignment.

## 5. Administrative

Please upload your completed copy of `StmtCodeGenerator.java` and `ExprCodeGenerator.java` to the course site within 7 days after your lab session. Please **only upload these two files, not any of the other files** in your workspace. Also please **do not zip up these two files** and submit a .zip file. Each project team should only hand in **one** copy of each of the two .java files.