# Second Laboratory Assignment

## 1. Overview

The goal of this lab is to implement a parser for the programming language PL/3007. In the previous lab, you implemented a lexer, and in the next two labs you go through semantic analysis and code generation, so by the last lab you will have a full, working compiler from PL/3007 to Java bytecode.

As in the previous lab, 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 a main file containing the beginnings of a parser specification using Beaver. The workspace also contains a rudimentary unit test suite, which you are encouraged to use and extend to test your implementation.

Your assignment will be graded by the instructors using a unit 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 lab2.zip from the course webpage. Unzip it somewhere on your hard drive to reveal a folder lab2. Start Eclipse and choose this folder lab2 as your workspace.[1] It contains a single project Lab2, 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 **frontend** contains a skeleton of a parser specification in file `parser.beaver`; to complete this lab assignment, you will need to complete the code of the parser as described below;
  - package **test** contains the unit test suite in file `ParserTests.java`; to test your grammar rules in `parser.beaver`, you should add more tests here;
- folder **gen** contains generated code, including a lexer for you to use in this assignment (so you do not need to use the lexer you developed in the previous lab);
- folder **lib** contains third-party libraries;
- `build.xml` is an Apache Ant build file that we will use to generate the parser from parser.beaver.

At first, Eclipse will display a build error; this is expected, since we have not yet generated the parser from `parser.beaver`.

To do so, click on the downward black triangle next to the "External Tools" button (a green circle with a white triangle in it and a red toolbox next to it). Choose "Lab2 build.xml" from the menu. This will run Beaver on `parser.beaver` and generate a file `Parser.java` in package parser in folder **gen**. You need to perform this step every time you have changed `parser.beaver`. Now, Eclipse should not report any build errors any more. The build process has been configured to automatically refresh the workspace, so that Eclipse immediately sees and compiles the updated version of `Parser.java`. Sometimes, however, the automated refresh does not seem to work. If this is the case, try right clicking on the Lab2 project in the package explorer, and select "Refresh" from the menu.

Next, we run the unit test suite. To do this, click on the downward black triangle next to the "Run Configurations" button (to the left of the "External Tools" button mentioned above), and choose "ParserTests" from the menu. This will bring up the JUnit pane on the left, where the results of the unit tests are displayed. Currently, there is one test, and it fails: not too surprising, since we have not implemented any real parser functionality yet.

## 3. Problem Description

[1] Do not import it as a project into another workspace. There are several important configuration options already set up for this workspace which you will need for development.

We will now give an informal but precise specification of the grammar of a PL/3007 program. It is your task to complete the Beaver specification in `parser.beaver` to parse programs adhering to this specification. Your parser does not need to build an AST.

A PL/3007 program consists of modules. Your parser should be able to parse a single module, thus its start symbol is Module. A module consists of the keyword **module**, followed by an identifier (which is the module's name), followed by an opening curly brace, followed by imports, followed by declarations, followed by a closing curly brace.

A module's imports (nonterminal Imports) specify what other modules this module will import. They consist of zero or more import statements. An import statement (nonterminal Import) consists of the keyword **import**, followed by an identifier, followed by a semicolon. The identifier is the name of a module to be imported.

A module's declarations (nonterminal Declarations) consist of zero or more declarations. A declaration is either a function declaration, or a field declaration, or a type declaration.

A function declaration consists of an accessibility specifier, a type name, an identifier, an opening parenthesis, a parameter list, a closing parenthesis, an opening curly brace, a possibly empty list of statements, and a closing curly brace.

A field declaration consists of an accessibility specifier, a type name, an identifier, and a semicolon.

A type declaration consists of an accessibility specifier, the keyword **type**, an identifier, a single equals symbol, a string literal, and a semicolon.

An accessibility specifier consists of either the keyword **public**, or of nothing at all.

A type name is either a primitive type, an array type, or an identifier.

A primitive type is one of the keywords **void**, **boolean** and **int**.

An array type is a type name, followed by a left bracket, followed by a right bracket. (Hint: In order to avoid LALR conflicts, you will need to expand the definition of the array type nonterminal as discussed in the tutorial.)

A parameter list is a possibly empty list of parameters separated by commas. A parameter is a type name followed by an identifier.

A statement is one of the following:

- a local variable declaration, which consists of a type name, followed by an identifier, followed by a semicolon;
- a block of statements, which consists of an opening curly brace, followed by a possibly empty list of statements, followed by a closing curly brace;
- an if statement, which consists of an **if** keyword, an opening parenthesis, an expression, a closing parenthesis, a statement, an **else** keyword, and another statement; optionally, the **else** keyword and the last statement may be missing (Note: as discussed, this will lead to a shift-reduce conflict, which is correctly resolved by favouring shift over reduce.);
- a while statement, which consists of a **while** keyword, an opening parenthesis, an expression, a closing parenthesis, and a statement;
- a break statement, which consists of a **break** keyword followed by a semicolon;
- a return statement, which consists of a **return** keyword, followed by an optional expression, followed by a semicolon;
- an expression statement, which consists of an expression followed by a semicolon.

An expression is either an assignment or a right hand side expression.

An assignment is a left hand side expression, followed by a single equals sign, followed by an expression.

A left hand side expression is either an identifier or an array access. An array access is a left hand side expression, followed by an opening bracket, followed by an expression, followed by a closing bracket.

A right hand side expression is an arithmetic expression, optionally followed by a comparison operator and another arithmetic expression. A comparison operator is one of EQEQ, NEQ, LT, LEQ, GT, GEQ.

An arithmetic expression is either an arithmetic expression, followed by an additive operator, followed by a term; or just a term. An additive operator is either PLUS or MINUS.

A term is either a term, followed by a multiplicative operator, followed by a factor; or just a factor. A multiplicative operator is one of TIMES, DIV, MOD.

A factor is either a MINUS followed by a factor, or a primary expression.

A primary expression is one of the following:

- a left hand side expression;
- a function call: an identifier, followed by an opening parenthesis, followed by a possibly empty list of expressions separated by comma, followed by a closing parenthesis;
- an array expression: an opening bracket, followed by a non-empty list of expressions separated by comma, followed by a closing bracket;
- a string literal;

- an integer literal;
- one of the Boolean literals TRUE and FALSE;
- a parenthesised expression: an opening parenthesis, followed by an expression, followed by a closing parenthesis.

**Note that you should not use extended rule syntax offered by Beaver, i.e. you should not use ?, * and + in any rules**. Instead, each rule defines a nonterminal symbol and declares the sequence of other symbols that will produce that nonterminal. Note also that your parser should have at most one shift-reduce conflict.

### 3.1. Unit Testing

As mentioned before, it is very important that you write your own unit tests to test your parser. Class ParserTests contains a utility method runtest that makes it very easy to do so: this method accepts as its first argument a string, and as its second test a Boolean flag that defaults to **true**. When invoked from inside a unit test, runtests invokes the parser on the string; if the flag is **true**, the parser should successfully parse the input, otherwise it should fail to parse the input. If this is not the case, an assertion failure is raised, which is displayed as a failed test by Eclipse.

Here is the invocation of runtest in the unit test in ParserTests provided for you:

```
runtest("module Test { }");
```

The call states that the parser should be able to successfully parse the string "module Test { }". Be sure to write both tests that are expected to parse (like the first one), and tests that are expected not to parse. The test suite we use to mark your submissions will contain both kinds of tests.

## 4. Resources

The Beaver homepage http://beaver.sourceforge.net contains some (though admittedly not very much) documentation. Beaver is a LALR(1) parser generator. You do not need to learn about invoking Beaver from the command line, this is taken care of by the provided Ant build script. You also do not need to understand how Beaver handles parse errors.

All you need to remember about Beaver's syntax is that it uses "=" instead of "→" in its rules, and that every rule needs to be terminated by a semicolon. The parser skeleton contains one example to show this syntax.

## 5. Administrativa

As the previous lab, please upload your completed copy of **parser.beaver** to the course site Assignments/Lab 2 by 11.59pm, 1st October. Please upload only this file, not any of the other files in your workspace. **Also please do not submit a zipped file**.

Each project group should only hand in *one* copy of the file.

Late submissions will be penalised. If you submit before the deadline and later find a mistake in your submission, you can resubmit, but only before the deadline (otherwise, it is a late submission).