IN2009 Tutorial 3

Testing

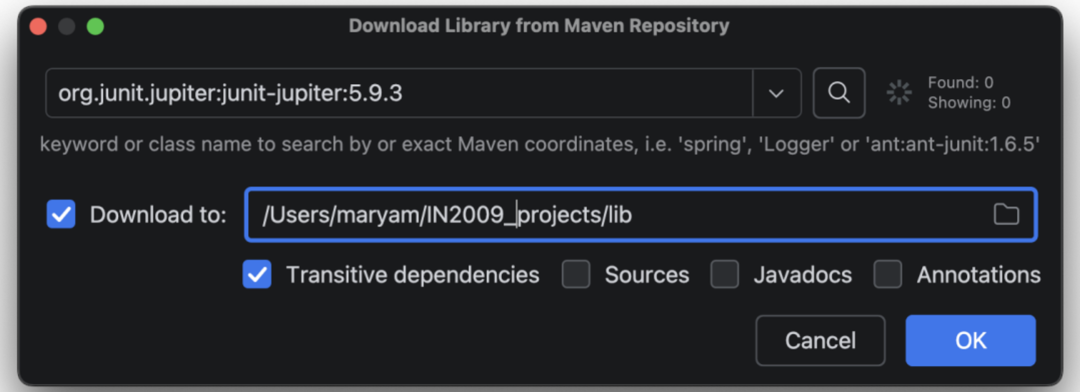
Your coursework project will be assessed by automated testing. Part of the purpose of this tutorial is to introduce you to the Junit test setup, so that you can run your own tests before you actually submit your work for assessment. **You will first need to install the necessary JUnit library support from Maven.**

Installing the Junit Libraries from Maven

This step installs Junit support as a global library (it will then be available for use in all future IntelliJ projects). Open your Calculon project in IntelliJ. Then:

File => Project Structure

1. Under *Platform Settings*, select *Global Libraries*
2. Top of the 2nd column (**not** the third) click **+**
3. Select *From Maven...*
4. Copy-paste the following into the Maven coordinates field:
   `org.junit.jupiter:junit-jupiter:5.9.3`
5. Check the *Download* check-box and *select a location on your local machine*
6. Check the *Transitive dependencies* check-box.
7. Click all the necessary OK buttons.


Download Library from Maven Repository

org.junit.jupiter:junit-jupiter:5.9.3

Found: 0
Showing: 0

keyword or class name to search by or exact Maven coordinates, i.e. 'spring', 'Logger' or 'ant:ant-junit:1.6.5'

[✓] Download to: /Users/maryam/IN2009_projects/lib

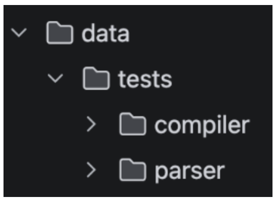
[✓] Transitive dependencies [ ] Sources [ ] Javadocs [ ] Annotations

Adding the Java Test Classes

In the Tutorial 3 Resources folder you will find the sub-folder *src/test* containing three Java classes for running Junit tests. Create a new package called *test* in your Calculon *src* folder (in the same way that you created the *handbuilt* package for Tutorial 2). Then copy the three Java files into your new *test* package.

Adding the Test Input Files

In the Tutorial 3 Resources folder you will find a Zip archive *tests-archive.zip*. The archive contains a folder called *tests* and within that folder two sub-folders, *tests/parser* and *tests/compiler*, which contain test inputs. Unpack the archive and move the entire *tests* folder into your Calculon *data* folder:



Running the Tests

Once you have all the support added, running tests is *very* easy. This is important, because it means you can – and absolutely 100% most definitely **should** – run the tests after every change you make to your code, however small the change. To run the parser tests, right-click on class *test.CalculonParserTest* in the IntelliJ project pane and select the ▷ *Run* option from the pop-up menu. It's the same procedure to run the compiler tests, but use *CalculonCompilerTest* instead. At the outset, few (if any) of the tests will pass. But don't be disheartened: by the end of this tutorial you should be able to get them all to pass.

Extending the Calculon Parser

You can now extend the Calculon parser to handle more language features. You already made the necessary changes and additions to the AST classes in Tutorial 2, so once your parser is working correctly, you will immediately have a compiler: Hey Presto! (If you didn't quite get there in Tutorial 2, you can use my solutions from Moodle – but you *will* have to write your own `StmIf` AST class, since my "solution" for this is a collection of hints, not the actual code.)

In the *Tutorial 3 Resources* folder you will find:

1. An extended grammar specification: file *data/Calculon.sbnf*. Replace your existing version with this file.

2. An extended parser implementation: file *src/parse/CalculonParser.java*. Replace your existing version with this file. Methods `parse()`, `Program()`, `Exp()` and `OperatorClause()` are complete for this version of the language; the rest is up to you. Carry on from where we got to in the lecture.

   Don't forget to use the test support: keep on hitting those test-class Run buttons! *Read* the failure reports: they show you what needs to be fixed. You will generally need to look at the specific test file for a failed test (in *data/tests*) to make proper sense of the failure report.

Debugging

When debugging you may find it useful to run your parser and/or compiler from the command line. Examples:

```
java parse.CalculonParser data/tests/parser/test_20_13.calc
java compile.Compile data/tests/compiler/Ex2.calc
java Run data/tests/compiler/Ex2.ssma
```

Writing your Own Compiler Tests

When you have your compiler working and passing all tests, try writing some new compiler tests of your own. A compiler test is just a source program with some comments at the start which specify the expected output. Have a look at some of the examples in *data/tests/compiler* (most of these should look familiar – they are the examples used in Tutorial 2). For example, consider these two test files:

```
//32
//72
//

begin
 println(5 + (3 * 9));
 println((5 + 3) * 9);
end
```

```
//32
//72

begin
 println(5 + (3 * 9));
 print((5 + 3) * 9);
end
```

The expected output on the left is 32, 72, each on a separate. line The expected output on the right is almost the same but without a newline after the 72.

Note: the one on the right is not actually a valid test for us, because Calculon doesn't include a `print` statement. But feel free to extend the language so that it does!