Programming Assignment 1
Simple Scanning and Parsing

The goal of this assignment is to get an introductory understanding of scanning and parsing, and of scanner and parser generator tools. You will:

- Generate a scanner by defining tokens using regular expressions
- Generate a simple parser by specifying a context-free grammar
- Implement a simple parser by hand, using the recursive-descent method

You will use the following Open Source tools for the assignment (see their web sites for documentation):

- The scanner generator *JFlex*, `https://jflex.de`
- The parser generator *Beaver*, `http://beaver.sourceforge.net/`. Beaver is based on the LALR parsing method. The original Beaver tool is no longer maintained, so we will use an implementation called *NeoBeaver*, `https://bitbucket.org/joqvist/neobeaver`, that is also a bit easier to use, and has better error messages than the original Beaver tool.
- The build tool *Gradle*, `https://gradle.org/`
- The testing framework *JUnit*, `https://junit.org/junit4/`

In the assignment you will practice agile software development with iterative development, automated tests and pair programming, as described in Appendix A.

You should have obtained a git repository for your work with the assignments. For each assignment, there should be several commits.

The language you will implement a scanner and parser for is called *MiniS* (minimal statement language), and to your help, there is a demonstration example *CalcParse*, containing an implementation of a scanner and a parser for another language called *Calc*.

1 Preparations

Try to solve all parts of this assignment before going to the lab session. If you get stuck, use the course forum to ask for help. If this does not help, you will have to ask at the lab session. Make notes about answers to questions in the tasks, and about things you would like to discuss at the lab session.

▶ Major tasks are marked with a black triangle, like this.

Before starting on the assignment, read through the corresponding course material, according to the course web site.

Before starting any programming, read through Appendix A on programming methodology. If you have taken the Software Development in Teams course (EDAF45), you will be familiar with this way of working.

These instructions assume that you are running on a platform with Java 8 or later installed (like the LTH student computers). You can also run on your own computer, but you might then need to install Java (version 8 or later). Furthermore, the instructions assume that you are running from a Unix command line, that you are familiar with basic Unix commands like `cp`, `ls`, `cd`, `man`, etc., and that you edit files using an ordinary text editor. You could also run Eclipse or some other integrated development environment, but your solution should in any case be possible to run from the command line. If you would like to run from Eclipse, see Appendix C for some advice. If you plan to use IntelliJ, see Appendix D.

2 The CalcParse demo

There is a small demonstration example, *CalcParse*, that contains a scanner and a parser for a calculator language called *Calc*. Download the CalcParse example, and make sure that you can run it according to the instructions below, and that you understand the directory structure and the code.

The Calc language contains the following kinds of expressions:

- *Floating point numerals* like 2.0 and 3.14. Every numeral has a decimal point and at least one digit before and after the point.
- *Identifiers* like `PI` and `radius`. An identifier consists of one or more letters from the English alphabet, a–z and A–Z.
- *Products* like `2.0 * radius` and `2.0 * PI * radius`. A product has two operands that are expressions. In the second example, 2.0 is the first operand and `PI * radius` the second.
- *Let* expressions like

  ```
  let PI = 3.1416 in 2.0 * PI end
  ```

  and

  ```
  let PI = 3.1416 in
       let r = 4.0 in
           2.0 * PI * r
       end
  end
  ```

  The strings `let`, `in`, and `end` are reserved words and may not be used as identifiers. Formally, a `let` expression is constructed according to the template

  ```
  let <identifier> = <expression> in <expression> end
  ```

2.1 Run the example from the command line

▶ Download the CalcParse example and run its tests from the command line, as follows.

Download `CalcParse.zip` from `https://cs.lth.se/edan65` into an appropriate directory, and unzip it.

To build the program, we will use Gradle. The Gradle build script, `build.gradle`, contains the tasks `build`, `test`, and `jar`, for building, testing, and creating a jar file for the Calc compiler. The task `clean` removes all generated files.

The CalcParse project includes scripts to launch Gradle on different platforms. On Linux and Mac, the `gradlew` script is used, and on Windows `gradlew.bat` is used.

Open a terminal window and run all the test cases with the following commands (replace `<path>` accordingly):

```
$ cd <path>/CalcParse
$ ./gradlew test
```

This should generate the scanner and parser, compile the resulting Java files, and finally run all the tests. No errors should be reported. How many tests were run?

▶ Generate the compiler as a jar file, and run it from the command line, as follows.

Generate the Jar file `compiler.jar`:

```
$ ./gradlew jar
```

The program `compiler.jar` takes a filename as an argument, parses the file and checks if its contents are valid according to the syntax of the Calc language. Try executing the compiler on a couple of the test case files:

```
$ cat testfiles/example.in
$ java -jar compiler.jar testfiles/example.in

$ cat testfiles/error.in
$ java -jar compiler.jar testfiles/error.in
```

▶ Create a new file, e.g., `testfiles/myexample.in`, with a Calc expression and check if it conforms to the syntax.

2.2 Directory layout

Take a look at the files in the CalcParse directory. The top level contents of the project are:

`build`	class files and other build artifacts (like test reports) generated by Gradle.	
`build.gradle`	the build script	
`gradle`	files for running Gradle	
`gradlew`	script to run Gradle on linux/mac	
`gradlew.bat`	script to run Gradle on windows	
`src`	All source files, both Java files and specification files.	
	`gen`	Java source files generated by JFlex and Beaver.
	`java`	The main Java source files excluding test code. The main program `Compiler` is located in the package `lang`, and is used in `compiler.jar`.
	`test`	Java source files for testing.
	`parser/parser.beaver`	The Beaver parser specification.
	`scanner/scanner.jflex`	The JFlex scanner specification.
`testfiles`	Input programs used in tests.	

Additionally, there are the following files/directories that are relevant if you use Eclipse:

`bin`	class files generated by Eclipse (will only show up if you use Eclipse)
`lib`	Jar libraries needed when running from an Eclipse project. (If you run from the command line or from IntelliJ, libraries automatically downloaded by Gradle are used. These libraries are in the `~/.gradle` directory.)
`.project`	project information for Eclipse
`.classpath`	class path used when running from Eclipse

▶ Add a new test case. Start by letting the test case fail and then fix it so that it passes. *Hint:* take a look at the files in the directories `src/test/lang` and `testfiles`. Add a new test in a similar manner, and run all tests using `./gradlew test`.

2.3 The Calc Language

The following context-free grammar describes the set of all possible sentences in Calc.

```
program  → exp
exp      → factor ("*" factor)*
factor   → let | numeral | id
let      → "let" id "=" exp "in" exp "end"
numeral  → <NUMERAL>
id       → <ID>
```

In the following sections, we describe how the grammar has been translated to JFlex and Beaver specifications. Terminal symbols in the grammar correspond to tokens defined in the JFlex specification. Productions in the grammar are defined in the Beaver specification, in terms of the tokens.

2.3.1 The scanner

The scanner takes a sequence of characters as input and generates a sequence of tokens which are fed to the parser. The scanner in this project is generated from a JFlex scanner specification. The part of the scanner specification that defines tokens is shown in Figure 1.

The specification shown in the figure consists of two parts: macros and lexical rules. Macros are symbolic names for regular expressions. A macro definition has the following form:

macro-identifier "=" *regular-expression*

Macros are used to make the specifications easier to read. The Calc scanner uses three macros: `WhiteSpace`, `ID` and `Numeral`.

Lexical rules are used to specify tokens, consisting of a regular expression (macros can be used) and an action. A rule has the following form:

regular-expression "{" *action* "}"

An action contains Java code and can be used to create an object that represents the token. These objects are passed to the parser. We can see that the rule for white space has an empty action, so all white space will be discarded—no tokens will be generated from them. Thus, white-space characters may be used to separate tokens, but will otherwise be ignored. Note in the rules for white-space, identifiers and numerals how macros are used: by enclosing the macro name with curly braces, for example: `{WhiteSpace}`.

A `Numeral` starts with one or more digits followed by a point and one or more digits.

`ID` represents an identifier consisting of one or more letters. The reserved words thus also match the pattern for an identifier. In such cases, i.e., when there is more than one longest match, the first matching rule is chosen. For example, the string `let` will match the rule `"let" { ... }` rather than the rule `{ID} { ... }`.

The regular expression `<<EOF>>` is a special expression, matching end of file.

The last lexical rule is used as an error fallback. The regular expression `[ab]` matches a or b, and the regular expression `[^ab]` matches any character except a or b. Hence, the rule `[^] { ... }` matches any character that has not been matched by any previous rule.[1] If this rule matches, then it will throw an exception that contains the matched character, which is not valid in the language. The exception can be caught by the main program in the compiler and reported to the user.

The method `sym` creates an object of the class `beaver.Symbol`. This class represents terminal symbols in parsers generated with Beaver. The class stores the token kind as an integer value, the matched string value (e.g., the value of an identifier token), and the line and column number. All token kinds are defined in the class `lang.ast.LangParser.Terminals`, which is generated by Beaver. Each token kind is represented by a unique integer value.

▶ Make sure you understand the scanner specification in Figure 1.

[1] It would also have been possible to use the metacharacter Dot (`.`), which matches any character except for a number of line separators like return and newline.

```
// macro definitions
WhiteSpace = [ ] | \t | \f | \n | \r
ID = [a-zA-Z]+
Numeral = [0-9]+ "." [0-9]+

//  ignore whitespace
{WhiteSpace}  { }

// token definitions
"let"         { return sym(Terminals.LET); }
"in"          { return sym(Terminals.IN); }
"end"         { return sym(Terminals.END); }
"="           { return sym(Terminals.ASSIGN); }
"*"           { return sym(Terminals.MUL); }
{ID}          { return sym(Terminals.ID); }
{Numeral}     { return sym(Terminals.NUMERAL); }
<<EOF>>       { return sym(Terminals.EOF); }

/* error fallback */
[^]           { throw new SyntaxError("Illegal character <"+yytext()+">"); }
```

Figure 1: Part of the JFlex scanner specification for the Calc language

2.3.2 The parser

The interesting part of the parser specification is shown in Figure 2. It consists of two parts: directives (starting with %) and production rules.

```
%terminals LET, IN, END, ASSIGN, MUL, ID, NUMERAL;

%goal program;

program = exp;
exp = factor | exp MUL factor;
factor = let | numeral | id;
let = LET id ASSIGN exp IN exp END;
numeral = NUMERAL;
id = ID;
```

Figure 2: Part of Beaver parser specification for the Calc language

All terminals used in any production must be specified with the `terminal` directive. This directive will create integer constants that represent terminal symbols in the class `lang.ast.LangParser.Terminals`, e.g., `Terminals.LET`. These integer constants are used in the actions in the scanner specification, see Figure 1. The `goal` directive specifies the start symbol of the grammar.

The production part follows the grammar defined in Section 2.3 rather closely. One difference is that the Kleene star (*) in the production `exp` has been replaced with left recursion. The reason for using left recursion instead of (*) is that it will make it easier later on to extend the parser specification to create abstract syntax trees, something we will do in the next assignment. Note that Beaver is an LALR parser generator, and thus supports left recursion.

▶ Make sure you understand the parser specification in Figure 2.

2.4 The Gradle build script

Build scripts are used to specify how a project should be built by a build system. This is useful as a project becomes larger. For a simple single-file Java project you could easily compile it from the command line, but it would take too long to type in the commands to generate the scanner and parser and package

```
plugins {
 id 'java'
 id 'idea'
}
...
// This specifies where the source code is located:
sourceSets {
  main.java.srcDirs = [ 'src/java', 'src/gen' ]
  test.java.srcDirs = [ 'src/test' ]
}
...
// Configuration for the generated Jar file.
jar {
  manifest.attributes 'Main-Class': 'lang.Compiler'
  destinationDir = projectDir
  archiveName = 'compiler.jar'
}

// Before compiling, we should generate some Java code:
compileJava.dependsOn 'generateScanner', 'generateParser'

task generateScanner(type: JavaExec) {
  description 'Generates the scanner with JFlex.'
  ...
}

task generateParser(type: JavaExec) {
  description 'Generates the parser with Beaver.'
  ...
}

// The following makes the clean task also remove generated code:
clean.dependsOn 'cleanGeneratedJava'
task cleanGeneratedJava(type: Delete) { ... }
```

Figure 3: Parts of the Gradle build script `build.gradle`

the compiler in a Jar file each time for our assignments. The build system helps by automating these tasks.

We use the Gradle build system for the assignments. The build scripts are written in the Groovy language. Most other build systems have similar features but use different languages in their build scripts.

When running Gradle we specify what it should do by listing one or more task on the command line. For example, `./gradlew jar`. Here are some of the tasks in the CalcParse build script (`build.gradle`):

jar Generates `compiler.jar`. The name of the Jar file is specified inside the `jar { ... }` block.

test Runs all tests in `src/test`.

generateScanner Generates the scanner with JFlex.

generateParser Generates the parser with Beaver.

compileJava Compiles all java code in `src`

clean Removes all generated (scanner, parser) code in `src/gen`.

We mainly use the `jar` and `test` tasks. The code generation and compilation tasks are automatically run as needed before running tests or building the Jar file. Relevant parts of the code from the CalcParse build script are shown in Figure 3.

Gradle uses online repositories to download Jar files for JFlex, Beaver, and JUnit, as needed to build and test the CalcParse project. Consequently, you must have an internet connection the first time you build the project, but this is not needed when re-building the project on the same computer. If Gradle could not download the required Jar files (dependencies), it reports an error message similar to this:

```
Could not resolve all dependencies for configuration ':testCompileClasspath'.
> Could not find junit:junit:42.
  Searched in the following locations:
      https://repo1.maven.org/maven2/junit/junit/42/junit-42.pom
      https://repo1.maven.org/maven2/junit/junit/42/junit-42.jar
  Required by:
      project :
```

3 MiniS

Figure 4 shows the context-free grammar for the MiniS language, a small language with just a few statements and expressions. A program is just a single statement. You may assume that `<ID>` and `<NUMERAL>` are the same as in the Calc language.

program	→	stmt
stmt	→	forStmt `|` ifStmt `|` assignment
forStmt	→	`"for"` `<ID>` `"="` expr `"until"` expr `"do"` stmt `"od"`
ifStmt	→	`"if"` expr `"then"` stmt `"fi"`
assignment	→	`<ID>` `"="` expr
expr	→	`<ID>` `|` `<NUMERAL>` `|` `"not"` expr

Figure 4: Context-free grammar for MiniS

3.1 Example program for MiniS

▶ Write an example program in MiniS that uses all constructs in the grammar in a syntactically correct way.

3.2 Scanner for MiniS

▶ Generate a scanner for MiniS using JFlex.

To make a scanner for the MiniS language, it is convenient to start by copying the CalcParse directory, and adapt the CalcParse scanner for the MiniS language. Run `./gradlew clean` to remove any generated files from CalcParse.

Adapt the scanner specification to MiniS, and run `./gradlew compileJava` to generate the scanner and compile. There will be compile errors, since the actions contain references to the `Terminal` class that is generated by Beaver, which is not yet adapted.

If at any point you want to rebuild your project from scratch, you can run `./gradlew clean` to remove generated files. Then build again.

3.3 Generate a parser with Beaver

- Generate a parser for MiniS using Beaver.

  Adapt the parser specification to MiniS. Run `./gradlew compileJava` to generate the parser and compile.

  Make sure everything compiles. Then generate the compiler Jar file (`./gradlew jar`), and try it out on your example program. If you have difficulty in making the parser work correctly, construct smaller examples to pinpoint the problem.

- Remember to commit often, as soon as you have got some small new functionality or test case to work.

- Turn your examples into automated test cases. Add test cases to make sure you check everything interesting. Make sure you have test cases both for parsing syntactically correct and incorrect programs. Make sure you can run them all using `./gradlew test`, and that they all pass. You should have at least the following kinds of test cases:

  - a program that uses all language constructs
  - the shortest possible program
  - a program that gives a parsing error
  - a program that gives a scanning error

- Remember to commit often.

3.4 Support integer numerals

- Change the definition of Numerals, so that also integers like `42` are supported (not only numbers with a decimal point like `42.0`). Adapt and extend the test suite as needed.

- Remember to commit often.

3.5 Hand-coded recursive-descent parser

- Implement a recursive-descent parser for MiniS by hand, not using Beaver.

  You should use the scanner for MiniS that you have generated with JFlex. Your parser can get tokens by calling the method `nextToken()` in the generated scanner. Token constants are defined in the file `LangParser.java`, which is created when generating the parser with Beaver. (If you didn't use Beaver at all, you would have to construct this file manually. So we are just reusing this file for convenience.)

  Make sure not to place your hand-coded parser in the `src/gen/` directory, as all files in that directory are removed by the build script when doing `./gradlew clean`.

  As a help, see the code in Appendix B. You can also get inspiration for your implementation by studying examples in the textbook or in the lecture slides.

  Create a new main program `RecursiveDescentCompiler.java` that uses your recursive-descent parser instead of the one generated from Beaver. To run the compiler, first build the Jar file with

  ```
  ./gradlew jar
  ```

  You can then run the new main program by giving it the Jar file on the class path, as follows:

  ```
  java -cp "compiler.jar" lang.RecursiveDescentCompiler minis-file
  ```

The compiler should only decide if an input program is correct according to the grammar – you should not build an abstract syntax tree. In case of erroneous input the compiler should indicate the first error with some explanation. The grammar is so simple that you should be able to implement the parser right away, without constructing a table first.

It is strongly adviced that you work incrementally. Start by getting the parser to work for the shortest possible program. Then extend your parser gradually until it can parse the whole MiniS language.

Make sure to run the compiler on all your existing test examples. (You can run these examples one by one, you don't need to automate them.)

▶ Remember to commit often.

4 What to show and discuss with assignment supervisor

When you are ready with the assignment, these are typical things your supervisor may ask you to do:

- Show your test cases.
- Show that you can run your generated parser (`compiler.jar`) on an example
- Show that you can run all your test cases automatically on the generated parser (`./gradlew test`).
- Show that you can run your hand-written parser on an example.
- Show that your git repository has several commits.
- What does your scanner and parser specifications look like? Is there anything you would like to ask or comment about them?
- In the scanner specification, what would happen if you replace `[^]` with `[^]+` in the last token (the error fallback)?
- Comment on the implementation of your hand-written recursive-descent parser. Was it easy to implement? Is it always that easy, for any grammar?

Appendix A Agile programming methodology

During the assignments you should use a programming methodology with small increments, automated tests, and pair programming. These techniques are part of XP (extreme programming) which is an agile programming methodology, and are briefly explained below.

Small increments

Work in small increments: For the scanner, start with handling only a very simple part of the language, e.g., blanks and some simple keyword. Make sure it works. Then move on and add more kinds of tokens. Similarly, when you work with other assignments: start with getting something simple to work. Then add a small piece of functionality and make sure it works before you continue with the next increment.

Working in small increments gives you good chances of finding sources of erroneous behavior quickly. In each increment, you should use Test First and Automated Tests, as explained below.

Test First

Here are the steps you do in Test First:

- *Make a test:* Create a new test case. You do this *before* you implement the corresponding production code that makes the test succeed. This is important to get focused, so you work only on a small functionality increment at a time, and so you know *what* it is you try to accomplish.
- *See it fail:* Before you start implementing the production code for the test, run the test. Of course, the test usually fails. But it is good to see it fail, because then you know you are testing something interesting. If the test succeeds, make sure you know why: Is the test wrong? Or is the functionality perhaps already there?
- *Make it right:* Implement the production code so that the test succeeds.
- *Refactor:* Now that you have working code, look over the test code and production code and refactor it, if needed, to make it cleaner. For example, rename things to better names, eliminate duplicated code, introduce new methods if some methods have become too long, etc.

Automated tests

The test cases should be automated so that you can run all of them with a single command. You can use JUnit to automate the tests, as done in this assignment. By running *all* tests after a change, you can check if everything that used to work (i.e., as covered by your test cases), still does work. This way of running a large suite of existing tests is also called *regression testing*: you run it to make sure the code does not regress to a less developed stage.

Automating tests allows you to work with greater confidence: If you happen to break something that worked previously, you will find out as soon as you run your test cases. Run them often! And make sure all current tests succeed before moving on and adding the next test case!

Pair programming

You will do the assignments in pairs. Practice pair programming. The person at the keyboard is called the driver. The person beside is called the navigator. Switch roles often so that both get to drive and to commit. Talk constantly with each other about what you are doing.

If you are collaborating with online meetings, you can screen share to discuss while you are programming, and commit often to the common repository in order to switch frequently who is coding on a particular part.

For more advice on pair programming, see e.g. the article, "All I really need to know about pair programming I learned in kindergarten"[2]. The article discusses how simple kindergarten rules like Share Everything, Play Fair, and Don't Hit Your Partner translate to pair programming.

[2]L. A. Williams, R. R. Kessler: All I really need to know about pair programming I learned in kindergarten, *Communications of the ACM*, 43(5):108-114, May 2000. `https://dl.acm.org/doi/10.1145/332833.332848`, or preprint at `https://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.PDF`

Appendix B Recursive descent parser – template

You can use this template to write your recursive descent parser:

```
package lang;
import lang.ast.LangParser;
import lang.ast.LangScanner;
import static lang.ast.LangParser.Terminals.*;

/**
 * Abstract base class for recursive decent parsers.
 * You should implement the parseProgram() method to parse a MiniS program.
 */
public abstract class RDPTemplate {
  private LangScanner scanner;
  private beaver.Symbol currentToken;

  /** Initialize the parser and start parsing via the parseProgram() method. */
  public void parse(LangScanner scanner) {
    this.scanner = scanner;
    parseProgram();
    accept(EOF); // Ensure all input is processed.
  }

  protected abstract void parseProgram(); // TODO

  /** Returns the current token without proceeding to the next. */
  protected int peek() {
    if (currentToken == null) accept();
    return currentToken.getId();
  }

  /** Read the next token from the scanner. */
  protected void accept() {
    try {
      currentToken = scanner.nextToken();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }

  /** Ensure the current token is of a certain type; then read the next. */
  protected void accept(int expectedToken) {
    if (peek() != expectedToken) {
      error("expected token " +
          LangParser.Terminals.NAMES[expectedToken] +
          " got token " +
          LangParser.Terminals.NAMES[currentToken.getId()]);
    }
    accept();
  }

  protected static void error(String message) {
    throw new RuntimeException(message);
  }
}
```

Appendix C Running from Eclipse

These instructions are written for Eclipse 4.3 (Kepler) on Mac OS X. The exact commands may differ on other platforms or Eclipse versions.

Workspace. We suggest that you create a new Eclipse workspace for this course. For example, give it the name `edan65`. Create the workspace directory, then download CalcParse into that directory. Then start Eclipse and go to that workspace.

Importing CalcParse. CalcParse is an Eclipse project, i.e., it has a `.project` file. Import it by **Package Explorer→Import...→General→Existing Projects into Workspace**

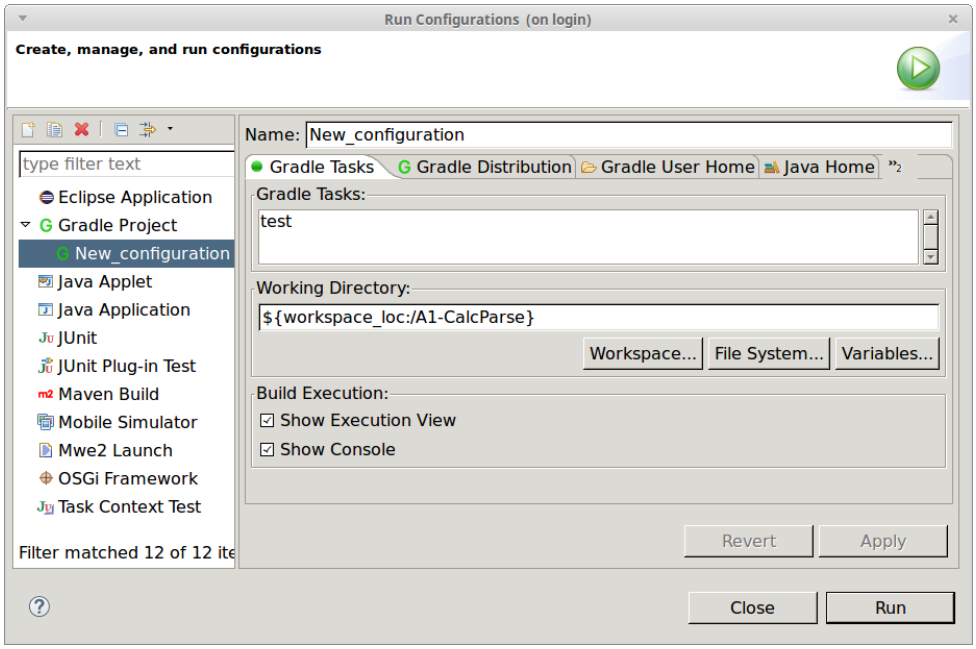
Create the MiniSParser project. Your project MiniSParser can be created by copying CalcParse inside Eclipse. Select the CalcParse project, copy it, and paste the new copy in the Package Explorer.

Note that if you have already copied the project from outside of Eclipse, for example from the command line, you might need to adjust its Eclipse name that is stored in the `.project` file. The Eclipse name of the project might be different from its directory name, and Eclipse will refuse to import a project that has the same Eclipse name as another existing project. Just edit the `.project` file in that case.

Build and Refresh. CalcParse (and MiniSParser) will not automatically build correctly because it is lacking generated files. You have to run the build file with the `test` task to generate the scanner and parser first. This can be done either by running `./gradlew test` in the command line (remember to refresh the Eclipse project after doing so), or by setting up to run Gradle via Eclipse.

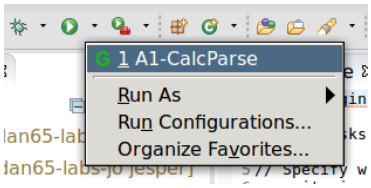
To run Gradle in Eclipse, follow these steps:
Run→Run Configurations...→Gradle Project (right-click)→New
Then type `test` in the *Gradle Tasks* text box. In the *Working Directory* text field you should enter the following: `${workspace_loc:/A1-CalcParse}`

You should have the following:



After the build, you need to *refresh* the project so that Eclipse scans the directory to become aware of the new files created by the build. Do this by **Select project→Refresh** This must be done after each time you rebuild the project.

Run configurations Since you will re-run the Gradle build often, it is recommended to rename the run configuration created in the previous step to something distinct, for example `A1-CalcParse`. To re-run the Gradle build, you can access the configuration via the regular Run menu in Eclipse:



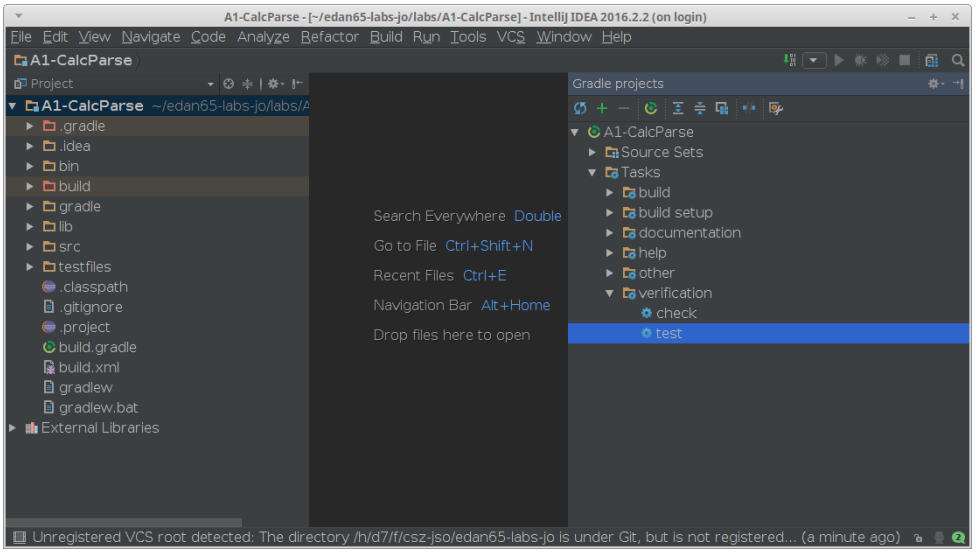
Running the tests You can run the tests by
Select project→Run As→JUnit Test

Opening text files In the labs we will use a lot of different kinds of text files: input files to generators, test files, input files for our generated compilers, etc. If you find that a file does not open in Eclipse when you double-click on it, you may have to configure Eclipse to recognize a certain file type as being a text file that should be edited with a text editor. You can do this in the preferences:
Eclipse or Window→Preferences→General→Editors→File Associations
Add a new file association, for example for `.out` files, and associate it with the Eclipse Text Editor.

Appendix D Using IntelliJ IDEA

IntelliJ can import Gradle projects directly. Alternatively, run the command `./gradlew idea` in the project directory, then open the project like a regular IntelliJ project.

To run the Gradle build within IntelliJ, open the Gradle window: **View→Tool Windows→Gradle**
You should then see the Gradle tool window on the right-hand side:



The test task can be run by double-clicking it in the Gradle tool window: **Tasks→verification→test**

When using IntelliJ, you won't need to manually refresh the project after each rebuild.