Lund University
Computer Science
Niklas Fors, Görel Hedin, Christoff Bürger

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*, `http://jflex.de`
- The parser generator *Beaver*, `http://beaver.sourceforge.net/`. Beaver is based on the LALR parsing method.
- The build tool *Ant*, `http://ant.apache.org`
- The testing framework *JUnit*, `http://junit.org`

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

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 Piazza 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 (EDA260), you will be familiar with this way of working.

These instructions assume that you are running on a platform with Java and Ant (like the LTH student computers). You can also run on your own computer with Linux, Mac OS X or Windows, but you might then need to install Java (version 1.7 or later) and Ant. Furthermore, the instructions assume that you are running from the 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.

2 The CalcParse demo

There is a small demonstration example, *CalcParse*, that contains a scanner and a parser for a calculator language called *Calc*. The first part of the assignment is to download the CalcParse example, and make sure that you can run it, that you understand the implementation, and that you can do small changes to it.

The Calc language contains the following four 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 `http://cs.lth.se/edan65` into an appropriate directory, and unzip it.

The ant build script, `build.xml`, contains targets `build`, `test`, and `jar`, for building, testing, and creating a jar file for the Calc compiler. The target `clean` removes all generated files.

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

```
$ cd <path>/CalcParse
$ ant test
```

This should generate the scanner and parser, compile the resulting java files, and finally run all the tests. No errors or failures should be reported. How many tests were run? 7

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

Generate the jar file *compiler.jar*:

```
$ ant 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.calc
$ java -jar compiler.jar testfiles/example.calc

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

▶ Create a new file containing 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:

`ant-bin`	contains generated class files - used by the build script.
`bin`	contains generated class files - used by Eclipse (if you use Eclipse).[1]
`build.xml`	an Ant build script.
`lib`	contains Jar files for *JFlex*, *Beaver*, *JUnit*, and *JastAdd2*. The last one is not used in this assignment, but is required in subsequent ones.
`src`	contains all source files, both Java files and specification files.
	`gen` contains Java files generated by JFlex and Beaver.
	`java` contains Java files that are not generated. The main program `Compiler` is located in the package `lang`, and is used in compiler.jar. The Java files used for testing are located in the package `tests`.
	`parser` contains the Beaver specification.
	`scanner` contains the JFlex specification.
`testfiles`	contains example programs used in the tests.

▶ 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 `src/java/tests` and `testfiles`. Add a new test in a similar manner, and run all tests using `ant 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 will describe how the grammar has been translated to JFlex and Beaver specifications. Terminal symbols in the grammar correspond to tokens, and are defined in the JFlex specification. Productions in the grammar are defined in the Beaver specification, in terms of the tokens.

[1] The reason for having two directories for class files is that you then can use the build script from inside Eclipse, without the risk of both the build script and Eclipse modifying the same files, which could lead to race conditions and inconsistent files.

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 and understand. Three macros are defined: `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 then fed to the parser and parsed according to the production rules.

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.

```
// 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

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.[2] If this rule matches, then it will throw

[2]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.

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.

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 represents 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 Ant build script

The Ant build script `build.xml` is shown in Figure 3. It is written using XML, a language similar to HTML. The script defines the `Compiler` project using a number of *properties*, *taskdefs*, and *targets*. A property is a symbolic name for a string that can be used later in the script. A target consists of a number of *tasks* that will execute commands or programs in the operating system. Some tasks are predefined and others can be introduced using the `taskdef` construct. A target can be executed from the command line, as we saw in Section 2.1.

The `lib.dir` property names the directory containing library jar files, for example the jar files for JFlex and Beaver. A property can be used by enclosing it with `${...}`, for example `${lib.dir}`. You can see how `lib.dir` is used in the classpath in the `taskdef` constructs.

The `build` target has two tasks. The `mkdir` task will create a directory and the `javac` task will compile the Java files. Each task specifies arguments for the execution. The `build` task also *depends* on another target `gen`, meaning that the target `gen` will execute before the target `build` whenever the target `build` is invoked. The target `gen` *depends* on the targets `scanner-gen` and `parser-gen`, which will generate the scanner and the parser, respectively.

```
<project name="Compiler" default="build">
    <property name="jar.file" value="compiler.jar" />
    <property name="lib.dir" value="lib" />
    <property name="src.dir" value="src" />
    <property name="gen.dir" value="${src.dir}/gen" />
    <property name="bin.dir" value="ant-bin" />
    <property name="classpath.lib"  value="..." />

    <taskdef name="jflex" classname="jflex.anttask.JFlexTask"
        classpath="${lib.dir}/jflex-1.6.0.jar"/>
    <taskdef name="beaver" classname="beaver.comp.run.AntTask"
        classpath="${lib.dir}/beaver-ant.jar"/>

    <target name="build" depends="gen">
        <mkdir dir="${bin.dir}" />
        <javac srcdir="." destdir="${bin.dir}" classpath="${classpath.lib}" />
    </target>

    <target name="gen" depends="scanner-gen, parser-gen">
    </target>

    <target name="scanner-gen">
        <mkdir dir="${gen.dir}" />
        <jflex file="${src.dir}/scanner/scanner.flex" outdir="${gen.dir}/lang/ast" nobak="yes"/>
    </target>

    <target name="parser-gen">
        <mkdir dir="${gen.dir}" />
        <beaver file="${src.dir}/parser/parser.beaver" destdir="${gen.dir}"
            terminalNames="yes"/>
    </target>

    <target name="test" depends="build">
        ...
    </target>

    <target name="jar" depends="build">
        ...
    </target>

    <target name="clean">
        ...
    </target>
</project>
```

Figure 3: The Ant build script `build.xml`

The Ant tool provides predefined tasks for `mkdir` and `javac`. But we also want to run JFlex and Beaver, that the Ant tool does not know about. However, both JFlex and Beaver are prepared to be run from Ant, so they each include an "ant task" class in their jar files. The build script defines two new tasks, `jflex` and `beaver`, using the `taskdef` construct. Each `taskdef` states the name of the task, the name of the ant task class, and the classpath for where to find the ant task class. The new tasks are used in the targets `scanner-gen` and `parser-gen`. To see what parameters the ant task classes take, you need to look at the documentation for JFlex and Beaver.

▶ Make sure you understand the build script in Figure 3. Find the documentation for the *javac* task (see the ant manual), and the *jflex* and *beaver* tasks (see their respective documentation). What parameters

javac:
-srcdir: Vart javafilerna finns,
-destdir: Vart .class filerna skall läggas, classpath (jars)

JFlex:
-file: Specifikation till jflex,
-outdir: Vart JFlex ska lägga genererade grejer,
-nobak: Gör ingen backup för genererade filer.

Beaver:
-file: Grammer spec,
-destdir: Vart skall genererade filer läggas

to these tasks are used in the build script, and what do they mean?

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 create support for the MiniS language, it is convenient to start by copying the CalcParse directory, and adapt the code to the MiniS language. Run `ant clean` to remove old generated files. Run `ant build`[3] to make sure you start from a consistent state that compiles correctly, before starting to adapt the specifications to MiniS.

Adapt the scanner specification to MiniS, and run the build target `scanner-gen` to generate the scanner. Note that the generated scanner will contain compile time errors, since the actions contain references to the `Terminal` class that is generated by Beaver, and which is not yet adapted.

3.3 Generate a parser with Beaver

▶ Generate a parser for MiniS using Beaver.

Adapt the parser specification to MiniS. Run the build target `parser-gen` to generate the parser (or the target `gen`, which depends on both `scanner-gen` and `parser-gen`).

Make sure everything compiles. Then generate the compiler jar file (`ant jar`), and try it out on your example program.

▶ Add some test cases for both correct and incorrect programs. Make sure you can run them all using `ant 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

[3] Or simply run `ant`, since `build` is the default target.

3.4 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.

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 `ant 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 compile it using `ant build`. Then you can run the compiler directly, without constructing a jar file, as follows:

```
java -cp "ant-bin:lib/beaver-rt.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.)

3.5 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.

4 Topics for discussion

1. Could we replace `[^]` with `[^]+` in the last token (the error fallback)?
2. Implementation of the recursive-descent parser could be done quite easily, just writing code following the grammar. Is it always that easy? What would be an example of a grammar that would be difficult to implement using recursive descent?

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.

Jag tror att det fungerar att byta ut [^] mot [^]+ i detta fallet.

Nej, det är inte alltid så enkelt. Detta fungerar när grammatiken är sådan att den första terminal symbolen talar om entydligt vilken production rule man skall använda.

Exempel 3.10 s 46 i boken

A.1 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.

A.2 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.

A.3 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!

A.4 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. Talk constantly with each other about what you are doing. For more advice, see e.g. the article, "All I really need to know about pair programming I learned in kindergarten"[4]. The article discusses how simple kindergarten rules like Share Everything, Play Fair, and Don't Hit Your Partner translate to pair programming.

[4] 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. `http://dl.acm.org/citation.cfm?id=332848`, or preprint at `http://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.PDF`

B Recursive descent parser – template

A template for a Recursive descent parser is:

```
package lang;
import lang.ast.LangParser;

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

/**
 * Abstract base class for recursive decent parsers. The class provides convenience methods
 * to integrate a scanner, like peeking the next token or reading a certain token. Subclasses
 * must implement the parseProgram() method, which is used as entry point for recursive decent
 * parsing.
 */
public abstract class RDPTemplate {
        private LangScanner scanner;
        private beaver.Symbol currentToken;

        /**
         * Initialize the parser and start parsing via the parseProgram() method.
         * @param scanner providing the token stream to process; used throughout parsing
         * @throws RuntimeException if the program is not syntactically correct
         */
        public void parse(LangScanner scanner) {
                this.scanner = scanner;
                parseProgram();
                accept(EOF); // Ensure all input is processed.
        }

        /**
         * Entry hook to start parsing.
         * @throws RuntimeException if the program is not syntactically correct
         */
        public abstract void parseProgram();

        /**
         * Peek the current token, i.e., return it without proceeding to the next token.
         * @return ID of the current token
         */
        public int peek() {
                if (currentToken == null) accept();
                return currentToken.getId();
        }

        /**
         * Test the current token without proceeding to the next.
         * @param expectedToken token type to test for
         * @return true, if the current token is of the expected type, otherwise false.
         */
        public boolean peek(int expectedToken) {
                return peek() == expectedToken;
        }

        /**
         * Read the next token (invokes the scanner to set the current token to the next).
         */
        public 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.
         * @param expectedToken token type to test for
         * @throws RuntimeException if the current token is not of the expected type
         */
        public void accept(int expectedToken) {
                if (!peek(expectedToken)) {
                        error("expected token " +
                                LangParser.Terminals.NAMES[expectedToken] +
                                " got token " +
                                LangParser.Terminals.NAMES[currentToken.getId()]);
                }
                accept();
        }

        /**
         * Throw runtime exception with the given message.
         * @param message of the thrown exception
         * @throws RuntimeException always thrown
         */
        public void error(String message) {
                throw new RuntimeException(message);
        }
}
```

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 an 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. As you may have noted, CalcParse (and MiniSParser) will not automatically build correctly. You have to run the build file with the `gen` target to generate the scanner and parser first. You do this by
Package Explorer→build.xml→Run As→Ant Build... Then select the `gen` target (and probably unselect the default target), and click `Run`.

After the build, there are probably still compile-time errors reported by Eclipse. 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**

Ant configurations Since you will do `build gen` often (generating the scanner, parser, etc.), you should set up an *ant configuration* for this, and which automatically refreshes the project. Do this by

build.xml→Run As→Ant Build...
Give the configuration a suitable name. Under `Targets` you can select and unselect ant targets as desired. Under `Refresh` you can make sure Eclipse refreshes the project after running ant. Click `Apply` to save the changes in the configuration. Click *Run* to run the configuration.

You will find the configurations you have created under
build.xml→Run As→External Tool Configurations
or conveniently using a button on the workspace toolbar:

To the right of the button is a small menu where you can run your different configurations, or edit them. Clicking on the button will run the latest run configuration.

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.