Dr. Annette Bieniusa
M.Sc. Peter Zeller

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lab 1: Compiler and Language Processing Tools (WS 2018)

Submission Deadline: Wednesday, 07.11.18, 9:00

- Late submissions will not be accepted.
- Submit your solution via your group's repository to a folder named "ex1". Specify the group members (name and email) of your group in a file named `readme.txt` at the top level of your repository.
- Write a readme file (a plain text file, a latex document with PDF output, or a markdown document) and submit it to the repository. The readme should be at most 2 pages long and give a brief overview over your solution.
- Your Git commits show us, who has worked on which part of the code. If this is not visible in the Git commits, please make it clear in the commit message or in the readme file. For example add "pair programming with Bob" to the commit message, when you worked with someone else from your team.
- Your code should be readable and documented with JavaDoc.
- Test your submission with the provided test cases. You will sometimes have to create the matching classes and methods for the Tests to compile. Feel free to add more tests, but do not change the existing test cases. You may add the `@Ignore` annotation to ignore failing test cases.
- Submissions which cannot be built using Gradle (e.g. because of compiler errors) will receive 0 points. If you submit partial solutions, make sure to comment out all the unfinished code, which does not compile yet.

1 Tool Setup (0 points)

1. Make sure that you have the Java JDK version 1.8 (not the JRE) or later installed. Running `javac -version` on the command line gives you the version number.
2. Make sure that you have the version control software Git installed, so that you can access your repository.
3. An integrated development environment (IDE) like IntelliJ IDEA or Eclipse is useful when working with bigger Java projects, so we suggest using one for the exercises.
4. Download the template `ex1_template.zip` from the lecture page and extract the files to a folder named "ex1" in your groups Git repository. Then commit and push the template files to your Git repository.
5. On the command line, change to the directory with the extracted files. In this folder you should see a file `gradlew` and a file named `gradlew.bat`. These scripts will download the build tool Gradle and can be used to build the project.

   First run `./gradlew compileJava` from the command line to run the `compileJava` task of the project (on Windows Systems execute `.\gradlew compileJava` instead). At the end you should see `BUILD SUCCESSFUL` in the output. If you are seeing an error like "Could not find tools.jar" it is most likely means that you have the JRE installed instead of the JDK.

   Besides `build` there are some other useful tasks: The `eclipse` and `idea` tasks generate projects, which you can import into the respective IDE. The `gen` task will only generate the source files. The `build` task builds the project and runs the tests, while the `test` task only runs the tests.

2 Expression Interpreter (10 points)

The goal of this exercise is to write a parser and an interpreter for a small expression language. The expressions can be written according to the following grammar:

$$
\begin{array}{rcl}
\textit{CExp} & \rightarrow & \textbf{if } \textit{CExp} \textbf{ then } \textit{CExp} \textbf{ else } \textit{CExp} \\
& | & \textbf{let } \textit{id} = \textit{CExp} \textbf{ in } \textit{CExp} \\
& | & \textbf{fun } \textit{id} \texttt{ -> } \textit{CExp} \\
& | & \textit{Exp} \\
\textit{Exp} & \rightarrow & \textit{Exp op Exp} \\
& | & \textit{Exp}\ \ \textit{Exp} \\
& | & \langle\text{identifier}\rangle \\
& | & \langle\text{integer literal}\rangle \\
& | & \texttt{(}\ \textit{CExp}\ \texttt{)} \\
\textit{op} & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid < \mid ==
\end{array}
$$

The expression "**fun** *id* **->***Exp*" is a lambda expression as known from the lambda calculus[1] and "*Exp Exp*" is function application, where the left expression evaluates to a function and the right expression to the argument.

Examples of the expressions are:

```
((5*3) + 4)                                          -- evaluates to 19
(fun x -> x * 2) 21                                  -- evaluates to 42
let x  = 5 in (x*3)                                  -- evaluates to 15
let y  = 2 in if (y==2) then (y + 3) else (y-1)      -- evaluates to 5
let y1 = 2 in ((fun x1 -> (x1*3))  y1)               -- evaluates to 6
```

You should start with the provided template project, which already includes a parser and a lexer for a part of the language.

1. **(3 points)** Extend the existing Lexer specification in `src/main/java/exprs.flex` with the following constructs:

   a) identifiers: an identifier starts with a letter followed by any combination of letters, digits and "_".

   b) single line comments: all characters after "`--`" should be ignored until the end of the line.

   c) multi-line comments: all characters between "{-" and "-}" should be ignored.

   d) the missing symbols and keywords required by the given grammar

   Use the terminal names, which are already defined in `exprs.cup`. Test your lexer with the provided JUnit tests in LexerTests.java.

   In your readme file you should describe, how you solved the problem of multi-line comments. Also explain why your lexer recognizes "if" as a keyword and "iff" as an identifier.

2. **(3 points)** Adjust the grammar in `exprs.cup`, so that it accepts the given grammar and builds an abstract syntax tree. Solve the ambiguities in the given grammar as follows:

   - The mathematical operators should be interpreted as usual.
   - Function application groups to the left and has the strongest binding. For example: `f 0 + g 1 2` is equivalent to `(f 0) + ((g 1) 2)`

   Implement the `toString` method for your AST-classes. Print parentheses around every expression in your `toString` method, so that it passes the provided JUnit tests.

   Use the readme file to explain, how you solved the ambiguities in your cup file. If you fail to resolve some conflicts, you can temporarily add additional parentheses to the grammar, so that you get a working version.

[1] If you are not familiar with the lambda calculus, you can still work on all mandatory parts of this exercise. However, we encourage you to read up on the topic, as it is a fundamental part of computer science.

3. **(3 points)** Implement an evaluator that calculates the value of an expression by processing the AST. You do **not** have to handle variables, let- expressions, and lambda-expressions yet. Also you can assume that all expressions evaluate to integer values (Boolean values can be represented by 0 and 1).

   Describe the basic approach of your evaluator in your readme file.

4. *(Optional)* Extend your implementation of the expression language to also evaluate variables, let- expressions, lambda-expressions, and function applications.

   The semantics for the expression `let` x `=` e_1 `in` e_2 is to first evaluate e_1 in the current environment. If evaluation of e_1 results in value v_1, then the expression e_2 is evaluated next with x set to $v1$ and the rest of the environment unchanged.

   The expression `fun` x `->` e evaluates to a closure value, which consists of the current variable environment, the variable name x and the expression e.

   For function applications e_1 e_2, the expression e_1 gets evaluated first. It should evaluate to a closure value containing some variable x, environment Σ, and expression e. Next the expression e_2 is evaluated to a value v_2. Finally, the expression e from the closure is evaluated using the environment Γ with variable x set to v_2.