SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Reinhard Wilhelm
Prof. Dr. Sebastian Hack
Dipl.-Inform. Daniel Grund
Michael Jacobs, M.Sc.

Compiler Construction Project WS11/12

Project task A. General Information

In the practical project you will implement a compiler for a small subset of Java. The milestones of the project roughly reflect the structure of a compiler, which also structures the lecture. Although the milestones will not be graded, you will receive feedback that hints at existing shortcomings in your compiler. The final submission will be graded on the basis of percentage of tests passed and a code review where the group members have to justify their work.

Technical requirements and restrictions:

- You are not allowed to use existing tools that more or less directly solve the assignments, e.g. you must not use lexer or parser generators. If you feel like it, however, you are free to implement your own tools.
- The compiler itself must be written in C or C++.
- Executing the command `make` in your top-level directory must produce an executable file `mjavac` in that same directory.[1]
- Upon acceptance of a source program, `mjavac` must terminate with return code 0. Rejection of a source program must be indicated by return code 1 and should print an error message.
- The individual assignments will refine the requirements.

Project task B. Lexer

To get started:

- Set up a version control system of your choice.
- Go to the web page of the lecture and download the project materials.
- Examine the contents of the starter kit. It provides a Makefile and the public tests for the lexer. Each `.ref` file contains the expected output for the respective `.java` file.
- Read the language specification of MiniJava and identify its symbols/tokens.

Implement a lexical analysis for MiniJava. You do not necessarily have to follow the automata approach presented in the lecture.

- Files that contain only valid tokens, comments, and whitespace must be accepted and `mjavac --tokens [file]` must print the corresponding token stream to the standard output. For example, the line `int i = 423;` must produce the following token stream:
  ```
  int
  IDENT i
  =
  INTEGER_LITERAL 423
  ;
  ```
  Remember that whitespace and comments are not part of the token stream.
- Files that contain lexical errors must be rejected.
- The running time of your parser should be proportional to the input size.
- Write more test cases, especially some that test the rejection capabilities of your lexer.
- The next project task will be to implement a parser. You therefore might want to structure your source code in a way that separates lexing and printing of tokens. . . .

Please check in your solution into your repository until 2011-11-03, 12:00. Solutions submitted later may not receive feedback.

[1] Do not even think about trying to include malicious or “funny” code.