Compiler Construction
Prof. Dr. Florian Heinz
florian.heinz@oth-regensburg.de
Study Project

OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

DCMC: Study Project

In the course "Compiler Construction" you have the chance to work on a study project closely related to the topics of the lectures and exercises

- Assessment mode:
  - Source code archive with the study project implementation
  - 2-5 pages description of your project (structure, characteristics, examples, etc.)
  - Short presentation (only live demo and Q&A, 10-15 mins, no slides needed) of your work
- Evaluation: Performance verification (graded)
- Deadline: Wed, 2025-02-05 23:59:59
- Submission: Send the archive with source code and description via E-Mail to florian.heinz@oth-regensburg.de

Task:

Your project should demonstrate your understanding of the lecture topics.

Mandatory parts:

- Building a lexicographical scanner
- Building a syntactical parser
- Creating an abstract syntax tree
- Then choose:
  - Create an interpreter to execute the program that is represented by the Abstract Syntax Tree by traversing it
  - Compile the Abstract Syntax Tree to machinecode (e.g. bytecode for a vm)

Optional parts:

- Post-process your abstract syntax tree for optimizations
- Run further semantic checks on the abstract syntax tree (e.g. type checks or similar)

General task description:

Your task is to create a programming language. There are no high requirements regarding the programming language design, the focus is lexing, parsing, abstract syntax trees and interpreting/compiling.

The programming language should be general purpose, three programming tasks should be fulfilled with it for demonstration. You may of course optimize it for specific needs.

The syntax and semantics of the programming language should be very individual and should deliberately deviate from other programming languages, specifically your syntax should not be too close to the syntax of the language C! This includes, but is not restricted to:

- Number formats
- String literals
- Identifiers
- Operators
- Keywords
- Control structures
- Functions
- Variable handling
- Data types
- ... and so on

The above listed features should be implemented as far as possible in some way.

Some ideas for possible optional features include:

- Static type system (checked at compile time)
- Advanced control structures (for example, a while-loop that has some kind of else-branch, if the condition is not fulfilled already before the first iteration)
- Special syntax for iterating over a range with specific step widths
- Variadic functions
- More individual syntactic elements for special purposes

Compiling:

If you choose the way to compile your program to bytecode, you can use the virtual machine VM3 introduced in the lecture

You can get it by cloning with GIT:

```
$ git clone https://github.com/fwheinz/vm3
$ cd vm3
$ make
```

The project contains an example lexer and parser/bytecode compiler **prolang**. You can use it as reference to get an idea, how certain constructs can be implemented, but it is in general unsuitable as a base for your own project. Start the implementation from scratch or based on the past exercises, if you have accomplished these. You can change the name of the target in the Makefile to reflect your own program name.

Feel free to extend and modify the virtual machine. Implement custom native functions and/or opcodes in **custom.c**, so you can still update the project without conflicts. But you can also modify other parts if you feel it is necessary.

Demonstration:

Implement four programs with your programming language:

- "Frame"
- "Towers of Hanoi"
- "Rock Paper Scissors Lizard Spock"
- "Language Demo"

Frame: The user enters a sentence, which is split into individual words. Each word is printed in a separate line and enclosed by a rectangular frame of asterisk (*)

```
Please enter sentence: This is a nice Frame
*********
* This  *
* is    *
* a     *
* nice  *
* Frame *
*********
```

Tower of Hanoi: The user enters a number of disks on Tower 1 and the program shows the sequence that transports it to Tower 3 using the auxiliary Tower 2. The usual rules for "Tower of Hanoi" apply. This problem should be solved recursively!

Rock Paper Scissors Lizard Spock: Implement the game Rock Paper Scissors Lizard Spock with a computer adversary. The player inputs his choice, then the computer reveals his choice and decides, who has won. The computer plays fair until he loses two times in a row; in that case (s)he cheats to win the round. Also track wins and losses.

Language demo: Your own program that demonstrates the specifics of your language

Bonus: Additional non-trivial programs give bonus points

Final words:

- If you have your own cool idea for this study project, that considerably deviates from the task described here, write me an e-mail and I will consider it!
- Do your task step by step, starting with the lexer, continuing with the parser, the abstract syntax tree and finally executing the program one way or another. This is not a trivial task, so don't get frustrated easily when you are stuck somehow.
- If you use VM3 and you can rule out own errors, chances are you might have hit a bug. Please report it to me and I will try to fix it
- There is help:
    - Have a look at the example implementations provided in the lecture slides, ELO and the github repository of VM3
    - Ask for support via email. You can either attach your code or send a git repository link. Provide all information necessary to reproduce the problem you have.

Good Luck!