Compiler Construction

Jonas Bork Jørgensen - jonjo21
Malthe Hedelund Petersen - malpe21
Mikkel Nielsen - mikkn21
Sofus Hesseldahl Laubel - slaub21

Supervisor: Jakob Lykke Andersen, Associate Professor, Ph.D.

February 2024

A compiler is software that is given a program written in a pre-defined language and outputs the program in another language. Compilers are widely used in software development as many popular languages, such as C, C++ and Rust, use them. A very useful application of compilers is to define a high-level language and create a compiler for that language to a lower-level language, allowing you to abstract away the details of the lower-level language which makes it easier to create larger programs and projects.

Our proposed Bachelor project aims to write a compiler for a language we design, which our compiler can then output in x86-64 Linux assembly. Building a compiler will give us a deeper understanding of how programming languages in general work, which will aid us in our further work and studies due to how central programming languages are in computer science. The compiler will be written in C++ and the structure of the compiler will follow the layer structure defined in "Modern Compiler Implementation in C" (Andrew W. Appel) The structure will be modified to better fit the use of particular libraries in C++, e.g., the scanner and parser phases will be implemented using the Boost Spirit library which uses Parsing Expression Grammars (PEG), that aims to approximate the Extended Backus-Naur Form (EBNF). Boost.Spirit was chosen because it is well-established with a dedicated community, good documentation, as well as its very high performance. Furthermore, it allows you to define the grammar directly in C++. However, Spirit comes with some trade-offs, such as increased compilation time of our code (not the code our compiler outputs) and increased complexity.

To start we will define and implement a compiler for a minimal language with the following features:

- Primitive types: booleans and integers.
- Aggregate types: arrays and structures.

- Structure mechanisms: if statements, while loops, and functions that can be nested.
- Simple optimizations of the generated code, e.g., peephole optimization.

We will define and implement multiple extensions, e.g., from the following list of possibilities:

- Extending structures to full classes, including access specifiers (public, private) and single inheritance with dynamic dispatch of methods.
- Deallocation of arrays and structures/classes, including garbage collection.
- More complicated optimizations of the generated code, e.g., liveness analysis, register allocation via graph colouring, and function inlining.
- Overloading of functions/methods, potentially including operator overloading.
- Support for generic programming, e.g., templating.
- Partial- and higher-order functions.