# Compiler Construction

## History

- The first computer were programmed by punched tapes. (which probably contained a type of microprograms.)
- The ENIAC was programmed by setting switches and connecting cables. Entering a program took 3 weeks. Later, ENIAC was changed to store the program in memory.
- Assembly Languages originate in the fifties.
- FORTRAN (1953) and Lisp (1958) seem to be the oldest programming languages.
- $C$ is from 1972, and still in good health.
- $C^{++}$ is from 1983, Java from 1992.

## Assemblers

Assemblers make it possible to write machine code in a readable way. Typically, an assembler performs the following tasks:

- Replace readable instructions, like for example `move D2, D3` by binary operation codes.
- Replace characters, integers, floating point numbers by their binary representations.
- Replace labels by addresses.
- Make some simple decisions. (about short or long jumps, long or short addresses, alignment)

Assembler programs are not portable, because they run only on one processor.

## Compilers

In its simplest form, a compiler reads a human readable input file, and outputs an executable.

Modern compilers are much more complex. The GCC has interfaces to many languages, many intermediate representations. It can generate code for many distinct machines.

If one wants GCC on a new machine, one only needs to add a code generator for the new machine.

## Interpreters

An interpreter stores the program in some intermediate representation. Usually, the intermediate representation is portable (like Java bytecode). The internal representation usually preserves information about the original program, which makes debugging easy. Interpreted programs are slower than compiled programs

There exist a lot of interpreters, like Python, Java, $C^{\#}$, Prolog. Some interpreters are (Java, $C^{\#}$) are almost compilers. Only the last stage, translation to native machine code, is missing.

## Convergence of Languages

Once (in 70ties), there were imperative languages, which had static variables, were command oriented, did not suppport recursive data structures, but efficient.

On the other hand, there were functional languages that supported recursion, were based on a mathematical model of computation, and which had automatic memory management.

The distinction is not so sharp anymore. Java, $C^{\#}$, Python, Ruby support recursion and have automatic memory management.

## True or False?

- In order to be a good driver, you need to understand how the car works.
- In order to be a good piano player, you need to understand how the piano works.
- In order to be a good photographer, you need to understand how a camera works.
- In order to be a good programmer, you need to understand how the compiler works.

## Compilers are Complex

Writing a small interpreter for a toy language is doable.

Only very few people write (or design) complete compilers. I know only two people who actually did this. (Tannel Tammet (Scheme), Michał Moskal (Nemerle))

(But everyone can contribute to GCC.)

## A Very Rich Subject

Compiler Construction is a Big Success of Theoretical Computer Science:

- Non-deterministic finite automata are used for token analysis.
- Push down automata are used for parsing.
- Rich type systems are used for type analysis.
- Logical expressions are used for code optimization.
- Tree automata are used for machine code generation.

## In the Future?

- Theorem proving used for Code Optimization?
- Statistical Methods for Code Optimization?

## Why take this Course

You will learn a lot of nice mathematics, nice programming techniques.

Many of these techniques (parsing, tokenizing, tree automata) have applications outside of compiler construction. (Namely, everywhere were complex input has to be analyzed)

(B.t.w., answering a question 4 slides back, if you need to understand the compiler in order to write better programs, there is something wrong with the compiler. It should be enough to know the language standard.)

## Example: A simple pocket calculator

Imagine a simple pocket calculator. (But it knows about priorities of operators.) Its input consists of:

- Floating point numbers, integers.
- Binary operators $+, -, *, /, \hat{}$
- Unary operators $-, +$.
- A postfix operator !.
- An =-key, which terminates all computations.
- Parentheses ( ) for grouping.

Tasks of the simple calculator:

- Structure the input (which is just a stream of key presses) into numbers and operators.
- Determine the order in which the calculations have to be made. In $1 + 2 * 3$, the addition has to be postponed. In $(1 + 2) * 3$, it is carried out immediately.
- Perform the calculations.

## Calculator vs Compiler

There are some similarities and some differences.

- A calculator must be liberal, i.e. accept as many inputs as possible. A compiler must reject every input that does satisfy the language standard.
- Both are confronted with a set of key presses (or characters), and have to structure the input according to some vocabulary. After that, they have to build a kind of tree representation, in order to determine the meaning of what the user typed.
- A pocket calculator can read the input only once. A compiler can read the input many times. (In practice, only assemblers do this.)

## Schedule of the Lecture

- Tokenizing. (Three lectures.)
- Parsing. (Three lectures.)
- Variable Lookup, Type Checking. (One lecture).
- Compilation of $C$. (Two lectures).
- Data Flow analysis. SSA-Form. Optimization. (Two lectures)
- Register Assignment. Code generation. (Two lectures).

## Requirements

- You must be good at $C$ or $C^{++}$.
- You must be good at theory.