Compiler Construction

History

- The first computers 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 were developed 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 1994, $C^{\#}$ from 1999.

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` or `moveq #$3, --(A7)` by binary operation codes.
- Replace characters, integers, floating point numbers by their binary representations.
- Replace labels and variable names by addresses.
- Make some simple decisions. (about short or long jumps, long or short addresses, alignment, relative or absolute jumps)

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

Compilers

In its simplest form, a compiler reads a text file, like for example

```
#include<iostream>
int main(int argc,char*argv[])
{std::cout<<"hello world!\n";return 0;}
```

and outputs an executable.

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

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

Interpreters

An interpreter stores the program in some intermediate representation. The interpreter looks up instructions in a table, and decides what should be done during execution.

Advantages: A lot of information about the original program is preserved, e.g. variable or procedure names. This makes testing and interaction with the program easier.

Interpreters usually do more run time checks. (But I see no reason why compiled code could not do the same checks.)

Usually (e.g. in Prolog or Caml), all input is parsed by the same parser, so that the program has essentially the same syntax as the data. This is nice if the syntax is right for your application, but can be a problem otherwise.

The intermediate code is portable to every computer for which an interpreter exists.

Interpreters

Disadvantages: Instructions have to be looked up by the interpreter, possibly variable and function names have to be looked up. This makes execution slower (and use more battery power.)

In beginning of 2000s, it was stated (by the Java lobby) that Jave bytecode is the future, because efficiency does not matter anymore.

Currently, there is a trend to return to compiled code on mobile devices.

Actually, compilation and interpretation are just the ends of a continuous scale.

Fully interpreted: Prolog, Scheme, Python.

Intermediate code: Java, $C^{\#}$, PHP.

Fully compiled: C, C^{++}, Fortran.

Convergence of Languages

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

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

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

Compilers Construction Reborn in 2000s

In the early 2000s, it was generaly believed that everything about compilers was known. We knew how to tokenize, how to parse, how to typecheck, how to optimize and how to generate code.

- In the beginning of 2000s, SSA (1991?) became mainstream.
- Intermediate representation got standardized in LLVM.

  Independent tools, working on LLVM, became available.

  The intermediate representation GIMPLE of `gcc` was badly documented, and unseparable from GCC.

  (Actually, the LLVM site is too negative about GIMPLE, you can get it using `-fdump-tree-gimple`. It seems they improved something.)
- Abstract Interpretation became more influential.

Compilers are Complex

Writing a small interpreter for a toy language is quite doable. A single person could probably still write a working C compiler, if he is willing to spend one or two years to this.

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` or `clang`.

Such big open source projects are a true realisation of communism, because everyone is working for free!

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.
- Polymorphic type systems are used for type analysis. (Even in *C*.)
- Symbolic computation and abstract interpretation are used for code optimization.
- Non-deterministic tree automata are used for machine code generation.

In the Future?

- Theorem proving for code optimization? (proving pointer distinctness.)
- 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, type checking, tree automata) have applications outside of compilers. (Namely, everywhere were complex input has to be analyzed)

It is nice to understand how the things work!

Example: A simple pocket calculator

Imagine a simple pocket calculator. (Simple, 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, and give understandable error messages about it.
- Both are confronted with a set of key presses (or characters), and have to structure the input according to some grammar. Based on the resulting parse, they have to decide on how to proceed.
- A pocket calculator can read its input only once. A compiler can read the input many times, and process it in many passes.