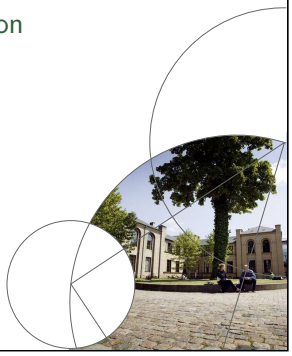
Faculty of Science

Compilers (Oversættere): Course Organisation / Introduction

Jost Berthold and Cosmin Oancea
{berthold,cosmin}@diku.dk
Department of Computer Science



Compilers, 2012

Denne forelæsning afholdes på engelsk.
The course language will be English.

1. Course Organisation
   - Contents, Goals, and Format of the Course
   - Exercises and Exam Schedule
   - Reading Material
2. Compilers: From Source Code to Machine Code
   - An Introductory Example
   - Compilation Phases (Overview)

Learning Goals of the Course

- Competences (kompetencer)
  - Design and implement a compiler from a high-level language to machine code;
  - Evaluate and use appropriate tools and libraries for compilation;
  - Evaluate resource (time and space) consumption of high-level programs from their translation into machine-level programs.
- Skills (færdigheder)
  - Competent usage of tools for lexical and syntactical analysis;
  - Describe and evaluate compiler development in written form.
- Knowledge (viden)
  - Divide compilation into phases with their particular purpose
  - Apply theoretical insight on: Formal languages, regular expressions, context-free grammars, graph colouring
  - Explain how compiler tools work, and their limitations

Contents and Goals for Today

1. Course Organisation
   - Contents, Goals, and Format of the Course
   - Exercises and Exam Schedule
   - Reading Material
2. Compilers: From Source Code to Machine Code
   - An Introductory Example
   - Compilation Phases (Overview)

Goals (for 1 and 2)

- Understand the course organisation
- (Cursorily) know the phases of a compiler and their purpose
- Develop ideas about code analysis and code generation

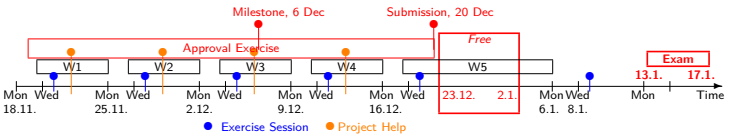
(more on 3 later. . .)

Course Format: Lectures, Exercises, Instructors

- Two weekly lectures: Monday and Wednesday
  - First part: Jost Berthold, second part: Cosmin Oancea
  - See lecture plan on Absalon for topics
- Exercise sessions with instructors:
  - Weekly on Wednesday afternoon
  - Complementing exercises for the lecture (see lecture plan on Absalon for exercises)
- Project help time:
  - Weekly on Friday afternoon (13-16)
  - Opportunity to ask individual questions and get technical help
- Discussion Forum in Absalon

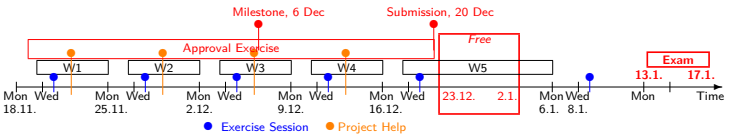
Instructors: Andreas Bock, Niels Serup, Simon Shine, Kristoffer Søholm, Jonas Brunsgård

Exercise and Exam Schedule



- Weekly Exercises: (W) 5 individual assignments
  - Need 4 approved exercises for admission to exam.
  - Published Tuesday afternoon, due Sunday 23:55
    (W1-W4 can be resubmitted to improve previous solution)
- Approval Exercise: 5 weeks implementation work and report
  - Needs to be approved for admission to exam. Solved in groups
  - Published 18/11, due 20/12 14:00, Milestone 6/12 23:59,
- Exam: individual graded implementation work and report

Final Exam



- Take-home exam over $4\frac{1}{2}$ days
  - Published: Monday, January 13 (9:00)
  - Due: Friday, January 17 (14:00) (in Absalon)
- Practical work on a compiler – similar to approval exercise
  - offers choice between several assignments
  - includes a report documenting the work;
  - may include small theoretical questions;
- Individual exam, do not use groups.

Reading Material

Mogensen, T. A. (2011).
Introduction to Compiler Design.
Springer, London. Previously available as [Mogensen, 2010].

Mogensen, T. A. (2000-2010).
Basics of Compiler Design.
DIKU, self-published. http://www.diku.dk/~torbenm/Basics/.

Patterson, D. A. and Hennessy, J. L. (1998).
Computer Organization & Design, the Hardware/Software Interface.
Morgan Kaufmann. Appendix A freely available at
http://www.cs.wisc.edu/~larus/HP_AppA.pdf.

Berthold, J. (2013).
MIPS, Register Allocation and MARS simulator.
Based on an earlier Danish version by Torben Mogensen. Available on Absalon.

Moscow ML (2013).
Moscow ML – a light-weight implementation of Standard ML (SML).
http://mosml.org. New release 2.10 in 2013 with substantial improvements.

Mars (2002-2013).
MARS, a MIPS Assembler and Runtime Simulator, version 4.4.
http://courses.missouristate.edu/kenvollmar/mars/.

Introduction

From Source Code to Machine Code. . .

Example Code (pseudo SML)

```
val result = let val x = 10 :: 020 :: 0x30 :: []
             in List.map (fn x => x div 2) x
             end
```

Please discuss and collect ideas about the following:

- What is the intention of this code (what does it do)?
- Write down a (roughly) equivalent code snippet in a low-level language like C or assembler.
- List the tasks necessary to translate code into machine code.

Possible "Translations"

```
val result = let val x = 10 :: 020 :: 0x30 :: []
             in List.map (fn x => x div 2) x
             end
```

A variant in C

- Initialise array
- in a loop: divide by 2, store in old x

```
int[3] x = {10, 024, 0x30 };
i = 0;
while (i < 3) {
    x[i] = x[i] / 2;
    i++;
}
```

A variant in pseudo MIPS assembler (only the loop)

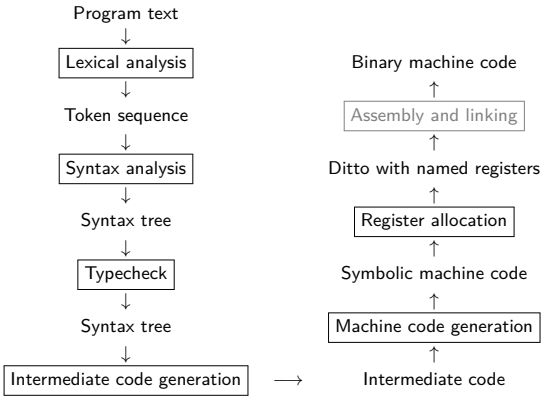
- registers instead of variables
- explicit load and store operations
- shift instead of division

```
        load_imm $2, 0          # $2: counter
        load_addr $3, x_addr    # $3: address in x
loop:   load_word $4, 0($3)     # load from array
        shift_right $4, $4, 1   # divide by 2
        store_word $4, 0($3)    # store to array
        add_imm $3, 4           # next address (4 bytes)
        add_imm $2, 1           # and next index
        load_imm $5, 3          # size 3, compare
        branch_ne $2, $5, loop
```

Tasks and Phases of Compilation

- Front End:
  - What is the input program?
  - Is the input a correct program?
  - What information can be derived and is needed to generate machine code?
- Intermediate code (machine-independent) in the middle
  - How are the data represented in memory?
- Back End:
  - What machine instruction has the desired effect (on the data)?
  - What operations are unnecessary, which order is efficient?

Compilation Phases



Lexical and Syntactic Analysis

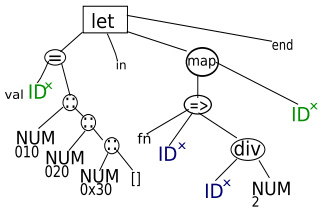
- Lexical: related to single words
  - Conventions for numbers and identifiers
  - Special (key)words
- Syntactic: concerning structure
  - Sequence of tokens assembled to syntax tree structure
  - For example: Function declarations start by `fun...`

```
val result
   = let val x = 10 :: 020 :: 0x30 :: []
     in List.map (fn x => x div 2) x
     end
```

```
valresult=letvalx=10::
020::0x30::[]inList.map
(fnx=>xdiv2)xend
```


An Abstract Syntax Tree

- Program structure as a tree
- Comments and "decoration" removed
- Additional information is added: Data types, code usage, simplifications, ...



- Central data structure in the compilation process
- All analysis and many transformations work on abstract syntax

Synthesis Phases: Code Generation

- Data structures translated to simple address arithmetics.
- Control structures and procedures translated to jumps.
- Often: Intermediate code generated, translated again
  - Easier to translate in several steps
  - Can share intermediate code among several platforms (intermediate code belongs to the front end).
- Code for an abstract machine:
  - Register machine: Arbitrary amount of abstract registers
  - Stack machine: Infinite stack of memory (no registers)
- Optimised and then transformed to real machine code

Phase Overview