Compiler Techniques

Lecture 1: Introduction

Christophe Dubach

Winter 2022

Timestamp: 2022/01/03 14:39:22

Disclaimer

Lectures are recorded live and will be posted unedited on *mycourses* on the same day.

It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the book, the course webpage or ask on Piazza for clarifications.

Christophe Dubach — brief CV

- 2005: MSc
- 2009: PhD, *Using machine-learning to efficiently explore the architecture/compiler co-design space*
- 2012: Lecturer (Assistant Professor)
- 2017: Reader (Associate Professor)
- 2010: Visiting Scientist
  LiquidMetal: a language, compiler, and runtime for high level synthesis of reconfigurable hardware
- 2020: Associate Professor (ECE/CS)
  - `ECSE 324` : Computer Organization
  - `COMP 520` : Compiler Design

Course outline

Course outline

Basics

Compiler Design

This course is an introduction to the full pipeline of modern compilers

- it covers all aspects of the compiler pipeline for modern languages (C, Java, Python, etc.);
- touches on advanced topics related to optimization
- will present how realworld compilers are built

By the end of this class you will have a working knowledge of compilers that allows you to:

- Produce fully functional compilers for general-purpose languages targetting real machine assembly.

Syllabus

- Overview
- Core topics
  - Scanning
  - Parsing
  - Abstract Syntax Tree
  - Semantic analysis
  - Code generation for machine assembly
  - SSA form & Dataflow analysis
  - Register allocation
- Advanced topics (if time allows)
  - Instruction selection
  - Instruction scheduling
  - Compiling object oriented languages
  - Garbage collection
  - Realworld IR (*e.g.* LLVM, WebAssembly)

Class information

4 credit courses

Schedule:

- Lectures: WF 8:30–10:00
- Prof. office hours: W 16:00–17:00

Lecture

- Live lecture on Zoom (until further notice)

Office hours

- Open Zoom session (one at a time), link on course website

Prerequisites:

- COMP 273, COMP 302

Teaching Team

Prof.:

- Christophe Dubach (christophe.dubach@mcgill.ca)

TAs:

Shakiba Bolbolian Kha

MSc student
Compilation & high-level synthesis for hardware accelerators

Jonathan Van der Cruysse

PhD student
Compilation for specialized high-performance libraries & accelerators

Course outline

Assessments

Evaluation

Coursework only, no exam

- Expect to spend a lot of hours on the coursework ($\sim$100+)
- A lot of programming!

Assessments:

- Five deadlines for the project scattered throughout the term
- One ($\sim$15min) demo just before exam period:
  purpose is to check you did the work yourself

All deadlines will be strictly enforced.

Demo

if no demo or cannot answer our questions
$\Rightarrow$ will be reported to faculty for suspected academic misconduct.

Academic Integrity

McGill University values academic integrity. Therefore, all students must understand the meaning and consequences of cheating, plagiarism and other academic offences under the Code of Student Conduct and Disciplinary Procedures. (approved by Senate on 29 January 2003)

Cheating

Cheating is a serious offense and all suspected cases will be reported to the faculty!

For this course:

- Never share your code
- Never use someone else code or any third party code
- Always write your own code
- You should not use previous years' solutions

On the other hand, you are allowed to:

- Share and discuss ideas
- Help someone else debug their code if they are stuck
  (you can point out at their errors, but never write code for them!)
- If you obtain any help, always write the name(s) of your sources and explicitly state how your submission
  (via a README file for instance)

Submission language

In accord with McGill University's Charter of Students' Rights, students in this course have the right to submit in English or in French any written work that is to be graded.
(approved by Senate on 21 January 2009)

Course outline

Course material

Course Material

Course website

- https://www.cs.mcgill.ca/~cs520/2022/
- Contains schedule, deadlines, slides

Slides:

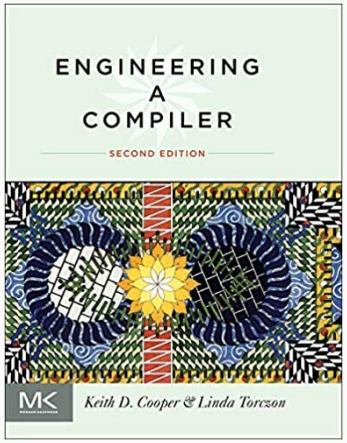
- Available on course website before each lecture

Recordings:

- Available on myCourses "shortly" after each lecture

Textbook (not strictly required):

- Keith Cooper & Linda Torczon: Engineering a Compiler, Elsevier, 2004.



Discussion forum:

- ED

Action

Create an account and subscribe to the course on ed.
ttps://edstem.org/us/join/p6wggP

Gitlab:

- https://gitlab.cs.mcgill.ca/cdubach/comp520-coursework-w2022
- contains project description and initial code template (will be released next week)

Action

Check that you can access gitlab. More information on the course webpage on how to obtain a CS account if needed.

Course outline

Project

Project

Write a compiler from scratch

- Written in Java for a subset of C
  includes pointers, recursion, structs, memory allocation, ...
- Backend will target a real RISC assembly (MIPS)
- Generated code executable in a simulator

Input: C code

```
int fact(int n) {
  if (n<1)
    return 1;
  return n*fact(n-1);
}
```

Output: MIPS assembly

```
fact:   li      $2,1    # 0x1
        blez    $4,$L9
        mult    $2,$4
$L7:    addiu   $4,$4,-1
        mflo    $2
        mult    $2,$4
        bne     $4,$0,$L7
$L9:    j       $31
```

Five coursework deadlines (exact dates on course website):

- week 5 (20%) Parser
- week 8 (20%) Abstract Syntax Tree construction
  + Semantic Analyser
- week 11 (20%) Code generator with naïve register allocator
- week 14 (20%) "Proper" register allocator
- week 17 (20%) Advanced topic (TBA)

One demo

- week 15, last two days of term
- passing the demo is mandatory to pass the course

Marking is done by pulling the content of your repository $\Rightarrow$
Whatever is there at the date/time of the deadline is what is marked.

Coursework is challenging

Coursework requires good programming skills

- Java + basic knowledge of C and assembly
- E.g. exceptions, recursion, inheritance, ...

Assumes basic knowledge of Unix command line (can be learnt on the fly to some extend)

- cp, mv, ls, ...

Git will be used for the coursework (will be learnt on your own)

Coursework marking and scoreboard

- Automated system to evaluate coursework
- Mark is a function of how many programs compile successfully
- Nightly build of your code with scoreboard updated daily

Provided as best-effort service, do not rely on it!!!

Auto-marking & scoreboard will start in 1–2 weeks from now.

Coursework will be rewarding

You will understand what happens when you type: `$ gcc hello.c`

But also:

- Will deepened your understanding of computing systems (from language to hardware)
- Will improve your programming skills

A few last words on the course

- Extensive use of projected material
  - Attendance and interaction encouraged
  - Feedback also welcome
- Reading book is optional
  (course is self-contain, book is more theoretical)
- Not a programming course!
- Start the practical early
- Help should be sought on Ed in the first instance
  😠 no email! 😠 (unless for personal matter)

What is a compiler?

Compilers

What is a compiler?

A program that *translates* an executable program in one language into an executable program in another language.
The compiler might improve the program, in some way.

What is an interpreter?

A program that directly *execute* an executable program, producing the results of executing that program

Examples:

- C is typically compiled
- R is typically interpreted
- Java is compiled to bytecode, then interpreted or compiled (just-in-time) within a Java Virtual Machine (JVM)

A Broader View

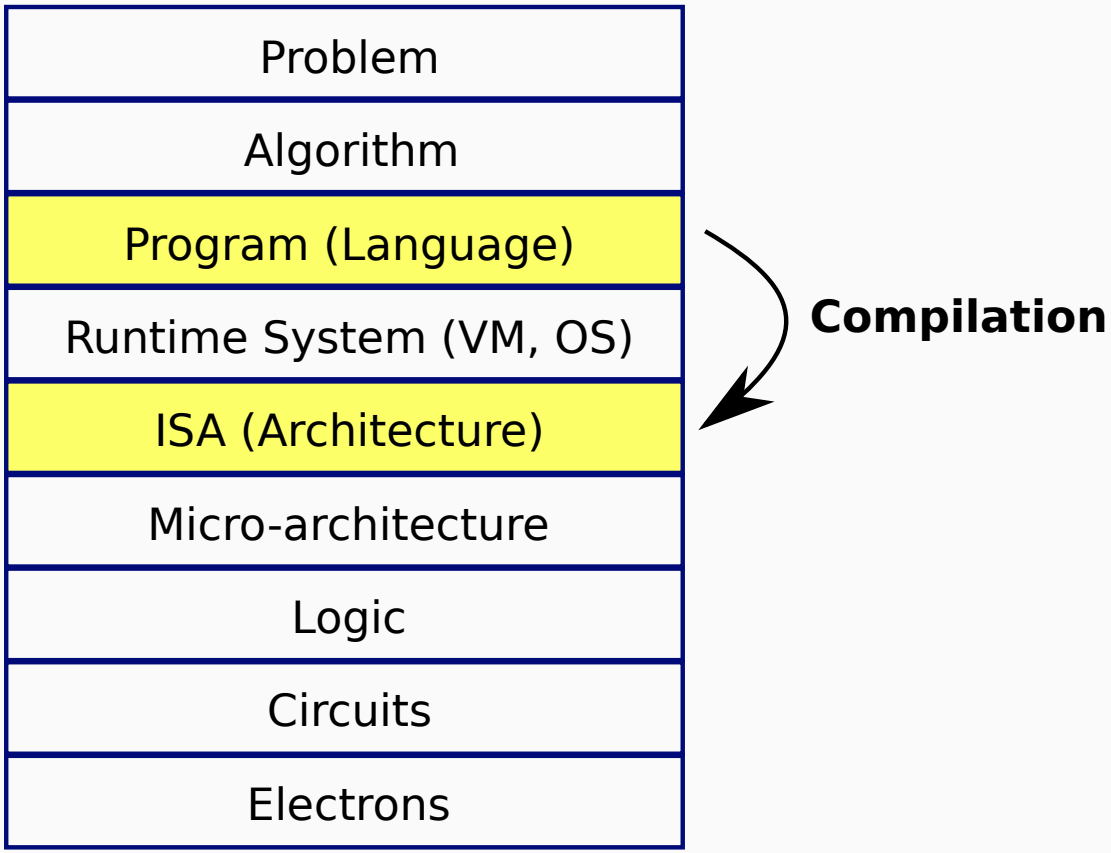
Compiler technology

- Goals: improved performance and language usability
  Making it practical to use the full power of the language
- Trade-off: preprocessing time versus execution time (or space)
- Performance of both compiler and application must be acceptable to the end user

Examples:

- Macro expansion / Preprocessing
- Database query optimisation
- Javascript just-in-time compilation
- Emulation: *e.g.* Apple's Intel transition from PowerPC (2006)

System Stack



Why studying compilers?

New programming languages keeps emerging

No less than 30 new general purpose languages designed just between 2010—2020

- Rust, Dart, Kotlin, TypeScript, Julia, Swift, ...

Plenty of DSLs (Domain Specific Languages):

- Latex, SQL, SVG, HTML, DOT, MarkDown, XPath

Perhaps one day you will create your own?

Why study compilation?

- Compilers are important system software components:
  they are intimately interconnected with architecture, systems, programming methodology, and language design
- Compilers include many applications of theory to practice:
  scanning, parsing, static analysis, instruction selection
- Many practical applications have embedded languages:
  commands, macros, formatting tags ...
- Many applications have input formats that look like languages:
  Matlab, Mathematica
- Writing a compiler exposes practical algorithmic & engineering issues:
  approximating hard problems; efficiency & scalability

Intrinsic interest

Compilers involve ideas from different parts of computer science

Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms Dynamic programming
Theory	DFA & PDA, pattern matching Fixed-point algorithms
Systems	Allocation & naming Synchronisation, locality
Architecture	Pipeline & memory hierarchy management Instruction set
Software engineering	Design pattern (visitor) Code organisation

Intrinsic merit

Compiler construction poses challenging and interesting problems:

- Compilers must do a lot but also run fast
- Compilers have primary responsibility for run-time performance
- Compilers are responsible for making it acceptable to use the full power of the programming language
- Computer architects perpetually create new challenges for the compiler by building more complex machines
- Compilers must hide that complexity from the programmer
- Success requires mastery of complex interactions

Making languages usable

It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger.

...

I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

John Backus (1978)

Next lecture

The View from 35000 Feet

- How a compiler works
- What I think is important
- What is hard and what is easy