Compiler Design

Lecture 1: Introduction

Christophe Dubach

Winter 2025

Timestamp: 2025/01/07 15:15:00

Disclaimer

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 the online forum for clarifications.

Short answer: **Prof.**

Longer answer, it's context-sensitive!

- For undergraduate classes, I prefer if students use "Prof."
- For graduate classes, I'm happy for students to use my first name.

Since COMP520 has both type of students, we will go with Prof. in the context of this class.

A Brief (Professional) History of Christophe Dubach

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

- 2005: MSc

THE UNIVERSITY OF EDINBURGH

- 2009: PhD, *Using machine-learning to efficiently explore the architecture/compiler co-design space*
- 2012: Lecturer (Assistant Professor)
- 2017: Reader (Associate Professor)

IBM

- 2010: Visiting Scientist
  LiquidMetal: a language, compiler, and runtime for high level synthesis of reconfigurable hardware

McGill

- 2020: Associate Professor (ECE/CS)
  - `ECSE-324` : Computer Organization
  - `COMP-520` : Compiler Design
  - `COMP-764/ECSE-688` :
    High-level Synthesis of Digital Systems

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
    - Compiling object oriented languages
    - Garbage collection
- Extra topics (if time allows)
    - Instruction selection
    - Instruction scheduling
    - Realworld IR (*e.g.* LLVM, WebAssembly)

Class information

4 credit courses

Schedule:

- Lectures: 3 hours per week
- Prof. office hours: 1 hour per week
- TAs office hours: 2 hours per week

Lecture

- In-person lecture

Prerequisites:

- COMP 273/ECSE 324, (COMP 302)

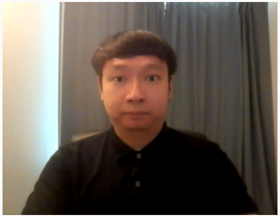
Teaching Team

Prof.:

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

TAs:

Tzung-Han Juang



PhD student
High-level Synthesis of CNN on FPGAs.

Paul Teng

PhD student
Accelerating PyTorch with FPGAs.

Aziz Zayed

MSc student
Equality saturation for multi-level IRs

Course outline

Assessments

Evaluation

Coursework only, no exam

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

Assessments:

- Five deadlines for the project scattered throughout the term
- One (∼15min) demo at end of course:
  purpose is to check you did the work yourself

All deadlines will be strictly enforced.

Demo

if no demo or cannot answer our questions
⇒ 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 or text.
- Never use someone else code or text in your solutions (even if you change it).
- Never consult project code or text that might be on the Internet.
- Always write your own code.
- Do not use automatically generated code (*e.g.* ChatGPT-generated code), unless you wrote the generator yourself and you submit it together with your code.

⚠ Even if a single line of code is plagiarized/copied, your WHOLE coursework will be deemed tainted. If caught, your will be given a zero!

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)

Marking

If you notice any marking issues with your assignment:

- you should raise the issue on ED (in a private post) immediately and
- this must be done within 7 days of receiving the grade.

The final caculated grade will be rounded to the closest integer, for instance:

- 64.4% $\Rightarrow$64%
- 64.5% $\Rightarrow$65%

Course outline

Course material

Course Material

Course website

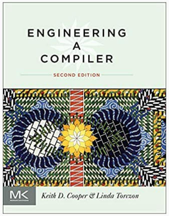
- https://www.cs.mcgill.ca/~cs520/
- Contains schedule, deadlines, slides

Slides:

- Available on course website before each lecture

Textbook (not strictly required):

- Keith Cooper & Linda Torczon: Engineering a Compiler, Elsevier, 2004.
- Older edition available freely here: https://dl.acm.org/doi/book/10.5555/2737838



Discussion forum: ED

⚠ Action

Create an account and subscribe to the course on ED
(link on the website).

Coursework on CS Gitlab:

- link to git repo wil be available from the website
- contains project description and initial code template
  (will be released next week)

⚠ Action

- Create a CS gitlab account. 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 parts, worth 20% each, due ∼every three weeks:

1. Parser
2. Abstract Syntax Tree + Semantic Analyser
3. Code generator
4. Register allocator
5. Support for object-oriented features (or maybe alternative project with LLVM, to be confirmed later)

⚠ Each part builds on top of the previous one, and no intermediate solution will be provided.

To pass this course you will have to build the whole compiler!

Deadlines:

- can be found on the course webpage
- are strictly enforced

No extension policy

- You must manage your time well (start early!);
- Only exception: if you have a sitation beyond your control that prevents your from working:
    - accident
    - death of a close family member
- on the other hand, the following would not be considered for an exception:
    - relationship broke down
    - job interview
    - part-time job

Marking

- done by pulling the content of your repository at the deadline
- mark only depends on number of passed tests:
  we reward results, not efforts

Demo

- Last few days of term (during exam session)
- passing the demo is mandatory to pass the course

Coursework is challenging

Coursework requires good programming skills:

- good knowledge of Java:
  e.g. exceptions, recursion, inheritance, ...
- basic knowledge of C:
  e.g. pointers, struct, ...
- basic knowledge of assembly:
  e.g. registers, branches, addresses, ...

Assumes basic knowledge of Unix command line and git

- cp, mv, ls, ...
- git commit, git merge, git checkout, ...

Using Git on the command line will be necessary for the coursework

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 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

Quotes from past course evaluations

Anonymous, Winter 2024 The hours needed to complete the work for this course it approx double any other 500 level I took.

Anonymous, Winter 2023 This course involved a lot of work, but that was made clear in the beginning.

Anonymous, Winter 2023 [...] i liked the emphasis on starting early as a chronic procrastinator and this is something that should be continued to be emphasised (maybe more) because the coursework takes a lot more time than other course i have completed at McGill.

Anonymous, Winter 2023 Even though I got the metaphorical beating of my life with the project, it was a great way to learn

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)
- Do make use of office hours! Especially if you are struggling.

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

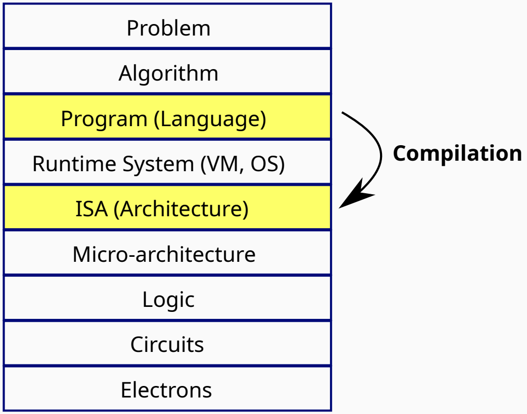
Compiler technology

- 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, Makefiles, …

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

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 …

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

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.

Speed:

- Compilers must do a lot but also run fast
- Compilers have primary responsibility for run-time performance

Intrinsic merit

Compiler construction poses challenging and interesting problems.

Speed:

- Compilers must do a lot but also run fast
- Compilers have primary responsibility for run-time performance

High-level language features:

- Compilers are responsible for making it acceptable to use the full power of the programming language

Intrinsic merit

Compiler construction poses challenging and interesting problems.

Speed:

- Compilers must do a lot but also run fast
- Compilers have primary responsibility for run-time performance

High-level language features:

- Compilers are responsible for making it acceptable to use the full power of the programming language

Computer architecture complexity:

- Computer architects perpetually create new challenges for the compiler by building more complex machines
- Compilers must hide that complexity from the programmer

Intrinsic merit

Compiler construction poses challenging and interesting problems.

Speed:

- Compilers must do a lot but also run fast
- Compilers have primary responsibility for run-time performance

High-level language features:

- Compilers are responsible for making it acceptable to use the full power of the programming language

Computer architecture complexity:

- Computer architects perpetually create new challenges for the compiler by building more complex machines
- Compilers must hide that complexity from the programmer

Interaction between software & hardware:

- Success requires mastery of complex interactions

The first compiler: FORTRAN

Before FORTAN, programs were written in assembly.

First compiler

- Implemented in the 1950s.
- Had to overcome deep skepticisim; and
- Paid less attention to language design and more on performance of generated code.

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)

Final words

This is (most likely) going to be the largest piece of software you will write while at McGill.

This course will:

- strengthen your programming skills
- help you understand more deeply programming language features
- help you achieve a much better understanding of the computing system stack
- add a very nice line on your CV: compilers are considered the most challenging software to write by many!

Next lecture

The View from 35000 Feet

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