Compiling Techniques

Lecture 1: Introduction

Christophe Dubach

19 September 2017

Table of contents

Essential Facts

- Lecturer: Christophe Dubach (christophe.dubach@ed.ac.uk)
- Office hours: Wednesdays 10am-12pm
- Textbook (not strictly required):
  - Keith Cooper & Linda Torczon: Engineering a Compiler Elsevier, 2004
  - Textbook can be reused in UG4 Compiler Optimisation
- Course website:
  http://www.inf.ed.ac.uk/teaching/courses/ct/17-18
- Discussion forum:
  http://piazza.com/ed.ac.uk/fall2017/ct

Action

Create an account and subscribe to the course on piazza.

Essential Facts

- Course is 20 credits
- Evaluation: no exam, coursework only
- Expect to spend a lot of hours on the coursework ($\sim$200+)
- A lot of programming! (mainly Java but also a bit of C++)
- 3 hours of lectures per week + 2 hours labs

Coursework (1)

There will be two distinct coursework assignments.

1. Write a full compiler from scratch (70% of total mark)
   - Will be written in Java
   - For a subset of C
     (includes pointers, recursion, structs, memory allocation, ...)
   - Backend will target a real RISC assembly
   - Generated code executable in a simulator
   - Three deadlines:
     - week 4 (20%) Parser
     - week 6 (20%) Abstract Syntax Tree (AST) + Semantic Analyser
     - week 9 (30%) Code generator

Coursework (2)

2. Write a new compiler pass in an existing compiler (30% of total mark)
   - LLVM-based
   - one deadline: week 1 semester 2

- Will be taught by Aaron Smith (Microsoft Research)
- Very practical knowledge to anyone interested in compiler industry

Coursework is challenging

Coursework requires good programming skills

- Java for 1st assignment + basic knowledge of C
- C/C++ for 2nd assignment
- E.g. exceptions, recursion, Java collections classes, inheritance, ...

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

- cp, mv, ls, ...
- ant, makefile

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

Coursework marking and labs

- Automated system to evaluate coursework
  - Mark is a function of how many programs compile successfully
  - Nightly build of your code with scoreboard
- Will rely on git/bitbucket

Action

- Create an account on http://bitbucket.org
- send account id via Google form: https://docs.google.com/forms/d/1z2EthflazoU2bvfnJlrCWB_-AqB4ZxIgsJW-8SWiXyM

- mandatory demo; if no demo $\rightarrow$ mark $= 0$
- Labs here to help with coursework in one session of 2 hours
  - Friday 14:10 - 16:00, Appleton Tower, 6.06

Labs start this week and end on week 11.

Coursework is also 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
- Will learn about using revision control system (git)

Class-taking Technique

- Extensive use of projected material
  - Attendance and interaction encouraged
  - Feedback also welcome
- Reading book is optional
  (course is self-content, book is more theoretical)
- Not a programming course!
- Start the practical early
- Help should be sought on Piazza in the first instance

Syllabus

- Overview
- Scanning
- Parsing
- Abstract Syntax Tree
- Semantic analysis
- Code generation
  - Virtual machines (Java) Bytecode
  - Real machines assembly
- LLVM compiler infrastructure (Aaron Smith from MSR)
- Advanced topics
  - Instruction selection
  - register allocation

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 = Off-line processing

- 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)
- Rule: 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)

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

Compiler construction involves ideas from many 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