Compilers

CS143
Tuesday/Thursday 9:45–11:15

Instructor: Fredrik Kjolstad
The slides in this course are designed by Alex Aiken,
with modifications by Fredrik Kjolstad.
Staff

- Instructor
  - Fredrik Kjolstad

- TAs
  - Scott Kovach
  - Wonyeol Lee
  - Nikhil Raghuraman
  - Toby Bell
  - Timothy Gu
Administrivia

- Syllabus is on-line
  - cs143.stanford.edu
  - Assignment dates will not change
  - Midterm (Thursday April 28)
  - Final

- Office hours
  - 22 office hours spread throughout the week
  - Some zoom office hours where SCPD students get preference
  - My office hours this week: Thursday 4-5pm (zoom) and Friday 10-11am (Gates 486)
  - Office hours starting next week to be announced

- Communication
  - Use discussion forum, email, zoom, office hours
Webpages and servers

• Course webpage at cs143.stanford.edu
  – Syllabus, lecture slides, handouts, assignments, and policies

• Canvas at canvas.stanford.edu
  – Lecture recordings available under the Panopto Course Videos tab

• Ed Discussion at https://edstem.org/us/courses/21322/discussion/
  – This is where you should ask most questions
  – Also accessible from Canvas

• Gradescope at gradescope.com
  – This is where you will hand in written assignments

• Computing Resources at myth.stanford.edu
  – We will use myth for the programming assignments
  – Class folder: /afs/ir/class/cs143/
Text

• The Purple Dragon Book

• Aho, Lam, Sethi & Ullman

• Not required
  – But a useful reference
Course Structure

• Course has theoretical and practical aspects
• Need both in programming languages!
• Written assignments + exams = theory
• Programming assignments = practice
Course Goal

- Open the lid of compilers and see inside
  - Understand what they do
  - Understand how they work
  - Understand how to build them

- Correctness over performance
  - Correctness is essential in compilers
  - They must produce correct code
  - CS143 is more like CS103+CS110 than CS107
  - Other classes focus on performance (CS149, CS243)
Academic Honesty

• Don’t use work from uncited sources

• We may use plagiarism detection software
  – many cases in past offerings
The Course Project

• You will write your own compiler!

• One big project

• ... in 4 parts

• Start early!

Diagram:

- PA1: lexer
- PA2: parser
- PA3: type checker
- PA4: code generation
How are Languages Implemented?

• Two major strategies:
  – Interpreters run your program
  – Compilers translate your program
Language Implementations

• Compilers dominate low-level languages
  – C, C++, Go, Rust

• Interpreters dominate high-level languages
  – Python, Ruby

• Some language implementations provide both
  – Java, Javascript, WebAssembly
  – Interpreter + Just in Time (JIT) compiler
History of High-Level Languages

• 1954: IBM develops the 704

• Problem
  – Software costs exceeded hardware costs!

• All programming done in assembly
The Solution

• Enter “Speedcoding”

• An interpreter

• Ran 10-20 times slower than hand-written assembly
FORTRAN I

- Enter John Backus

- Idea
  - Translate high-level code to assembly
  - Many thought this impossible
  - Had already failed in other projects
FORTRAN I (Cont.)

- 1954-7
  - FORTRAN I project

- 1958
  - >50% of all software is in FORTRAN

- Development time halved

- Performance close to hand-written assembly!
FORTRAN I

• The first compiler
  – Huge impact on computer science

• Led to an enormous body of theoretical and practical work

• Modern compilers preserve the outlines of FORTRAN I

• Can you name a modern compiler?
The Structure of a Compiler

1. Lexical Analysis
   — identify words
2. Parsing
   — identify sentences
3. Semantic Analysis
   — analyse sentences
4. Optimization
   — editing
5. Code Generation
   — translation

Can be understood by analogy to how humans comprehend English.
Lexical Analysis

• First step: recognize words.
  – Smallest unit above letters

This is a sentence.
More Lexical Analysis

• Lexical analysis is not trivial. Consider:
  ist his ase nte nce
And More Lexical Analysis

• Lexical analyzer divides program text into “words” or “tokens”

\[
\text{If } x == y \text{ then } z = 1; \text{ else } z = 2;
\]

• Units:
• Once words are understood, the next step is to understand sentence structure

• Parsing = Diagramming Sentences
  – The diagram is a tree
Diagramming a Sentence

This line is a longer sentence

article noun verb article adjective noun

subject object sentence
Parsing Programs

- Parsing program expressions is the same
- Consider:
  \[
  \text{If } x == y \text{ then } z = 1; \text{ else } z = 2;
  \]
- Diagrammed:
Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is too hard for compilers

- Compilers perform limited semantic analysis to catch inconsistencies
Semantic Analysis in English

• Example:
  Jack said Jerry left his assignment at home.
  What does “his” refer to? Jack or Jerry?

• Even worse:
  Jack said Jack left his assignment at home?
  How many Jacks are there?
  Which one left the assignment?
Semantic Analysis in Programming

• Programming languages define strict rules to avoid such ambiguities

• This C++ code prints “4”; the inner definition is used

```cpp
int Jack = 3;
{
    int Jack = 4;
    cout << Jack;
}
```
More Semantic Analysis

• Compilers perform many semantic checks besides variable bindings

• Example:

  Jack left her homework at home.

• Possible type mismatch between her and Jack
  – If Jack is male
Optimization

• Akin to editing
  – Minimize reading time
  – Minimize items the reader must keep in short-term memory

• Automatically modify programs so that they
  – Run faster
  – Use less memory
  – In general, to use or conserve some resource

• The project has no optimization component
  – CS243: Program Analysis and Optimization
Optimization Example

\[ X = Y \times 0 \] is the same as \[ X = 0 \]

(the \(*\) operator is annihilated by zero)

Is this optimization legal?
Code Generation

• Typically produces assembly code

• Generally a translation into another language
  – Analogous to human translation
Intermediate Representations

- Many compilers perform translations between successive intermediate languages
  - All but first and last are intermediate representations (IR) internal to the compiler

- IRs are generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly
Intermediate Representations (Cont.)

• IRs are useful because lower levels expose features hidden by higher levels
  – registers
  – memory layout
  – raw pointers
  – etc.

• But lower levels obscure high-level meaning
  – Classes
  – Higher-order functions
  – Even loops…
Issues

• Compiling is almost this simple, but there are many pitfalls

• Example: How to handle erroneous programs?

• Language design has big impact on compiler
  – Determines what is easy and hard to compile
  – Course theme: many trade-offs in language design
Compilers Today

• The overall structure of almost every compiler adheres to our outline

• The proportions have changed since FORTRAN
  – Early: lexing and parsing most complex/expensive
  – Today: optimization dominates all other phases, lexing and parsing are well understood and cheap

• Compilers are now also found inside libraries