The slides in this course are designed by Prof. Alex Aiken, with modifications by Fredrik Kjolstad.
Administrivia

• Syllabus is on-line
  - cs143.stanford.edu
  - Assignment dates will not change
  - Midterm
    • Thursday April 29, via Gradescope
  - Final
    • Thursday June 3, via Gradescope

• Office hours
  - 20 office hours spread throughout the week
  - On Zoom scheduled through Canvas

• Communication
  - Use discussion forum, email, zoom, office hours
• Course webpage at cs143.stanford.edu
  - Syllabus, lecture slides, handouts, assignments, and policies

• Canvas at canvas.stanford.edu
  - Zoom links to lectures and office hours (see Zoom tab)
  - Lecture recordings available under the Zoom tab -> Cloud Recordings

• Piazza at piazza.com/stanford
  - This is where most questions should be asked
  - Live Q&A during lectures

• Gradescope at gradescope.com
  - This is where you will hand in written assignments and midterm/final

• Computing Resources
  - We will use myth.stanford.edu for the programming assignments
Staff

• Instructor
  - Fredrik Kjolstad

• TAs
  - Caleb Donovick
  - Nikhil Athreya
  - Sicheng Zeng
  - Yinghao Sun
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 = theory

• Programming assignments = practice
Academic Honesty

• Don’t use work from uncited sources

• We use plagiarism detection software
  - many cases in past offerings

PLAGIARISM
The Course Project

- You will write your own compiler!
- One big project
- ... in 4 parts
- Start early! (Start early)
How are Languages Implemented?

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

• Batch compilation systems dominate “low level” languages
  – C, C++, Go, Rust

• “Higher level” languages are often interpreted
  – Python, Ruby

• Some (e.g., Java, Javascript) provide both
  – Interpreter + Just in Time (JIT) compiler
History of High-Level Languages

• 1954: IBM develops the 704
  - Successor to the 701

• 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 is 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
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

The first 3, at least, 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”

  If x == y then z = 1; else z = 2;

• Units:
Parsing

• 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

• No strong counterpart in English, but akin to editing

• 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 * 0 is the same as X = 0

(the * operator is annihilated by zero)
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