Compilers

CS143

Lecture 1

Instructor: Fredrik Kjolstad

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

Staff

- Instructor
  - Fredrik Kjolstad
- TAs
  - Tejas Narayanan
  - Colin Schultz
  - James Dong
  - Olivia Hsu
  - Daniel Rebelsky
  - Trevor Gale

Administrivia

- Syllabus is on-line
  - cs143.stanford.edu
  - Assignment dates
  - Midterm (Thursday May 2)
  - Final

- Office hours
  - Office hours spread throughout the week (some on zoom)
  - My office hours: Thursday 5-6pm (zoom) and Friday 9-10am (Gates 486)
  - Office hours starting next week to be announced

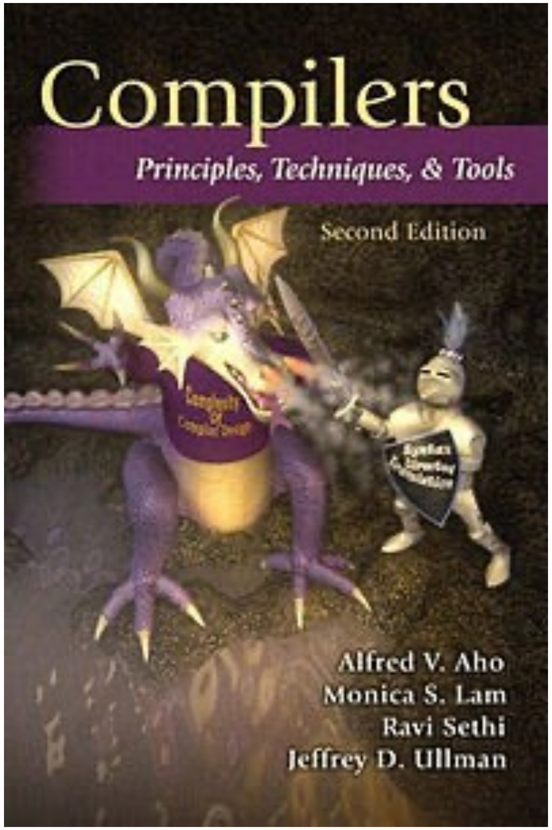
- Communication
  - Use ed, email, zoom, office hours

Webpages and servers

- Course webpage at cs143.stanford.edu
  - Syllabus, lecture slides, handouts, assignments, and policies
- Canvas at https://canvas.stanford.edu/courses/190387
  - Lecture recordings available under the Panopto Course Videos tab
- Ed Discussion at https://edstem.org/us/courses/57833/discussion/
  - This is where you should ask most questions
  - Also accessible from Canvas
- Gradescope at https://www.gradescope.com/courses/761000
  - 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

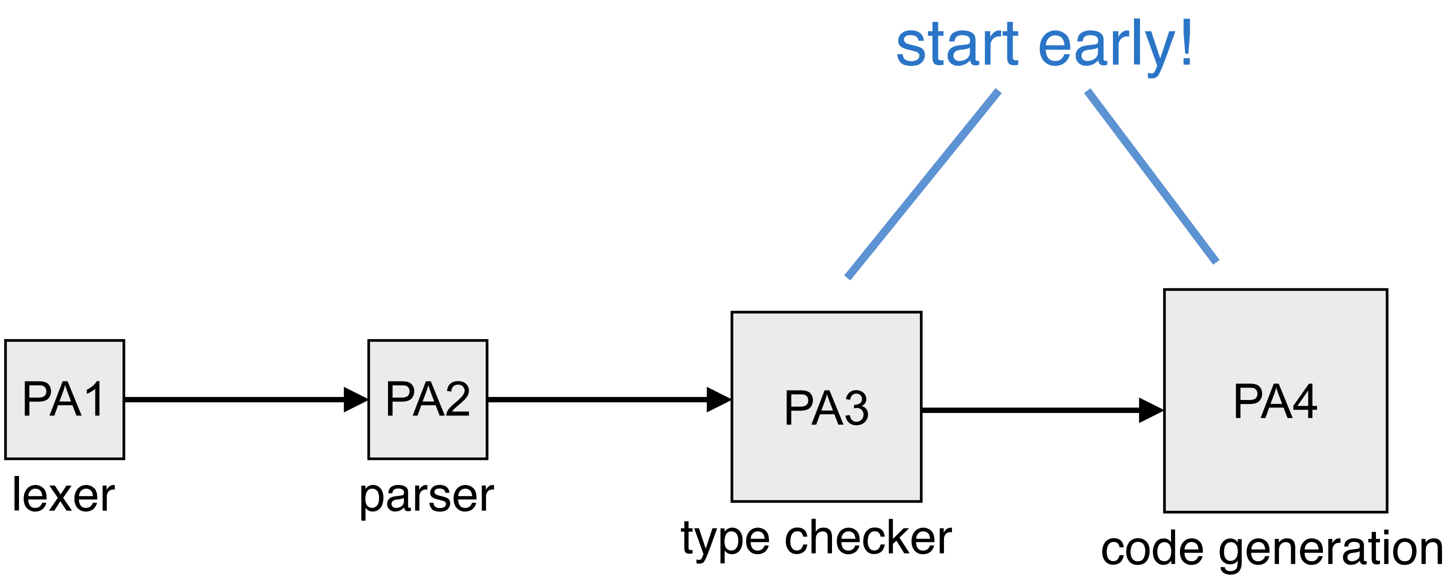
Academic Honesty

- 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



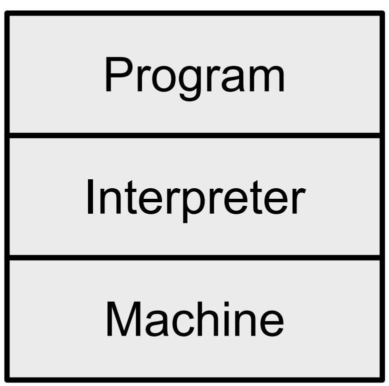
Course Goal

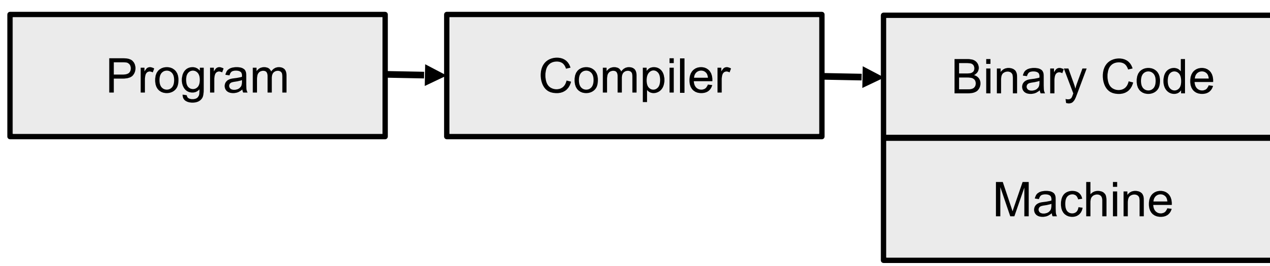
- Open the lid of compilers and see inside
  - Understand what they do
  - Understand how they work
  - Understand how to build them

- **Correctness** over performance
  - Correctness is essential in compilers
  - They must produce correct code
  - Enormous consequences if they do not
  - Other classes focus on performance (CS149, CS243)

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, JavaScript
- Many 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

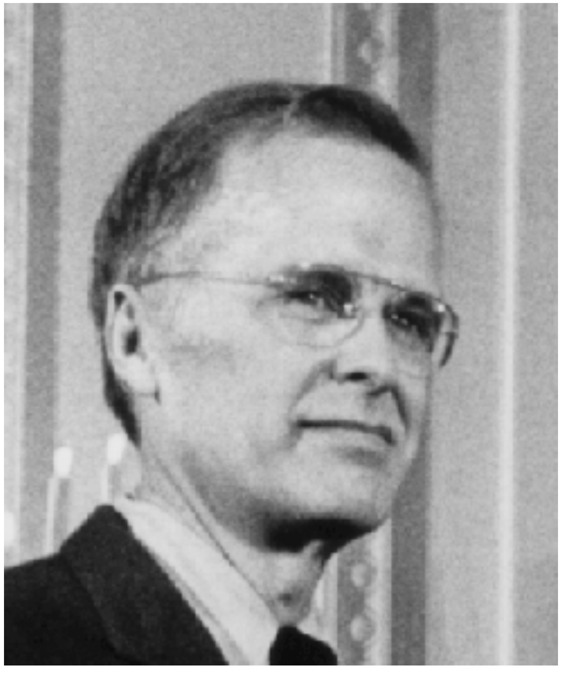
The Solution

- Enter “Speedcoding”
- An interpreter
- Ran 10-20 times slower than hand-written assembly

FORTRAN I

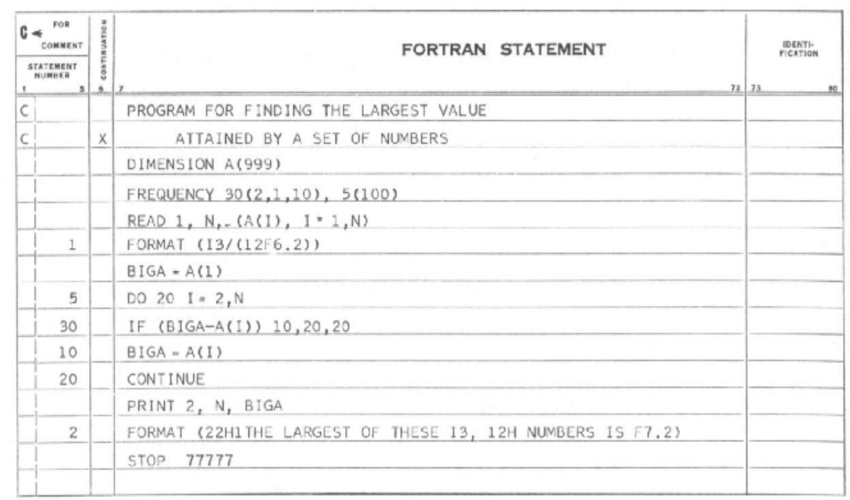
- Enter John Backus

- 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!



C ← FOR COMMENT | STATEMENT NUMBER | CONTINUATION | FORTRAN STATEMENT | IDENTIFICATION

```
C       PROGRAM FOR FINDING THE LARGEST VALUE
C     X      ATTAINED BY A SET OF NUMBERS
        DIMENSION A(999)
        FREQUENCY 30(2,1,10), 5(100)
        READ 1, N, (A(I), I = 1,N)
    1   FORMAT (I3/(12F6.2))
        BIGA = A(1)
    5   DO 20 I = 2,N
   30   IF (BIGA-A(I)) 10,20,20
   10   BIGA = A(I)
   20   CONTINUE
        PRINT 2, N, BIGA
    2   FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)
        STOP 77777
```

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.

- Suppose we scramble the whitespaces:

  ist his ase nte nce

- Suppose we replace whitespace with z:

  iszthiszazsentence

And More Lexical Analysis

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

```
if x == y then z = 1; else z = 2;
```

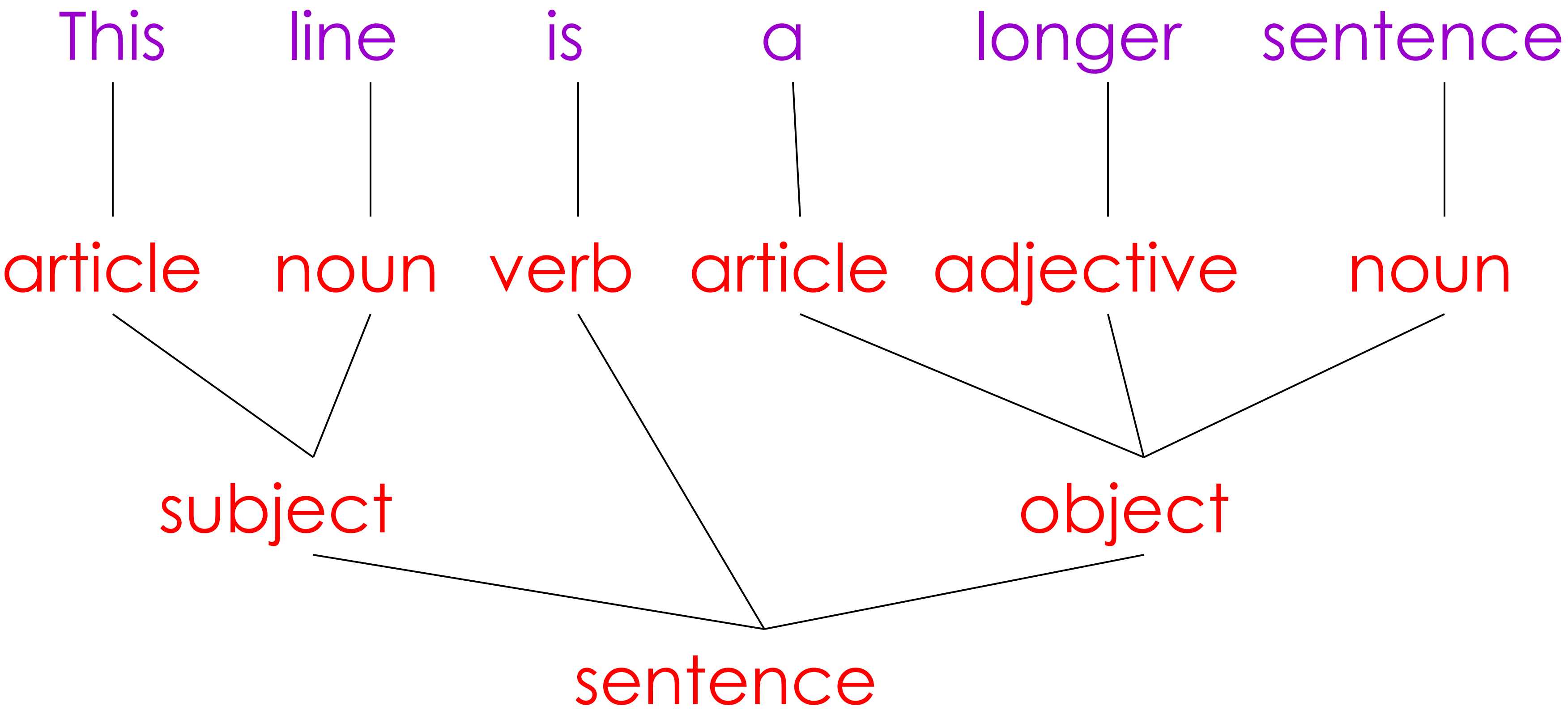
- Units:

Parsing

- Once words are understood, the next step is to understand sentence structure

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence

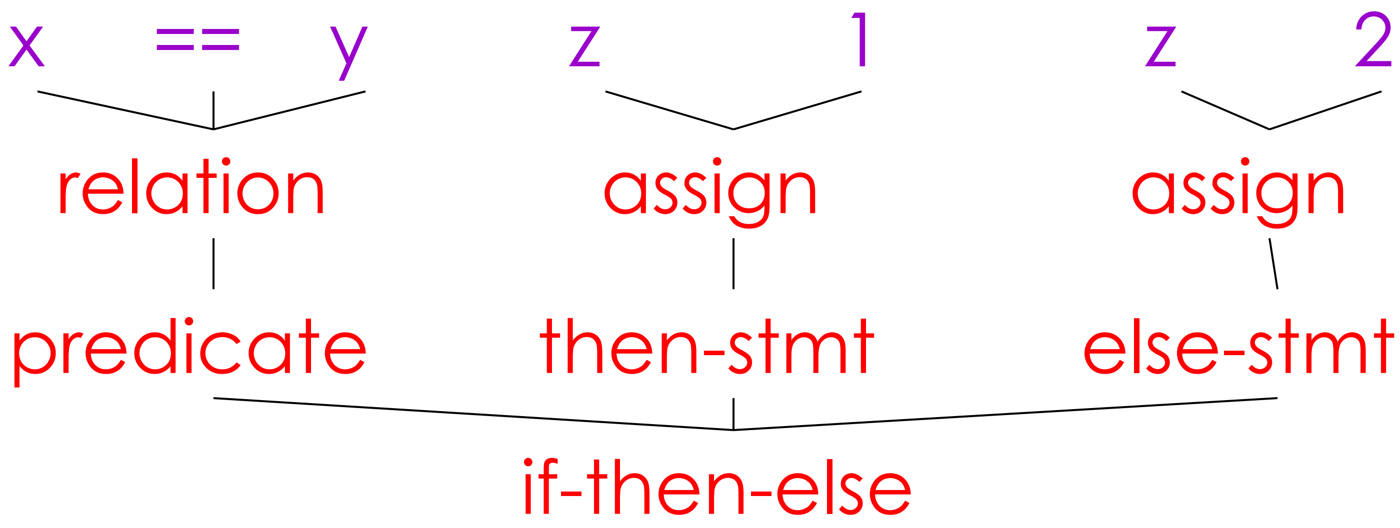


Parsing Programs

- Parsing program expressions is the same
- Consider:

if x == y then z = 1 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:

  Jill said Jill left her assignment at home?

  How many Jills 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

```
{
    int i = 3;
    {
        int i = 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 conserve some resource
- The project has little optimization.
  - See CS243 Program Analysis and Optimization

Optimization Example

x = y * 0   is the same as x = 0

(the * operator is annihilated by zero)

Is this optimization legal?

Code Generation

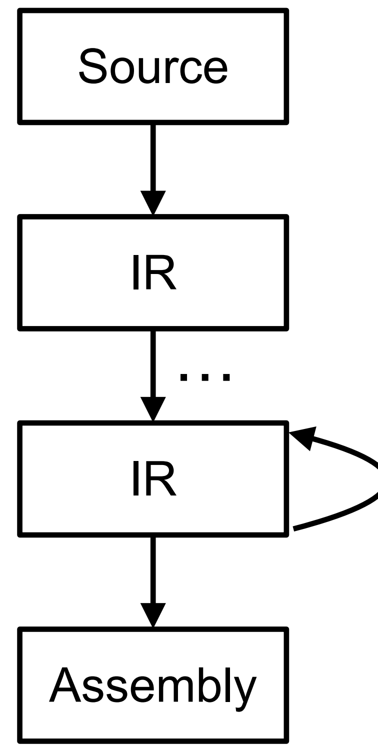
- Typically produces assembly code

- Generally a translation into another language
  - Analogous to human translation

Intermediate Representations (IR)

- Compilers typically 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 (IR) (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 a big impact on the 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:
  - XLA, TVM, Halide, DBMS, …