Welcome!

• I’m glad you’re here!
• This is a rough time for everyone. If there’s anything I can do to make things more survivable, please let me know.
• Instructor: William Mansky (he/him)
• Office hours Tuesday 1-2, Friday 10:30-11:30, and by appointment, on Blackboard Collaborate
• Office hours are great for homework help, or just to say hi!

• This class: CS 473/MCS 411, Compiler Design
• Prerequisites: CS 301 (languages and automata), CS 251 (trees), CS 261 (C and assembly programming)
• Web site: https://www.cs.uic.edu/~mansky/teaching/cs473/sp21/
• Anonymous in-class questions: https://pollev.com/wmansky771
• Discussion board: https://piazza.com/uic/spring2021/cs473
• Lectures and recordings: Blackboard, via Collaborate
• Assignment submission: Gradescope (course code 74DX7V)
What is a Compiler?

- Computers don’t actually understand programming languages!
What is a Compiler?

• CPUs don’t actually understand programming languages!
• A compiler is a program that translates from one programming language to another.
• Typically: high-level source code to low-level machine code
What is a Compiler?

- CPUs don’t actually understand programming languages!
- A compiler is a program that translates from one programming language to another.
- Typically: *high-level source code to low-level machine code*
- Provides the *abstraction* that computers understand C, Java, etc.
Why Study Compilers?

• They’re what makes programming possible!
• Useful if you’ve ever wanted to make your own language, or tweak an existing one.
• But you don’t have to know engine design to drive a car (anymore)
  – If you’re going to be a professional driver, maybe you should.
  – When things go wrong, the abstraction breaks.
• (demo)

• When programs don’t compile, the error messages are often more about what went wrong in the compiler than what you did wrong
• So understanding compilers helps you understand compiler errors!
When Things Go Wrong, part 2

Questions

Top
Textbook

• Course textbook: *Modern compiler implementation in C* (Appel)
  - Green tiger book (there are also Java and ML versions)
  - Can get an ebook via RedShelf (might be cheaper)
  - Small number of copies at the library
  - Errata, etc. at https://www.cs.princeton.edu/~appel/modern/c/
Grading

In-class exercises: 25%
Assignments: 60%
Project: 15%
Participation: up to 5% extra credit (turning on your camera in class, asking questions in class, posting on Piazza, etc.)
In-Class Exercises

• One question every class, submitted through Gradescope
• Answer them in-class if you attend live, or whenever you watch the lecture if you’re watching the recordings
• You don’t have to get them right to get credit!

• Today’s exercise: What’s one question you’d like to be able to answer by the end of this course?
Assignments

- We will have six programming assignments, each over ~2 weeks
- Each assignment will be submitted twice
  - First submission: write as much as you can; you’ll receive full credit as long as you submit anything, and I’ll give you feedback on your code
  - Second submission: I’ll actually test your code and grade you on how well it works
- Put together, the assignments will be most of a compiler for a simple C-like language
- Final project: add another feature to the compiler

- Academic integrity: don’t copy code, and cite sources!
  - High-level discussions are fine, but don’t show people your code
  - General principle: *When in doubt, ask!*

- Submitted and returned via Gradescope
Asking Questions

• In class, raise your hand anytime

• You can ask questions anonymously with PollEverywhere (https://pollev.com/wmansky771)

• On Piazza
  – Can ask/answer anonymously
  – Can post privately to instructors
  – Can answer other students’ questions

• In office hours, Tuesday 1-2 and Friday 10:30-11:30, on BB Collaborate

• If you have a question, someone else probably has the same question!
Questions

Top
INTRO TO COMPILERS
What is a Compiler?

• A compiler is a program that translates from one programming language to another.
• Typically: high-level source code to low-level machine code (object code)
  – Not always: Source-to-source translators, Java bytecode compiler, Java ⇒ Javascript, etc.
History of Compilers

• 1945: ENIAC, the first programmable computer, is built, programmed by setting switches (machine language, 1’s and 0’s)
• 1947: Kathleen Booth invents assembly language
• 1951-1952: Grace Hopper and her team invent A-0, a very low-level compiler
• 1955-1959: Hopper invents FLOW-MATIC, the first programming language with English-like syntax, which later becomes part of COBOL
• 1957: an IBM team led by John Backus releases the FORTRAN compiler, the first commercially available compiler
• 1960: FORTRAN, COBOL, ALGOL, and LISP become the four main programming languages
• 1970—: They inspire an explosion of new languages, almost all of which use compilers!
Source Code

• Optimized for human readability
  – *Expressive*: matches human ideas of grammar / syntax / meaning
  – *Redundant*: more information than needed to help catch errors
  – *Abstract*: exact computation possibly not fully determined by code

• Example C source:

```c
#include <stdio.h>

int factorial(int n) {
    int acc = 1;
    while (n > 0) {
        acc = acc * n;
        n = n - 1;
    }
    return acc;
}

int main(int argc, char *argv[]) {
    printf("factorial(6) = %d\n", factorial(6));
}
```
• Optimized for hardware
  – Hard for people to read
  – Many steps, each one simple
  – Redundancy, ambiguity reduced
  – Abstraction & information about intent are lost

• Example assembly target:

```assembly
_factorial:
## BB#0:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl 8(%ebp), %eax
movl %eax, -4(%ebp)
movl $1, -8(%ebp)
LBB0_1:
cmpl $0, -4(%ebp)
jle LBB0_3
## BB#2:
movl -8(%ebp), %eax
imull -4(%ebp), %eax
movl %eax, -8(%ebp)
movl -4(%ebp), %eax
subl $1, %eax
movl %eax, -4(%ebp)
jmp LBB0_1
LBB0_3:
movl -8(%ebp), %eax
addl $8, %esp
popl %ebp
retl
```
How to translate?

• Source code and target code aren’t just different languages – they’re trying to express different things

• Some languages are farther from machine code than others:
  – Consider: C, C++, Java, Lisp, F#, Ruby, Python, Javascript, Prolog

• Goals of translation:
  – Correctly convey what the source code meant to do
  – Best performance for the concrete computation
  – Reasonable translation efficiency (< \(O(n^3)\))
  – Maintainable implementation
Idea: Translate in Steps

• Compile via a series of program representations

• Intermediate representations are optimized for program manipulation of various kinds:
  – Semantic analysis: type checking, error checking, etc.
  – Optimization: dead-code elimination, common subexpression elimination, function inlining, register allocation, etc.
  – Code generation: finding corresponding assembly instructions

• Representations are more machine specific, less language specific as translation proceeds
(Simplified) Compiler Structure

Source code
if (b == 0) a = 0;

Lexical Analysis
- Token Stream

Parsing
- Abstract Syntax Tree

Translation and Optimization
- Intermediate Code

Code Generation

Front End
(machine independent)

Middle End
(compiler dependent)

Back End
(machine dependent)

Assembly code
CMP ECX, 0
SETBZ EAX
Typical Compiler Stages

- Lexing → token stream
- Parsing → abstract syntax
- Semantic analysis → annotated abstract syntax
- Translation → intermediate code
- Control flow analysis → control-flow graph
- Dataflow analysis → interference graph
- Register allocation → assembly
- Code emission

- Different source language features may require more/different stages
- Assembly code is not the end of the story – still have linking and loading (out of scope for this class)

- At each stage: what do we start with, what do we turn it into, and how do we get from one to the other correctly and efficiently?
Questions

Top
First Step: Lexical Analysis

• Change the *character stream* "if (b == 0) a = 0;" into *tokens*:

```
if ( b == 0 ) { a = 0 ; }
```

```
IF; LPAREN; ID("b"); EQEQ; NUM(0); RPAREN; LBRACE; ID("a"); EQ; INT(0); SEMI; RBRACE
```

• Token: data type that represents indivisible “chunks” of text:

  – Identifiers: a y11 elsex _100
  – Keywords: if else while
  – Integers: 2 200 -500 5L
  – Floating point: 2.0 .02 1e5
  – Symbols: + * ` { } ( ) ++ << >> >>>
  – Strings: "x" "He said, "Are you?"
  – Comments: (* CS476: Project 1 ... *) /* foo */

• Often delimited by *whitespace* (‘’, \t, etc.)

  – In some languages (e.g. Python or Haskell) whitespace is significant