Welcome!

• **Instructor**: Zak Kincaid
• **TA**: Shaowei Zhu


**Piazza**: [https://piazza.com/princeton/spring2020/cos320](https://piazza.com/princeton/spring2020/cos320)

**Office hours**: Monday 4:30-6:30pm (Shaowei), Wednesday 3-5pm (Zak)
or by appointment
What is a compiler?

- A compiler is a program that takes a program written in a source language and translates it into a functionally equivalent program in a target language.
  - Source languages: C, Java, OCaml, ...
  - Target languages: x86 Assembly, Java bytecode, C, ...
What is a compiler?

- A **compiler** is a program that takes a program written in a *source language* and translates it into a functionally equivalent program in a *target language*.
  - Source languages: C, Java, OCaml, ...
  - Target languages: x86 Assembly, Java bytecode, C, ...
- A compiler can also
  - Report errors & potential problems
    - Uninitialized variables, type errors, ...
  - Improve (“optimize”) the program
Why take COS320?

You will learn:

• How high-level languages are translated to machine language
• How to be a better programmer
  • What can a compiler do?
  • What can a compiler not do?
• Lexing & Parsing
• (Some) functional programming in OCaml
• A bit of programming language theory
• A bit of computer architecture
Course resources

- **Recommended textbook:**
  Modern compiler implementation in ML (Appel)

- **Real World OCaml** (Minsky, Madhavapeddy, Hickey)
  realworldocaml.org
Grading

Homework teaches the practice of building a compiler; midterm & final skew towards theory.

• 60% Homework
  • 5 assignments, not evenly weighted
  • HW1: Build an x86 simulator
  • HW2-5: Build a compiler
  • Expect homework to be time consuming!

• 20% Midterm
  • Thursday March 12, in class

• 20% Final
Homework policies

- Homework can be done individually or in pairs
- Late assignments will be penalized 1% per hour past the deadline.
- Five late passes, can submit up to 24 hours late without penalty (at most 3/HW).

Feel free to discuss with others at conceptual level.

Submitted work should be your own.
Lecture expectations

• Lecture 1: Intro
• Lecture 2: x86 (review COS217)
• Lecture 3 + k: not review
Compilers
(Programming) language = syntax + semantics

- **Syntax**: what sequences of characters are valid programs?
  - Typically specified by context-free grammar
    \[
    
    \begin{align*}
    \text{<expr> ::= } & \text{<integer>} \\
    & | \text{<variable>} \\
    & | \text{<expr> + <expr>} \\
    & | \text{<expr> * <expr>} \\
    & | (\text{<expr>})
    \end{align*}
    
    - **Semantics**: what is the behavior of a valid program?
      - *Operational semantics*: how can we execute a program?
        - In essence: an interpreter
      - *Axiomatic semantics*: what can we prove about a program?
      - *Denotational semantics*: what mathematical function does the program compute?

The job of a compiler is to translate from the syntax of one language to another, but preserve the semantics.
(Programming) language = syntax + semantics

- **Syntax**: what sequences of characters are valid programs?
  - Typically specified by context-free grammar
    
    `<expr> ::= <integer> 
    | <variable> 
    | <expr> + <expr> 
    | <expr> * <expr> 
    | (<expr>)`

- **Semantics**: what is the behavior of a valid program?
  - *Operational semantics*: how can we execute a program?
    - In essence: an interpreter
  - *Axiomatic semantics*: what can we prove about a program?
  - *Denotational semantics*: what mathematical function does the program compute?

The job of a compiler is to translate from the syntax of one language to another, but **preserve the semantics**.
```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));
}
```
factorial:
   movl  $1, %rax
   cmpq  $2, %rdi
   jl    .LBBO_2
.LBB0_1:
   imulq %rdi, %rax
   decq %rdi
   cmpq  $1, %rdi
   jg    .LBBO_1
.LBB0_2:
   retq

main:
   movl  $ .str, %rdi
   movl  $720, %rsi
   callq printf
   retq

.globl .str
.str:
   .asciz "Factorial is %ld\n"
Compiler phases (simplified)

Frontend

- Source text
  - Lexing
  - Token stream
  - Parsing
  - Abstract syntax tree
  - Translation
  - Intermediate representation
    - Optimization
    - Code generation
    - Assembly
```c
int acc = 1;
while (n > 0) {
    acc *= n;
    n --;
}
return acc;
```
```assembly
%count = alloca i64
%acc = alloca i64
store i64 %n, i64* %count
store i64 1, i64* %acc
br label %loop
%t1 = load i64, i64* %count
%t2 = icmp sgt i64 %t1, 0
br i1 %t2, label %body, label %exit
%t3 = load i64, i64* %acc
%t4 = mul i64 %t1, %t3
store i64 %t4, i64* %acc
%t5 = sub i64 %t1, 1
store i64 %t5, i64* %count
br label %loop
%t6 = load i64, i64* %acc
ret i64 %t6
```
%count = i64 %n
%acc = i64 1
br label %loop

%count2 = phi i64 %count, %count1
%acc2 = phi i64 %acc, %acc1
%t2 = icmp sgt i64 %count2, 1
br i1 %t2, label %body, label %exit
%count1 = sub i64 %count2, 1
%acc1 = mul i64 %acc2, %count2
br label %loop
%t6 = load i64, i64* %acc
ret i64 %t6

T

F

Code generation

factorial:
1  movl   $1, %rax
2  cmpq   $2, %rdi
3  jl     .LBB0_2
4  .LBB0_1:
5  imulq  %rdi, %rax
6  decq   %rdi
7  cmpq   $1, %rdi
8  jg     .LBB0_1
9  .LBB0_2:
10  retq
COS320 assignments

By the end of the course, you will build (in OCaml) a complete compiler from a high-level type-safe language ("Oat") to a subset of x86 assembly.

- HW1: X86lite interpreter
- HW2: LLVMlite compiler
- HW3: Lexing, Parsing, simple compilation
- HW4: Higher-level Features
- HW5: Analysis and Optimizations

We will use the assignments from Penn’s CIS 341, provided by Steve Zdancevic.
OCaml
modern compiler implementation in Java

andrew w. appel
Why OCaml?

- Algebraic data types + pattern matching are very convenient features for writing compilers

OCaml is a *functional* programming language

- *Imperative* languages operate by mutating data
- *Functional* languages operate by producing new data

OCaml is a *typed* language

- Contracts on the values produced and consumed by each expression
- Types are (for the most part) *automatically inferred*.
  - Good style to write types for top-level definitions
• Next week’s lecture: x86lite
  • Simple subset of x86 (~20 instructions)
  • Suitable as a compilation target for Oat

• HW1 on course webpage. Due Feb 18
  • You will implement:
    • A simulator for X86lite machine code
    • An assembler
    • A loader
  • You may work individually or in pairs