COS320: Compiling Techniques

Zak Kincaid

January 24, 2022
Welcome!

- **Instructor:** Zak Kincaid
- **TA:** Nicolas Koh
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

  - Assignments and zoom link available through **canvas**
    - Email me at least one hour before lecture if you need me to activate zoom
  - Discussion forum on **ed**
- **Office hours**: Monday 2:00-3:00pm (Zak), more TBA or by appointment
- Recommended textbook: Modern compiler implementation in ML (Appel)
- Real World OCaml (Minsky, Madhavapeddy, Hickey) [realworldocaml.org](http://realworldocaml.org)
Grading

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

- **60% Homework**
  - 5 assignments, not evenly weighted
  - *Expect homework to be time consuming!*

- **20% Midterm**
  - Wednesday March 2, in class

- **20% Final**
Homework policies

- Homework can be done individually or in pairs
- Due on Mondays at 11pm, with 1 hour grace period
- Can be submitted max 5 days late. 10% penalty per day late, with first four late days (across all assignments) waived.
- Feel free to discuss with others at conceptual level. Submitted work should be your own.
Compilers
(Programming) language = syntax + semantics

- **Syntax**: what sequences of characters are valid programs?
  - Typically specified by context-free grammar
    
    \[
    \text{<expr>} ::= \text{<integer>}
    \]
    
    | \text{<variable>}
    | \text{<expr>} + \text{<expr>}
    | \text{<expr>} \ast \text{<expr>}
    | (\text{<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?
(Programming) language = syntax + semantics

• **Syntax**: what sequences of characters are valid programs?
  • Typically specified by context-free grammar
    \[<\text{expr}> ::= <\text{integer}> | <\text{variable}> | <\text{expr}> + <\text{expr}> | <\text{expr}> * <\text{expr}> | (<\text{expr}>)\]

• **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 .LBB0_2
.LBB0_1:
    imulq %rdi, %rax
    decq %rdi
    cmpq $1, %rdi
    jg .LBB0_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
  - Optimization

Backend

- Intermediate representation
  - Optimization
  - Code generation
- Assembly
```c
int acc = 1;
while (n > 0) {
    acc *= n;
    n --;
}
return acc;
```
%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

%acc2 = phi i64 %acc, %acc1
%acc1 = mul i64 %acc2, %count2
%count1 = sub i64 %count2, 1
br label %loop

%count2 = phi i64 %count, %count1
%t2 = icmp sgt i64 %count2, 1
br i1 %t2, label %body, label %exit

%t6 = load i64, i64* %acc
ret i64 %t6

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
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-to-X86lite code generation
- HW3: Lexing, Parsing, Oat-to-LLVMlite translation
- HW4: Higher-level features
- HW5: Analysis and Optimizations

We will use the assignments from Penn’s CIS 341, provided by Steve Zdancevic.
Historical note

• First “modern” compiler for FORTRAN developed at IBM in 1957
  • Grace Hopper’s 1951 A-O loader/linker
• 18 person-years to complete
• Led by John Backus, who won 1977 Turing award
Historical note

- First “modern” compiler for FORTRAN developed at IBM in 1957
  - Grace Hopper’s 1951 A-O loader/linker
- 18 person-years to complete
- Led by John Backus, who won 1977 Turing award
- You will implement one in a semester
OCaml
• 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
- Wednesday’s lecture: x86lite
  - Simple subset of x86 (~20 instructions)
  - Suitable as a compilation target for Oat
- HW1 on course webpage. Due Feb 7.
  - You will implement:
    - A simulator for X86lite machine code
    - An assembler
    - A loader
  - You may work individually or in pairs