Lecture 1

CS 473: COMPILER DESIGN

Adapted from slides by Steve Zdancewic
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.

```
C program

gcc

a.out
```
Why Study Compilers?

• 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.
When Things Go Wrong, part 1

• (demo)

• Understanding compilers helps you understand compiler errors
Finding and Understanding Bugs in C Compilers

Xuejun Yang  Yang Chen  Eric Eide  John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

Abstract

we reported more than 525 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. In this paper we present our compiler-testing tool and the results.

Questions

Top
Class Information

- **Prerequisites:** CS 301 (languages and automata), CS 251 (trees), CS 261 (C and assembly programming)
- **Instructor:** William Mansky, office hours Tuesday 3:30-4:30, Friday 12:00-1:00, and by appointment, SEO 1331
- **TA:** Shaika Chowdhury, office hours Monday 11-1, location TBA
- Office hours are great for homework help!

- **Web site:** [https://www.cs.uic.edu/~mansky/teaching/cs473/sp20/](https://www.cs.uic.edu/~mansky/teaching/cs473/sp20/)
- **Discussion board:** [https://piazza.com/class/k3hhmc7agl86wy](https://piazza.com/class/k3hhmc7agl86wy)
- **Recorded lectures:** Blackboard
- **Assignment submission:** Gradescope
Resources

• Course textbook: *Modern compiler implementation in C* (Appel)
  – Green tiger book (there are also Java and ML versions)
  – Small number of copies at the library
  – Code, errata, etc. at https://www.cs.princeton.edu/~appel/modern/c/

• Additional reference: *Compilers – Principles, Techniques & Tools* (Aho, Lam, Sethi, Ullman)
Homework

- Two kinds of homework: written assignments and programming assignments
- Homework accepted up to 2 days late at a 20% penalty
- Programs that don’t compile may not receive credit!
- Academic integrity: don’t copy code, and cite sources!
  - You can find solutions online
  - High-level discussions are fine, but don’t show people your code
  - General principle: *When in doubt, ask!*
Grading

Assignments: 30%
Midterms (2): 40%
Final: 30%
Participation: up to 5% extra credit
Asking Questions

• In class, raise your hand anytime
• You can ask questions anonymously with PollEverywhere
• On Piazza
  – Can ask/answer anonymously
  – Can post privately to instructors
  – Can answer other students’ questions

• If you have a question, someone else probably has the same question!
Questions
What is a compiler?

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

• This is an old problem!

• Until the 1950’s: computers were programmed in assembly.

• 1951—1952: Grace Hopper developed the A-0 system for the UNIVAC I
  – She later contributed significantly to the design of COBOL

• 1957: the FORTRAN compiler was built at IBM
  – Team led by John Backus

• 1960’s: development of the first bootstrapping compiler for LISP

• 1970’s: language/compiler design blossomed

• Today: thousands of languages (most little used)
  – Some better designed than others...
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));
}
```
Target code

- Optimized for hardware
  - Machine code hard for people to read
  - Redundancy, ambiguity reduced
  - Abstraction & information about intent are lost

- Assembly language
  - then machine language

- Figure at right shows (unoptimized) 32-bit code for the factorial function

```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 machine 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:
  – Source code is expressive enough for the task
  – Best performance for the concrete computation
  – Reasonable translation efficiency (< $O(n^3)$)
  – Maintainable code
  – Correctness!
**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: instruction selection

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

Source Code
(Character stream)
if (b == 0) a = 0;

Lexical Analysis
Token Stream
Parsing
Abstract Syntax Tree
Translation and Optimization
Intermediate Code
Code Generation

Assembly Code
CMP ECX, 0
SETBZ EAX

Front End
(machine independent)

Middle End
(compiler dependent)

Back End
(machine dependent)
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

- 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