CS143: Intro

David L. Dill
Stanford University
Introduction

- Still getting organized – more info on web page tonight and tomorrow.
- Web page: cs143.stanford.edu
- Exams
  - Midterm – in class,
  - Final 6/9 3:30 PM (If it disagrees with University schedule, University wins).
- Piazza – first place to go for questions and answers.
Graded work

• Project 50%
  • Programming problems: Implement a compiler for “COOL” language
  • Parts 1 & 2 – 10% each
  • Parts 3 & 4 – 15% each

• Written assignments 10%
  • More theoretical questions
  • Submitted on Scoryst (more details later)

• Midterm 15% (in class).

• Final 25%
Outline

• What are compilers and why are they worth studying?
• Compiler organization
  • Front end
    • Lexical analysis
    • Syntactic analysis (parsing)
    • Semantic analysis
  • Back end
    • Optimization
    • Code generation
• Interpreters
• Engineering
What are compilers and why are they worth studying?
What is a compiler?

Programming language translator.

Source code

```
void *get foo(m) {
    
    3
}
```
Other programs

Human readable notation
Other programs

Human readable notation $\Rightarrow$ Program $\Rightarrow$ Something else

Often uses similar architecture/technology
Examples

Document formatters
Description languages
Graphics
Data (many kinds)
Query languages
Compiler compilers

Knowing about compilers can help you with lots of other problems
Why study computers?

Practical
- Standard architecture, ideas, tools for many applications
- Implementing a programming language is fun.

Theory: Applied formal language theory, algorithms, logic.
Compiler organization
Structure of a compiler

Source code → Front End → Intermediate language → Back End → Object code

Front End:
- Parses & checks input
- Target independent

Back End:
- Optimizes & generates code
  - Target dependent
  - x86, ARM, etc.
Betheer diagram

Front End → IR → IR Optimization → IR → Back End

↑

target independent optimizations

↑

target dependent optimizations

Oversimplified!
$m$ languages x $n$ targets
IR can be virtual machine instructions, trees, graphs, ...

Intermediate Representation

\[ L_1 \rightarrow FE_1 \rightarrow \ldots \rightarrow BE_1 \rightarrow T_1 \]
\[ L_2 \rightarrow FE_2 \rightarrow \ldots \rightarrow BE_2 \rightarrow T_n \]

m x n combinations
m front ends + n back ends
Front end
Front End

Input (string) → Lexical analysis → Syntactic analysis (Parsing) → Semantic Analysis

Concepts from linguistics

Symbol table
lexical analysis

letters $\rightarrow$ words

Characceess $\rightarrow$ Lexical Analyzer (lexer) $\rightarrow$ Tokens
lexical analysis

if \( x == y \) then \( z = 1 \), else \( z = 2 \);

tokens ?
lexical analysis

if $x = y$ then $z = 1$, else $z = 2$.
Theory in compilers

Precise definitions

Automated generation

Regular languages $\rightarrow$ automatic lexer generation

Context-free languages $\rightarrow$ automatic parser generation
Lexical Analysis Theory

Precise description — Regular expressions

\[[a-z \ A-\Z] [a-z \ A-\Z \ 0-9-\_]^*\]

Automatic generation

Regular expressions → NFA → DFA
→ lexical analyzer program
Parsing

Stream of tokens

if x == y

tree

if

==

x

y
Parsing Theory
Precised descriptions - context-free grammars
("Backus Naur Form")

Automatic generation - parser generators

\[
\text{CFG} \xrightarrow{\text{parser generator}} \text{Parser}
\]
Semantic Analysis

"Static semantics" - checked at "compile time" - catch errors before program is run.

Type checking

Other ( $1 = 2$ )

Theorey is programming language specific.
Back end
Back end

Intermediate Representation → Optimization → Code Generation → Object Code
Optimization

Transform to "equivalent", better performing code.

"better performing"

faster (may be camphorated!)

resource usage (code size, registers, memory)

"equivalent" - often compromised a bit.
Optimization

Sophisticated analysis often required.

Analyze code to understand properties.
Results say which transformations are valid/useful.

We will not cover optimization in depth.
(See CS 243)
Code Generation

Usually generates intermediate code

Map intermediate representation to instructions

\[ x + y = z \]

put \( x \) value in right place (e.g., register)

put \( y \) in right place

add instruction

put result in right place.
Code Generation

Issues

- Managing scarce resources (e.g., registers)
- Minimizing redundant moves

Complex machine instructions may be able to do several intermediate operations
Interpreters
Interpreters vs Compilers

.Compiler
  Source code \rightarrow \text{compiler} \rightarrow \text{object code}

.input \rightarrow \text{object code} \rightarrow \text{output}
Interpreters vs Compilers

Early phases — esp. lexical & syntax — are similar.
Source code → Frontend → IR

IR → Interpreter → Output

E.g.
Java
IR = JVM code
Interpreter = JVM
Engineering
Engineering issues
Modern computers are frustratingly complex

- Programming language inconsistency
- Target language complexity
- Complex optimization
- Code evolution