Welcome to CSE131b: Compiler Construction

Lingjia Tang

pic from: http://xkcd.com/303/
• Course Information
• What are compilers?
• Why do we learn about them?
• History of compilers
• Structure of compilers
• A bit info on course projects
Course Staff

• Instructor: Lingjia Tang (2108)
  • lingjia@cs.ucsd.edu
• TA: Xinxin Jing
  • x7jin@cs.ucsd.edu
• Tutors: Haronid Moncivais Miller and more
Welcome to CSE 131!

Instructor: Dr. Lingjia Tang
- Office: CSE 2108
- Email: lingjia@cs.ucsd.edu
- Office Hours: Mon/Wed 3-4:00pm

TA: Xinxin Jin
- Office Hours: TBA
- Location: TBA
- Email: x7jin@cs.ucsd.edu

Course Info:
- Lecture: HSS 1330, MW 5:00p - 6:20p
- Discussion: TBA

Course Textbook:

Slides
- 01-intro to compiler (04/01)

CSE 131 -- Compiler Construction
Spring 2013, Instructor: Lingjia Tang

Course Description
This course is an introduction to compiler construction. Topics covered will include the following: lexical scanning, parsing, abstract syntax trees, semantic analysis, intermediate code generation, optimization, and code generation.

- You can't modify a constant, float upstream, win an argument with the IRS, or satisfy this compiler* - a real compiler error message from Apple's MPWC compiler. More funny compiler msgs.

Guidelines and Grading Info.

- REQUIRED: PIAZZA
  - I will be using PIAZZA for the majority of class correspondence. Piazza connects everyone in the class as well as the instructors. Use it to ask questions, discuss homeworks/projects, etc. Myself and the TAs will be checking Piazza regularly to chime in.
  - URL: https://piazza.com/ucsd/spring2013/cse131b/home
  - The initial class roster will receive invites, however it is your responsibility to make sure you are signed up. If you are not already on, click the link and sign up.
  - I will put slides and other materials on Piazza from time to time in PDF format.

- Grading
  - The grade for 131 will be based on programming projects, one midterm, and a final, as follows:
    - Projects: 45%
    - Midterm: 20%
    - Final: 35%
    - Earn extra credit by class participation: 5%
  - The final will be inclusive of all course material.
  - Late project submissions are not encouraged. You will have three grace days during the quarter. I.e., you can turn one project in three days late, or three assignments in one day late, etc. I recommend not spending those days frivolously early in the quarter. After you have spent your grace days, late project will be accepted, but with no guarantees that they will be graded, and with significant penalties if they are. Anytime after the end of class counts as a day late. The second day begins 24 hours later.
  - You have the right of appeal for grading on all tests. You have one week from the time the midterms are returned to make appeals, including addition errors on your score. Check it over carefully when you get it. All appeals must be made in writing and given to the instructor.

- Integrity
  - Cheating WILL be taken seriously. It is not fair to honest students to take cheating lightly, nor is it fair to the cheater to let him/her go on thinking that is a reasonable alternative in life. Don't test me on this one.
  - The following is not considered cheating:
    - Collaboration on homework
    - Using instructor's notes
    - Reading the textbook
    - Using the internet
    - Discussing with classmates
Grading Policies

- **Project** 35%
- **Midterm** 45%
- **Final** 20%
Grading Policies

Extra credits:
• Class participation
• Piazza participation
• Projects/exam extra credits
Prerequisites

- CSE 70 / CSE 110
- CSE 100
- CSE 105
- CSE 130
What is a compiler?
• Compiler
• Compiler

program in source language → Compiler
• Compiler

program in *source* language → Compiler → program in *target* language
• Compiler

program in source language → Compiler → program in target language

Error message
- **Compiler**

  program in *source* language → **Compiler** → program in *target* language

  Error message → **Compiler** → Output

  Input
• **Compiler**

  - Program in *source* language
  - **Compiler**
  - Error message
  - Program in *target* language
  - Input
  - Output

• **Translates**

Monday, April 1, 13
• Compiler

program in source language

Compiler

program in target language

Error message

Input

Output

• Translates

• Typically lower the level of abstraction of the program
• Compiler

- Translates

- Typically lower the level of abstraction of the program
  - High-level programming language -> machine code
    - C/C++ -> X86, etc
• Compiler

program in \emph{source} language \quad \xrightarrow{\text{Compiler}} \quad \text{program in } \emph{target} \text{ language}

- Error message
- Output

• Translates

• Typically lower the level of abstraction of the program
  • High-level programming language $\rightarrow$ machine code
    (C/C++ $\rightarrow$ X86, etc)

• We expect the program produced by the compiler to be better, (faster, consumes less memory, etc) than the original program
• Compiler

program in *source* language → Compiler → program in *target* language

Input → Compiler → Output

Error message
• Compiler

program in *source* language → Compiler → program in *target* language

Input → Error message → Output

• Interpreter
- Compiler

  program in source language → Compiler → program in target language

  Error message

- Interpreter
• **Compiler**

```
program in source language
```

```
Compiler
```

```
program in target language
```

```
Error message
```

```
Input
```

```
Output
```

• **Interpreter**

```
Program
```

```
Input
```

```
Interpreter
```

```
```

```
Monday, April 1, 13
```
- **Compiler**

  - Program in *source* language → **Compiler** → program in *target* language
  - Error message
  - Input → **Compiler** → Output

- **Interpreter**

  - Program → **Interpreter** → output
  - Input
• **Compiler**

  ![Diagram of a compiler process]

  - program in *source* language
  - Error message
  - program in *target* language

  Static: before runtime

• **Interpreter**

  ![Diagram of an interpreter process]

  - Input
  - Output

  Program Input

  output
• Compiler

program in *source* language

Static: before runtime

Compiler

Error message

program in *target* language

Input

Output

• Interpreter

Program

Input

Interpreter

Dynamic: during runtime

output
• Compiler

Static: before runtime

• Interpreter

Dynamic: during runtime

Examples: python, shell, javascript, PHP
• Hybrid

• Example: Java

program in source language → Compiler → intermediate program

Input → Virtual Machine → output
• Hybrid

• Example: Java

- program in source language
- Compiler
- intermediate program
- Interpreter
- Just-in-time compiler
- Garbage Collection
- Virtual Machine
- native machine code
- output
- Hybrid
- Example: Java

```
program in source language \rightarrow \textbf{Compiler} \rightarrow \text{intermediate program} \rightarrow \text{Virtual Machine}
```

```
\text{Input} \rightarrow \textbf{Interpreter} \rightarrow \textbf{Just-in-time compiler} \rightarrow \text{native machine code} \rightarrow \text{output}
```

```
\text{Garbage Collection}
```

```
\text{Static}
```

Monday, April 1, 13
• Hybrid

• Example: Java

program in source language → Compiler → intermediate program

Static

Input → Just-in-time compiler → native machine code → output

Runtime

Virtual Machine

Garbage Collection
• Hybrid

• Example: Java

Sacrifice efficiency for portability, “safety” and other dynamic features

program in source language → Compiler → intermediate program

Input → Just-in-time compiler → Garbage Collection → Virtual Machine → native machine code

output
Why Learning Compiler?

• Build awesome, complex software
• Theory meets practice
  • remember DFA, CFG in CSE105?
• Understand how SW interacts with HW
• Understand program language designs and features
Why Learning Compiler?
cont.

• My code runs faster than your code...
  • consumes less power
  • consumes less memory
  • more reliable..
Why Learning Compiler? cont.

• Who use compilers?
  • everyone!

• Who develop compilers?
  • many many companies....
Why Learning Compiler? cont.

• Who use compilers?
  • everyone!

• Who develop compilers?
  • many many companies....
Why Learning Compiler? cont.

- Who use compilers?
  - everyone!

- Who develop compilers?
  - many many companies....
Why Learning Compiler? cont.

- Who use compilers?
  - everyone!

- Who develop compilers?
  - many many companies....

program in **PHP**

program in **C++**
Why Learning Compiler? cont.

- Who use compilers?
  - everyone!

- Who develop compilers?
  - many many companies....

program in PHP

HipHop for

program in C++
Goal of this course

• For you to become compiler NINJA!
History of Compiler

- Machine Language
- Assembly
  - programming too time consuming
History of Compiler

- Machine Language
- Assembly
  - programming too time consuming
- High-level language
• Grace Hopper

pics from wikipedia
Grace Hopper conceptualized the idea of machine-independent programming languages. COBOL - (1959)

"Nobody believed that," she said. "I had a running compiler and nobody would touch it. They told me computers could only do arithmetic."
- Grace Hopper
  conceptualized the idea of machine-independent programming languages
  COBOL - (1959)
  "Nobody believed that," she said. "I had a running compiler and nobody would touch it. They told me computers could only do arithmetic."

- John Backus
  Won ACM Turing Award “for profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on FORTRAN"
• Compilers: map high-level abstract language to assembly language
• Compilers: map high-level abstract language to assembly language

• Simple mapping of a program to assembly language produces inefficient execution
  • Higher the level of abstraction, more inefficiency
  • If not efficient, high-level abstractions are useless
Compilers: map high-level abstract language to assembly language

Simple mapping of a program to assembly language produces inefficient execution

- Higher the level of abstraction, more inefficiency
- If not efficient, high-level abstractions are useless

Compilers need to:

- provide a high level abstraction
- with performance of giving low-level instruction
- bridge the efficiency gap b/t high-level PL and low-level processor/ISAs

saman amarasinghe's slide
The Structure of a Modern Compiler
The Structure of a Modern Compiler


Front end
The Structure of a Modern Compiler

Source Code

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

Back end

Machine Code

Monday, April 1, 13
while (y < z) {
    int x = a + b;
    y += x;
}

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical Analysis</td>
<td>Syntax Analysis</td>
<td>Semantic Analysis</td>
</tr>
<tr>
<td>IR Generation</td>
<td>IR Optimization</td>
<td>Code Generation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optimization</td>
</tr>
</tbody>
</table>

**Lexical analysis (Scanning):** Identify logic pieces of the description.
Lexical analysis (Scanning): Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)
while (y < z) {
    int x = a + b;
    y += x;
}

Syntax analysis (Parsing): Identify how those pieces relate to each other
while (y < z) {
    int x = a + b;
    y += x;
}

Syntax analysis (Parsing): Convert a linear structure – sequence of tokens – to a hierarchical tree-like structure - abstract syntax tree (AST)
Syntax Analyzer (Parser)

```c
int * foo(i, j, k))
    int i;
    int j;
    {
        for(i=0; i < j) {
            if(i>j) {
                return j;
            }
        }
    }
```

- Extra parentheses
- Missing increment
- Not an expression
- Not a keyword
while (y < z) {
    int x = a + b;
    y += x;
}

**Semantic Analysis**: Identify the meaning of the overall structure
while (y < z) {
    int x = a + b;
    y += x;
}

**Semantic Analysis:** Rules of the language are checked (variable declaration, type checking)
```c
int * foo(i, j, k)
{
    int i;
    int j;
    int x;
    x = x + j + N;
    return j;
}
```

- Type not declared
- Mismatched return type
- Uninitialized variable used
- Undeclared variable
Intermediate Representation (IR) Generation:
Generate intermediate code
while (y < z) {
    int x = a + b;
    y += x;
}

Loop: x = a + b
     y = x + y
    _t1 = y < z
    if _t1 goto Loop

**IR Generation:**

- Makes it easy to port compiler to other architectures (e.g. X86 to MIPS)
- Enables optimizations that are not machine specific
while (y < z) {
    int x = a + b;
    y += x;
}

Loop: x  = a + b
      y  = x + y
      _t1 = y < z
    if _t1 goto Loop

IR Optimization: Optimize intermediate code (machine-independent)
while (y < z) {
    int x = a + b;
    y += x;
}

x = a + b

Loop: y = x + y
	_t1 = y < z
	if _t1 goto Loop

IR Optimization: Optimize intermediate code
(machine-independent)
while (y < z) {
    int x = a + b;
    y += x;
}

x = a + b

Loop: y = x + y
    _t1 = y < z
    if _t1 goto Loop

**Code Generation**: Intermediate code is translated into native code
while (y < z) {
    int x = a + b;
    y += x;
}

add $1, $2, $3

Loop: add $4, $1, $4
slt $6, $1, $5
beq $6, loop

**Code Generation**: Intermediate code is translated into native code
while (y < z) {
    int x = a + b;
    y += x;
}

add $1, $2, $3

Loop: add $4, $1, $4
slt $6, $1, $5
beq $6, loop

**Code Optimization**: machine dependent optimization (register allocation, instruction selection, peephole, etc)
while (y < z) {
   int x = a + b;
   y += x;
}

add $1, $2, $3
Loop: add $4, $1, $4
blt $1, $5, loop

**Code Optimization:** machine dependent optimization (register allocation, instruction selection, peephole, etc)
Modern Compilers

- Matured Frontend
- Heavy Backend
- many passes of optimization

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

while (y < z) {
  int x = a + b;
  y += x;
}
Compiler Optimization

• Frances E. Allen

• first woman to win Turing Award

• “Introduced many of the abstractions, algorithms, and implementations that laid the groundwork for automatic program optimization technology”
Architecture of GCC (GNU Compiler Collection)

Source Code

AST

GENERIC

High GIMPLE

SSA

Low GIMPLE

RTL

IRs

many optimization passes here

Machine Code
Architecture of GCC (GNU Compiler Collection)

Source Code → AST → GENERIC → High GIMPLE → SSA → Low GIMPLE → RTL → Machine Code

FrontEnd supports: C (gcc), C++ (g++), Objective-C, Objective-C++, Fortran (gfortran), Java (gcj), Ada (GNAT), and Go (gccgo)

IRs: many optimization passes here

Machine code: X86, ARM, Alpha, MIPS, SPARC, PowerPC, etc
Architecture of GCC (GNU Compiler Collection)

FrontEnd supports:
C (gcc), C++ (g++), Objective-C, Objective-C++, Fortran (gfortran), Java (gcj), Ada (GNAT), and Go (gccgo)

IRs
many optimization passes here

Machine code:
X86, ARM, Alpha, MIPS, SPARC, PowerPC, etc

Jeanne Ferrante et. al

Monday, April 1, 13
Architecture of LLVM

• LLVM (Low Level Virtual Machine)
  • developed in UIUC (2000)
  • “language-agnostic” design
This course covers

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

Midterm

while (y < z) {
    int x = a + b;
    y += x;
}
The Course Project

• Compiler: Decaf -> MIPS
  • have a working compiler by the end of the quarter
  • will be able to run your Decaf programs!

• Source Language: Decaf
  • Custom programming language: similar to Java or C++
    (simplified feature set)
  • Objected-oriented, inheritance, Strong type

• Target language: MIPS
  • RISC ISA

• Implementation: C++
Programming Assignments

Decaf

- Source Code
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization
- MIPS

- P1 10% ~1 week
- P2 10% 1.5 week
- P3 10% 3 weeks
- P4 15% 4 weeks

Monday, April 1, 13
• 2 person group
• find your partner by the end of this week
• Group members get the same grade
• Honor code:
  • Discussion among groups is encouraged
  • But implement the solution on your own
• Discussion Session ?