Outline

• Course Organization
  – General course information
  – Homework & project information

• Introduction to Compilers
  – What are they?
  – Why do we need them?
  – What is their general structure?
# General Information

<table>
<thead>
<tr>
<th>When</th>
<th>MWF 10:10 - 11:00AM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where</td>
<td>Phillips 213</td>
</tr>
<tr>
<td>Instructor</td>
<td>Tim Teitelbaum</td>
</tr>
<tr>
<td>Teaching Assistant</td>
<td>TBD</td>
</tr>
<tr>
<td>Course staff email</td>
<td><a href="mailto:cs412-l@cs.cornell.edu">cs412-l@cs.cornell.edu</a></td>
</tr>
<tr>
<td>Web page</td>
<td>courses.cs.cornell.edu/cs412</td>
</tr>
<tr>
<td>Newsgroup</td>
<td>cornell.class.412</td>
</tr>
</tbody>
</table>
Important

• CS 413 is required!
• Large implementation project
• Substantial amount of theory
Textbooks

• Optional texts
  
  

• They will be on reserve in Engineering Library
Work Distribution

• Theory:
  - Homeworks = 20%
    • 4 homeworks: 5% each
  - Exams = 35%
    • 2 prelims: 17% and 18%; no final exam

• Practice:
  - Programming Assignments = 45%
    • 6 assignments: 5/9/9/9/9/9
    • Project demo
Homeworks

• 4 homework assignments
  – Three assignments in first half of course
  – One homework in second half

• Not done in groups
  – do your own work
Project

• Implementation:
  - Designed language = a subset of Java
  - Generated code = assembly x86
  - Implementation language = Java

• 5 programming assignments

• Groups of 3-4 students
  - Usually same grade for all
  - Group information due Friday
  - We will respect consistent preferences
Assignments

• Due at beginning of class
  – Homeworks: paper turn in (at beginning of class)
  – Project files: electronic turn in (day before class)
  – Assignments managed with Course Management System (CMS)

• Late homework, programming assignments increasingly penalized
  – Penalty linearly increasing: 10% per day
  – 1 day: 10%, 2 days: 20%, 3 days: 30%, etc.
Why Take This Course?

• CS412/413 is an elective course

• Reason #1: understand compilers/languages
  - Understand code structure
  - Understand language semantics
  - Understand relation between source code and generated machine code
  - Become a better programmer
Why Take This Course? (ctd.)

• **Reason #2:** nice balance of theory and practice:
  
  - **Theory:**
    • Lots of mathematical models: regular expressions, automata, grammars, graphs, lattices
    • Lots of algorithms that use these models
  
  - **Practice:**
    • Apply theoretical notions to build a real compiler
    • Better understand why “theory and practice are the same in theory, but different in practice”
Why Take This Course? (ctd.)

• Reason #3: Programming experience
  - Write a large program that manipulates complex data structures
  - Learn how to be a better programmer in groups
  - Learn more about Java and Intel x86 architecture and assembly language
Why Take This Course? (ctd.)

• Reason #4: Technical background for emerging field of software assurance
  - Software assurance will be major priority of coming decade
  - Bug-finding and security-violation finding tools build on compiler techniques
What Are Compilers?

• Compilers translate *information* from one *representation* to another.
• Most commonly, the information is a *program*
• Typically
  - “Compilers” translate from high-level *source* code to low-level code (e.g., *object* code)
  - “Translators” transform representations at the same level of abstraction
Examples

• Typical compilers: gcc, javac

• Non-typical compilers:
  – latex (document compiler):
    • Transforms a LaTeX document into DVI printing commands
    • Input information: document (not program)
  – C-to-Hardware compiler:
    • Generates hardware circuits for C programs
    • Output is lower-level than typical compilers

• Translators:
  – f2c: Fortran-to-C translator (both high-level)
  – latex2html: LaTeX-to-HTML (both documents)
  – dvi2ps: DVI-to-PostScript (both low-level)
In This Class

• We will study typical compilation: from programs written in high-level languages to low-level object code and machine code

• Most of the principles and techniques in this course apply to non-typical compilers and translators
Why Do We Need Compilers?

- It is difficult to write, debug, maintain, and understand programs written in assembly language.

- Tremendous increase in productivity when first compilers appeared (about 55 years ago).

- There are still few cases when it is better to manually write assembly code:
  - E.g., to access low-level resources of the machine (device drivers).
  - These code fragments are very small; the compiler handles the rest of the code in the application.
Overall Compiler Structure

High-level source code

Compiler

Low-level machine code
Source Code

• Optimized for human readability
  - Matches human notions of grammar
  - Uses named constructs such as variables and procedures

```c
int expr(int n)
{
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```
Assembly and Machine Code

- Optimized for hardware
  - Consists of machine instructions; uses registers and unnamed memory locations
  - Much harder to understand by humans

```assembly
lda $30,-32($30)   addq $3,1,$4
stq $26,0($30)    mull $2,4,$2
stq $15,8($30)    ldq $26,0($30)
bis $30,$30,$15   ldl $15,8($30)
bis $16,$16,$1    mull $2,4,$2
stl $1,16($15)    stl $2,20($15)
lds $f1,16($15)   stl $2,20($15)
sts $f1,24($15)   br $31,$33
ldl $5,24($15)    $33:
bi$5,$5,$2   bis $15,$15,$30
s4addq $2,0,$3    ldq $26,0($30)
ldl $4,16($15)    ldl $15,8($30)
mull $4,$3,$2    addq $30,32,$30
ldl $3,16($15)    ret $31,($26),1
```
Translation Efficiency

• **Goal:** generate machine code that describes the same computation as the source code

• Is there a unique translation?

• Is there an algorithm for an “ideal translation”? (ideal = either fastest or smallest generated code)

• Compiler optimizations = find *better* translations!
Example: Output Assembly Code

Unoptimized Code

lda $30,-32($30)
stq $26,0($30)
stq $15,8($30)
bis $30,$30,$15
bis $16,$16,$1
stl $1,16($15)
lds $f1,16($15)
sts $f1,24($15)
ldl $5,24($15)
bis $5,$5,$2
s4addq $2,0,$3
ldl $4,16($15)
mull $4,$3,$2
ldl $3,16($15)
addq $3,1,$4
mull $2,$4,$2
ldl $3,16($15)
addq $3,1,$4
mull $2,$4,$2
stl $2,20($15)
ldl $0,20($15)
br $31,$33

$33:
bis $15,$15,$30
ldq $26,0($30)
ldq $15,8($30)
addq $30,32,$30
ret $31,($26),1

Optimized Code

s4addq $16,0,$0
mull $16,$0,$0
addq $16,1,$16
mull $0,$16,$0
mull $0,$16,$0
ret $31,($26),1
Translation Correctness

• The generated code must execute precisely the same computation as in the source code

• Correctness is very important!
  – hard to debug programs with broken compiler…
  – implications for development cost, security
  – this course: techniques known to ensure correct translation
How To Translate?

• Translation is a complex process
  – source language and generated code are very different

• Need to structure the translation
  – Define intermediate steps
  – At each step use a specific program representation
  – More machine-specific, less language-specific as translation proceeds
CMP $0, ECX
CMOVZ EDX, ECX

if (b == 0) a = b;

Simplified Compiler Structure

Source code

if (b == 0) a = b;

Understand source code

Intermediate code

Optimize

Intermediate code

Generate assembly code

Assembly code

cmp $0, ecx
cmovz edx, ecx

Front end
(machine-independent)

Optimizer

Back end
(machine-dependent)
Simplified Front-End Structure

Source code (character stream)

if (b == 0) a = b;

Lexical Analysis

Token stream

Syntax Analysis

Abstract syntax tree

Semantic Analysis

Abstract syntax tree

Lexical errors

Syntax errors

Semantic errors
Analogy

• Front end can be explained by analogy to the way humans understand natural languages

• Lexical analysis
  - Natural language: “He wrote the program”
    words: “he” “wrote” “the” “program”
  - Programming language “if (b == 0) a = b”
    tokens: “if” “(" “b” “==” “0” “")” “a” “=” “b”
Analogy (ctd)

- **Syntactic analysis**
  - **Natural language:**
    
    ![Sentence Tree](image)

  - **Programming language**
    
    ![If Statement Tree](image)
Analogy (ctd)

- **Semantic analysis**
  - Natural language:
    
      He  wrote  the  computer  
      noun  verb  article  noun

    Syntax is correct; semantics is wrong!

  - Programming language
    
    if ( b  ==  0 )  a = foo  
    test  assignment

    if a is an integer variable and foo is a procedure, then the semantic analysis will report an error
Big Picture

Source code

Lexical Analysis
Syntax Analysis
Semantic Analysis
Optimization
Code Generation

Assembly code

Object code (machine code)

Fully-resolved object code (machine code)

Executable image
Tentative Schedule

Lexical analysis 3 lectures
Syntax analysis 6 lectures
Semantic analysis 5 lectures
Prelim #1
Simple code generation 6 lectures
Analysis 8 lectures
Optimizations 3 lectures
Advanced topics 3 lectures
Prelim #2
Advanced topics 3 lectures