Course Information

- MWF 1:25-2:15 PM in Phillips 203
- Instructor: Andrew Myers
- Teaching Assistants: Anthony Jawad
- E-mail: cs4120-l@cs.cornell.edu
- Web page: http://www.cs.cornell.edu/courses/cs4120
- Newsgroup: cornell.class.cs4120

Academic integrity

- Taken seriously.
- Do your own (or your group’s) work.
- Report who you discussed homework with (whether student in class or not).
CS 4121 is required!

– most coursework is in the project

Textbooks

• Required text
  – Modern Compiler Implementation in Java. Andrew Appel.
  – on reserve in Engineering Library

• Optional texts
  – Compilers—Principles, Techniques and Tools. Aho, Lam, Sethi and Ullman (The Dragon Book)
  – Advanced Compiler Design and Implementation. Steve Muchnick.

Work

• Homeworks: 4, 20% total
  – 5/5/5/5
• Programming Assignments: 6, 50%
  – 5/7/8/10/10/10
• Exams: 2 prelims, 30%
  – 15/15
  – No final exam

Homeworks

• Three assignments in first half of course; one homework in second half
• Not done in groups—you may discuss with others but do your own work
  – Report who you discussed homework with
Projects

- Six programming assignments
- Implementation language: Java
  - talk to us if your group wants to use something else (e.g., OCaml)
- Groups of 3-4 students
  - same group for entire class (ordinarily)
  - same grade for all (ordinarily)
  - workload and success in this class depend on working and planning well with your group. Be a good citizen.
  - tell us early if you are having problems.
- End of this class: some time to form groups
  - create your group on CMS for PA1.
  - contact us if you are having trouble finding a group.

Assignments

- Due at beginning of class
- Late homeworks, programming assignments increasingly penalized
  - 1 day: 5%, 2 days: 15%, 3 days: 30%, 4 days: 50%
  - weekend = 1 day
  - Extensions often granted, but must be approved 2 days in advance
- Projects submitted via CMS

Why take this course?

- CS 4120 is an elective course
- Expect to learn:
  - practical applications of theory, algorithms, data structures
  - parsing
  - deeper understanding of what code is
  - how high-level languages are implemented
  - a little programming language semantics
  - Intel x86 architecture, Java
  - how programs really execute on computers
  - how to be a better programmer (esp. in groups)

What are Compilers?

- Translators from one representation of program code to another
- Typically: high-level source code to machine language (object code)
- Not always:
  - Java compiler: Java to interpretable bytecodes
  - Java JIT: bytecode to executable image
Source Code

- Source code: optimized for human readability
  - expressive: matches human notions of grammar
  - redundant to help avoid programming errors
  - computation possibly not fully determined by code

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

Machine code

- Optimized for hardware
  - Redundancy, ambiguity reduced
  - Information about intent and reasoning lost
  - Assembly code ≈ machine code

```assembly
expr:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    movl 8(%ebp), %eax
    movl %eax, %edx
    imull 8(%ebp), %edx
    movl 8(%ebp), %eax
    incl %eax
    imull %eax, %edx
    movl 8(%ebp), %eax
    incl %eax
    imull %edx, %eax
    sall $2, %eax
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax
    leave
    ret
```

Example (Output assembly code)

<table>
<thead>
<tr>
<th>Unoptimized Code</th>
<th>Optimized Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr:</code></td>
<td><code>expr:</code></td>
</tr>
<tr>
<td><code>pushl %ebp, %ebp</code></td>
<td><code>pushl %ebp, %ebp</code></td>
</tr>
<tr>
<td><code>subl $4, %esp</code></td>
<td><code>movl 8(%ebp), %edx</code></td>
</tr>
<tr>
<td><code>movl %eax, %edx</code></td>
<td><code>movl %edx, %eax</code></td>
</tr>
<tr>
<td><code>imull %edx, %eax</code></td>
<td><code>movl %edx, %eax</code></td>
</tr>
<tr>
<td><code>imull %edx, %eax</code></td>
<td><code>imull %edx, %eax</code></td>
</tr>
<tr>
<td><code>imull %edx, %eax</code></td>
<td><code>imull %edx, %eax</code></td>
</tr>
<tr>
<td><code>sall $2, %eax</code></td>
<td><code>movl -4(%ebp), %eax</code></td>
</tr>
<tr>
<td><code>movl %eax, -4(%ebp)</code></td>
<td><code>movl %eax, -4(%ebp)</code></td>
</tr>
<tr>
<td><code>movl -4(%ebp), %eax</code></td>
<td><code>movl -4(%ebp), %eax</code></td>
</tr>
<tr>
<td><code>leave</code></td>
<td><code>leave</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>ret</code></td>
</tr>
</tbody>
</table>

How to translate?

- Source code and machine code mismatch
- Goal:
  - source-level expressiveness for task
  - best performance for concrete computation
  - reasonable translation efficiency (< O(n^3))
  - maintainable compiler code
How to translate correctly?

- Programming languages describe computation precisely
- Therefore: translation can be precisely described (a compiler can be correct)
- Correctness is very important!
  - hard to debug programs with broken compiler...
  - non-trivial: programming languages are expressive
  - implications for development cost, security
  - this course: techniques for building correct compilers
  - some compilers have been proven correct!

[X. Leroy, Formal Certification of a Compiler Back End, POPL '06]

How to translate effectively?

Idea: translate in steps

- Compiler uses a series of different program representations.
- Intermediate representations that are good for program manipulations of various kinds (analysis, optimization, code generation).
Compilation in a Nutshell 2

Intermediate Code Generation

Optimization, Code Generation

Register allocation, optimization

boolean int
int 0 int a int b

SEQ(CJUMP(TEMP(b) == 0, L1, L2),
LABEL(L1),
TEMP(a) = TEMP(b)
LABEL(L2))

 cmp ecx, 0
 cmovz [ebp+8],ecx

cmp r, 0
jnz L2
L1: mov r, r
L2:

Simplified Compiler Structure

Source code (character stream)

Lexical analysis

Token stream

Parsing

Front end
(machine-independent)

Abstract syntax tree

Intermediate Code Generation

Program analysis & Optimization

Intermediate code

Control flow graphs

Back end
(machine-dependent)

Code generation

Source code

Parser

Semantic analysis

Intermediate code

Prelim #1

Code generation

Separate compilation and objects

Optimization

Prelim #2

Run-time, link-time support

Advanced topics

 Even bigger picture

Source code

Compiler

Assembly code

Assembler

Object code (machine code + symbol tables)

Linker

Fully-resolved object code (machine code + symbol tables, relocation info)

Loader

Executable image in memory

Schedule

• Detailed schedule on web page, with links

Lexical analysis and parsing: 6
Semantic analysis: 5
Intermediate code: 4
Prelim #1
Code generation: 3
Separate compilation and objects: 4
Optimization: 8
Prelim #2
Run-time, link-time support: 2
Advanced topics: 7
First step: Lexical Analysis

Source code (character stream) → Lexical analysis → Token stream → Parsing → Abstract syntax tree → Intermediate Code Generation → Intermediate code → Code generation → Assembly code

What is Lexical Analysis?

- Converts character stream to token stream of pairs \( \langle \text{token type, attribute} \rangle \)

```java
if (x1 * x2<1.0) {
    y = x1;
}
```

Token stream

- Gets rid of whitespace, comments
- Only \( \langle \text{Token type, attribute} \rangle \):
  - \( \langle \text{Id, "x"}, \langle \text{Float, 1.0e0} \rangle \)
  - Token location preserved for debugging, run-time/compile-time error messages (source file, line number, character posn...)
    - \( \langle \text{Id, "x", "Main.java", 542} \rangle \)

- Issues:
  - how to specify tokens
  - how to implement tokenizer/lexer