Course Information

- MWF 10:10 - 11:00AM in Phillips 203
- Faculty: Andrew Myers
- Teaching Assistants: Michael Clarkson, Sunny Gleason, Lantian Zheng
- E-mail: cs412@cs.cornell.edu
- Web page: http://www.cs.cornell.edu/courses/cs412
- Newsgroup: cornell.class.cs412

Textbooks

- Required text
  - Modern Compiler Implementation in Java. Andrew Appel.
- Optional texts
  - Compilers – Principles, Techniques and Tools. Aho, Sethi and Ullman (The Dragon Book)
  - Advanced Compiler Design and Implementation. Steve Machnirck.
- Java reference
- All are on reserve in Engineering Library

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
  - Write down who you discussed problems with

Projects

- Six programming assignments
- Groups of 3-4 students
  - same grade for all
- Group information due Friday
  - we will respect consistent preferences
- Java will be implementation language

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
- Project files turned in to CSUGLAB directory

Why take this course?

- CS412 is an elective course
- Expect to learn:
  - practical applications of theory
  - parsing
  - deeper understanding of code
  - manipulation of complex data structures
  - how high-level languages are implemented in machine language
  - a little programming language semantics
  - Intel x86 architecture, Java
  - how to be a better programmer (esp. in groups)

What are Compilers?

- Translators from one representation of a program 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 lost
  - Assembly code = machine code

How to translate?

- Source code and machine code mismatch
- Some languages farther from machine code than others (“higher-level”)
- Goal:
  - source-level expressiveness for task
  - best performance for concrete computation
  - reasonable translation efficiency (< O(n³))
  - maintainable code

Example (Output assembly code)

Unoptimized Code

Optimized Code

Correctness

- 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

How to translate effectively?

Idea: Translate in Steps

- Series of program representations
- Intermediate representations optimized for program manipulations of various kinds (checking, optimization)
- More machine-specific, less language-specific as translation proceeds
**Simplified Compiler Structure**

- Source code (character stream)
  - if (b == 0) a = b;

- Lexical analysis
- Token stream
- Intermediate code
- Front end (machine-independent)
- Back end (machine-dependent)

- Parsing
- Abstract syntax tree
- Code generation
- Assembly code

- Intermediate Code Generation
- Code generation
- Assembly code

---

**Big picture**

- Source code
  - Compiler
  - Assembler
- Object code (machine code)
  - Linker
- Fully-resolved object code (machine code)
  - Loader
- Executable image

---

**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)
  - if (b == 0) a = b;

- Lexical analysis
- Token stream
  - /G68
  - /G69
- 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 (token type, attribute)
  - if (x1 * x2<1.0) {
    - y = x1;
  }

- Token stream
  - /G68
  - /G69

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

---

**Token stream**

- Gets rid of whitespace, comments
- Only ‘Token type, attribute’:
  - (Id, “x”), (Float, 1.0e0)
- Token location preserved for debugging, error messages (source file, line number)
  - (Id, “x”, “Main.java”, 542)

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