Course Information

- MWF 1:25-2:15 PM in Hollister 306
- Instructor: Andrew Myers
- Teaching Assistants: Wenzel Jakob, Gregor Stocks
- E-mail: cs4120-l@cs.cornell.edu
- Web page: http://www.cs.cornell.edu/courses/cs4120
- Newsgroup: cornell.class.cs4120

Outline

- About this course
- Introduction to compilers
  - What are compilers?
  - Why should we learn about them?
  - Anatomy of a compiler
- Introduction to lexical analysis
  - Text stream to tokens

4 = 5 & 0 = 1

- CS 4120 and 5120 are really the same course
  - same lectures
  - same assignments or nearly so
  - 5120 is for MEng students, 4120 for others
- CS 4121 (5121) is required!
  - most coursework is in the project
Textbooks

- Lecture notes provided; no required textbook
- On reserve in Uris Library:
    (strength: parsing)
  - *Modern Compiler Implementation in Java*. Andrew Appel.
    (strength: translation)
    (strength: optimization)

Work

- Homeworks: 4, 20% total
  - 5/5/5/5
- Programming Assignments: 7, 50%
  - 5-10% each
- Exams: 2 prelims, 30%
  - 15/15
  - No final exam

Academic integrity

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

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

- Seven 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 midnight on due date
- 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
- Solutions available 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 %eax, %eax
    incl %eax
    imull %eax, %edx
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax
    leave
    ret
```

**Example (Output assembly code)**

**Unoptimized Code**

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

**Optimized Code**

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

**How to translate?**

- Source code and machine code mismatch
- Goals:
  - source-level expressiveness for task
  - best performance for concrete computation
  - reasonable translation efficiency (< O(n³))
  - 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?

High-level source code

? 

Low-level machine code

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 1

Source code (character stream)

if (b == 0) a = b;

Lexical analysis

Token stream

if ( b == 0 ) a = b ;

Parsing

Abstract syntax tree (AST)

if b == 0 a b ;

Semantic Analysis

Decorated AST

boolean if int int a int b

int b

int 0

int a

int b

lvalue
Compilation in a Nutshell 2

if \( b == 0 \) goto L1 else L2
L1: \( a = b \)
L2:

Simplified Compiler Structure

Source code (character stream)
Token stream
Lexical analysis
Parser
Front end (machine-independent)

Abstract syntax tree
Intermediate code Generation
Intermediate code
Control flow graphs
Code generation
Back end (machine-dependent)

Even bigger picture

Source code
Assembly code
Compiler
Assembler
Object code (machine code + symbol tables)
Linker
Loader
Fully-resolved object code (machine code + symbol tables, relocation info)
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?**

- Aka tokenizing, scanning, lexing
- 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...)
    - \( \langle \text{Id, \"x\"}, \text{\"Main.java\"}, 542 \rangle \)
- Issues:
  - how to specify tokens
  - how to implement tokenizer/lexer