CS [45]12[01]

Introduction to Compilers

Spring 2020

Andrew Myers

Lecture 1: Overview

Fill out this form! **https://forms.gle/C2mFDurAjALxXvXP8**

Course Information

- MWF 2:30–3:20PM in Hollister B14
- Instructor: Andrew Myers
- Teaching Assistants:
  - Kushal Babel, Griffin Berlstein, Daniel Weber, Sam Zhou, Hadi Zayer, Gautam Mekkat, Young Kim, Mark Anastos, Daniel Glus
- Web page: http://www.cs.cornell.edu/courses/cs4120
- Piazza page: https://piazza.com/class/k5eh0c04nur545

Outline

- Introduction to compilers
  - What is the point of a compiler?
  - Why should we learn about them?
  - Anatomy of a compiler
- More administration
- Mingle to recruit group members

What is a compiler?

- Translator between representations of program code
- Typically: high-level source code to machine language (object code)
- Not always:
  - Java compiler: Java to interpretable JVM bytecode
  - Java JIT: bytecode to machine code

Do we need a compiler?

- No. Can run programs with an *interpreter* that simulates execution.
- But: best (non-HW) interpreters are at least 10× slower than compiled code (e.g., Python ~30–50×)

  ⇒ use up >10× more energy,

  generate >10× more heat, CO_2

  ⇒ Facebook compiles PHP to C++

- Run only once ⇒ interpret
- Run many times ⇒ compile.

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

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

Assembly and machine code

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

```
expr:
        push    ebp                     55
        mov     ebp, esp                89 e5
        sub     esp, 4                  83 ec 04
        mov     eax, [ebp+8]            8b 45 08
        mov     edx, eax                89 c2
        imul    edx, [ebp+8]            0f af 55
        mov     eax, [8+ebp]            08
        inc     eax                     8b 45 08
        imul    edx, eax                40
        mov     eax, [ebp+8]            0f af d0
        inc     eax                     8b 45 08
        imul    eax, edx                40
        sal     eax, 2                  0f af c2
        mov     [ebp-4], eax            c1 e0 02
        mov     eax, [ebp-4]            89 45 fc
        leave                           8b 45 fc
        ret                             c9
                                        c3
```

Example (Output assembly code)

Unoptimized Code

```
expr:
        push    ebp
        mov     ebp, esp
        sub     esp, 4
        mov     eax, [ebp+8]
        mov     edx, eax
        imul    edx, [ebp+8]
        mov     eax, [8+ebp]
        inc     eax
        imul    edx, eax
        mov     eax, [ebp+8]
        inc     eax
        imul    eax, edx
        sal     eax, 2
        mov     [ebp-4], eax
        mov     eax, [ebp-4]
        leave
        ret
```

Optimized Code

```
expr:
        push    ebp
        mov     ebp, esp
        mov     edx, [ebp+8]
        mov     edx, eax
        imul    eax, edx
        inc     edx
        imul    eax, edx
        imul    eax, edx
        sal     eax, 2
        leave
        ret
```

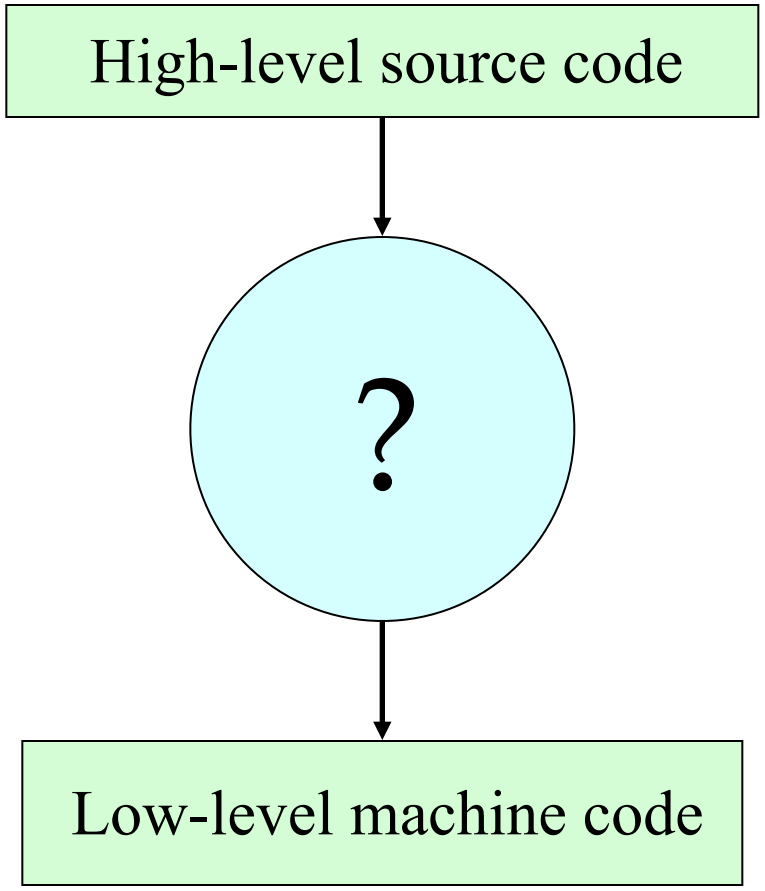
How to translate?

- Source code and machine code mismatch
- Goals:
  - source-level expressiveness for task
  - best performance for concrete computation
  - reasonable translation efficiency
    (< $O(n^3)$ + separate compilation)
  - correct, maintainable compiler code

How to translate correctly?

- Programming languages describe computation precisely (they have *semantics*)
- Therefore: translation can be precisely described (a compiler can be *correct*)
- Correctness is crucial!
  - 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** to generate correct code! [X. *Leroy,* Formal Certification of a *Compiler* Back End, POPL '06]
- This course: a little semantics; more in CS 4110/6110

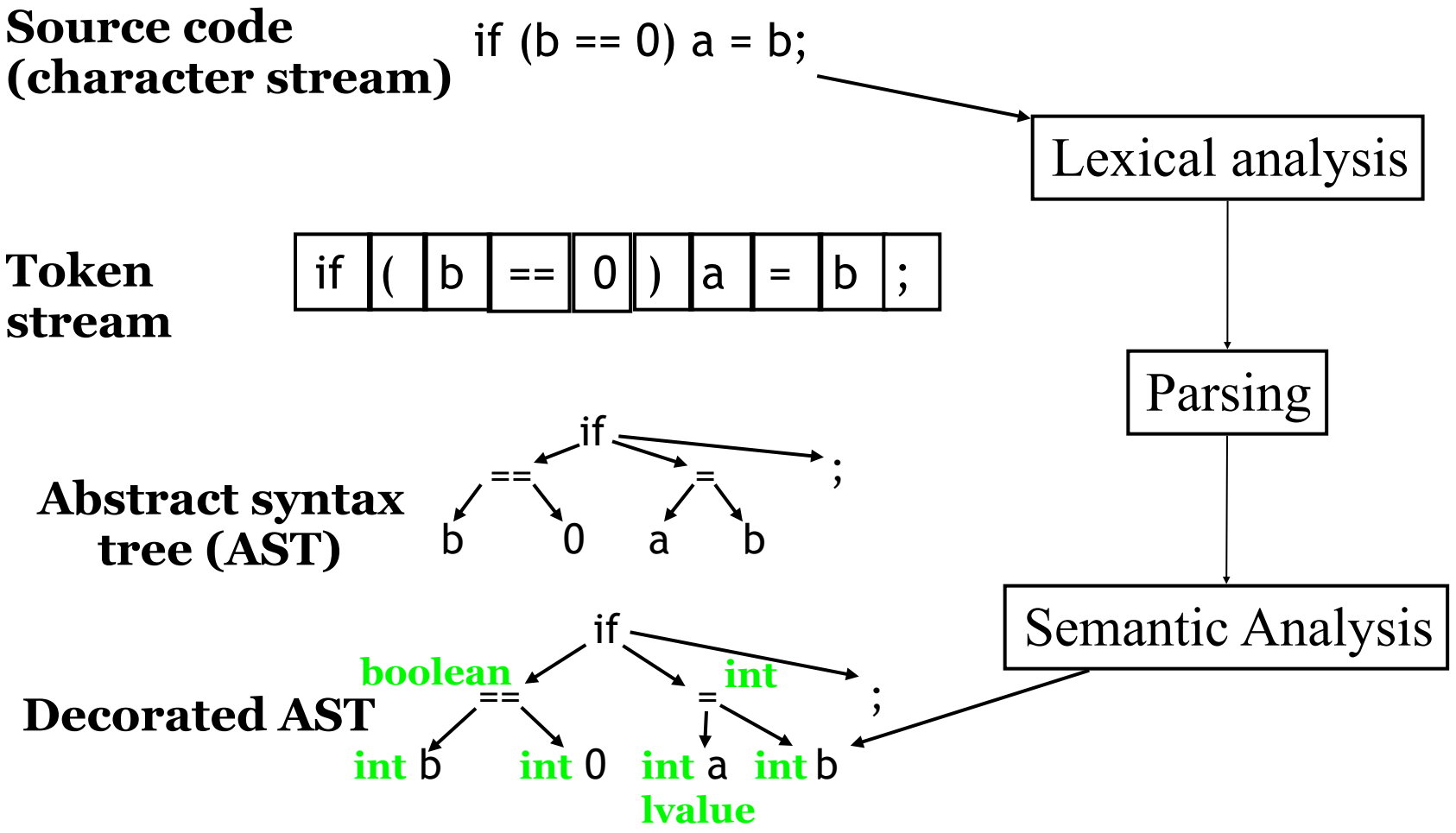
How to translate effectively?



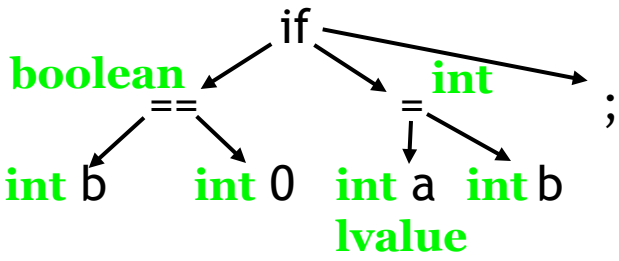
Idea: small easy pieces

- Compiler translates via a series of different **program representations**.
- Intermediate representations designed to support the necessary program manipulations:
  - type checking
  - analysis
  - optimization
  - code generation

Compilation in a Nutshell 1



Compilation in a Nutshell 2



Intermediate Code Generation

```
if b == 0 goto L1 else L2
L1: a = b
L2:
```

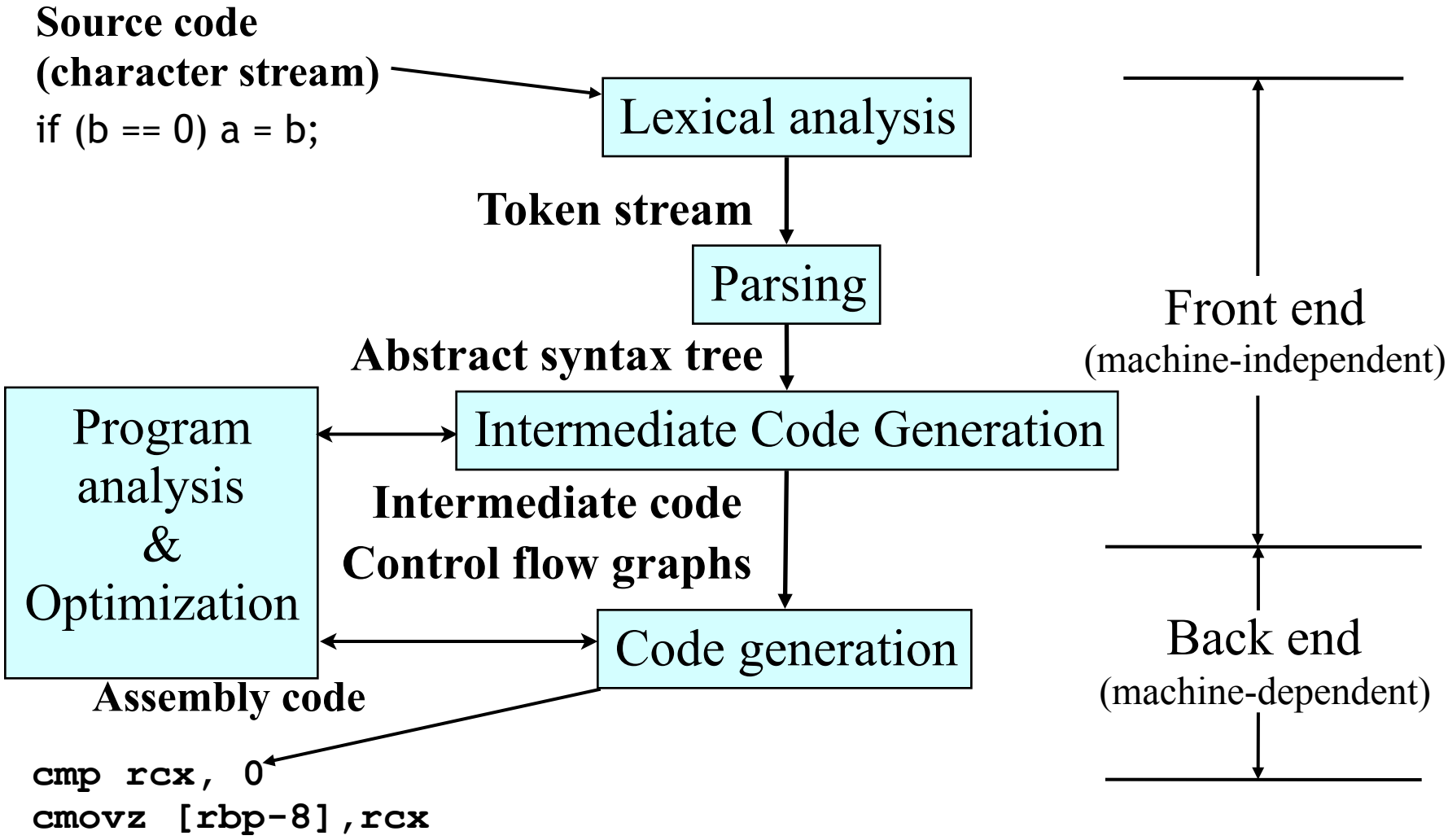
Optimization, Code Generation

```
cmp r_b, 0
jnz L2
L1: mov r_a, r_b
L2:
```

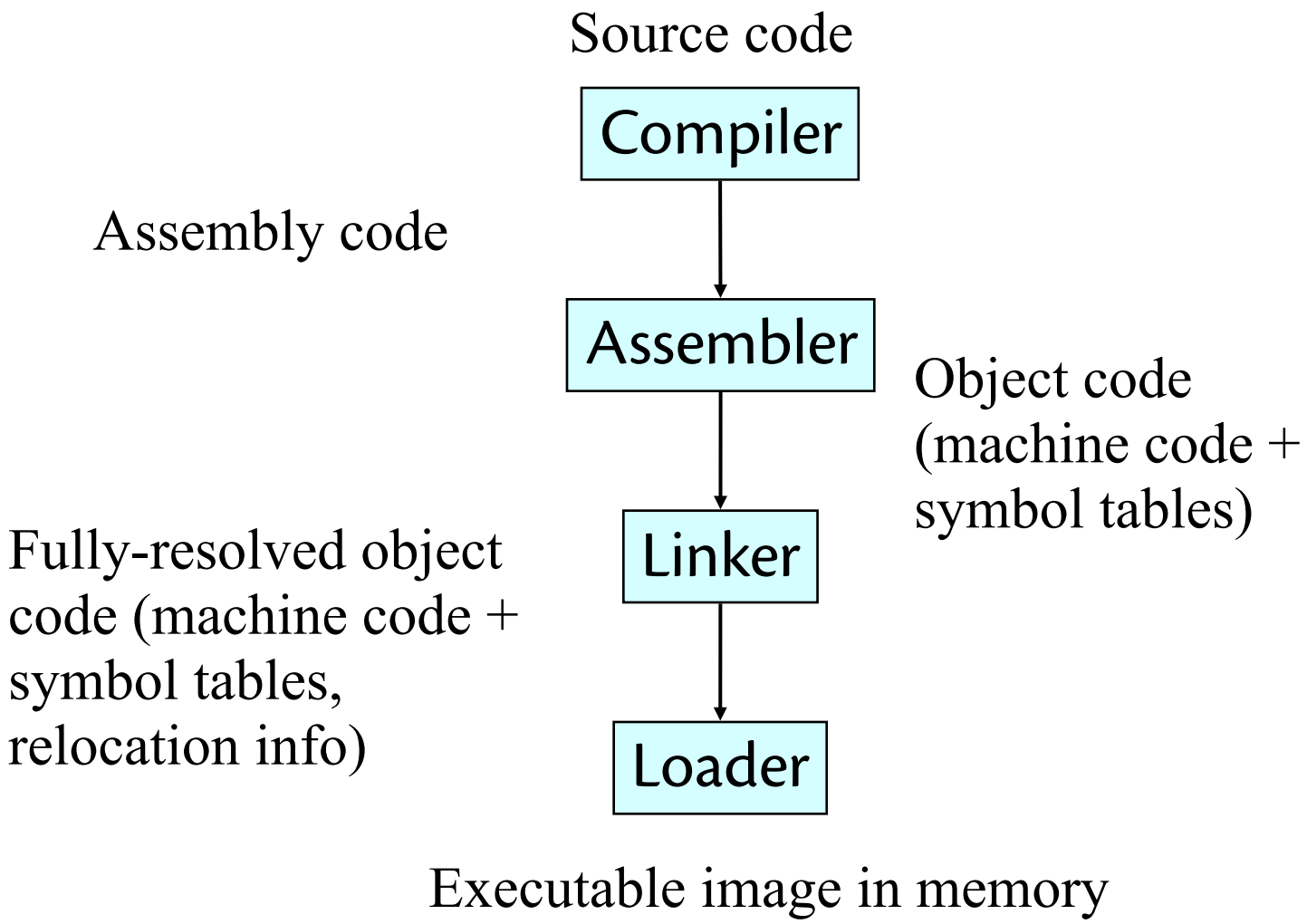
Register allocation, optimization

```
cmp rcx, 0
cmovz [rbp+8],rcx
```

Bigger picture of compiler



Even bigger picture



Schedule

- Detailed schedule on web page+links to slides/notes

Lexical analysis and parsing:	7
Semantic analysis:	4
Intermediate code:	3
Code generation:	2
Prelim 1: March 19	
Program analysis and optimization:	13
Advanced language features:	7
Run-time support:	2
Prelim 2: May 5	

No final exam, final project only

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
  - meets at the same time as CS 4120
- Both parts of course **must** be taken for a **grade**

Textbooks

- Lecture notes provided; no required textbook
- On reserve in Uris Library (Real Soon Now)
  - **Compilers—Principles, Techniques and Tools.** Aho, Lam, Sethi and Ullman (The Dragon Book) (strength: parsing and analysis)
  - **Modern Compiler Implementation in Java.** Andrew Appel. (strength: translation)
  - **Advanced Compiler Design and Implementation.** Steve Muchnick. (strength: analysis and optimization)

Coursework

- Homeworks: 4, 15% total
- Programming Assignments: 7, 50%
  - Building a working compiler
  - 5–10% for each stage
  - Final assignment due in finals week
- Exams: 2 prelims, 35%
  - 15%/20%
  - No exam in finals week

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 with whom you discussed homework

Projects

- Seven programming assignments
- Implementation language: usually Java
  - talk to us if your group wants to use something else (e.g., OCaml, Scala, Haskell, Swift, …)
- 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 assignment "Project"
  - contact us if you are having trouble finding a group.

Assignments

- Due at midnight on due date
- 6 slip days
  - Extensions granted in some unusual circumstances but must be approved 2 days in advance
- Projects submitted, solutions available via CMSX (cmsx.cs.cornell.edu)

Why take this course?

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

Student comments

"This class overall taught me how to be a much much much better programmer."

"Writing a compiler was the most fulfilling and educational programming project I have done."

Mingle!

- Feel free to recruit people who haven't signed up yet…