CS [45]12[01]

Introduction to Compilers
Spring 2022
Andrew Myers

Lecture 1: Overview

Course Information

- MWF 2:40–3:30 in Gates G01 (but initially on Zoom)
- Instructor: Andrew Myers
- Teaching Assistants:
  - Priya Srikumar, Sam Parkinson, Charles Sherk, Shiyuan Huang, Kaya Ito Alpturer, Susan Garry, Prarthi Jain, Qian Meng
- Web page: http://courses.cs.cornell.edu/cs4120

Outline

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

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  rbp                 55
   mov   rbp, rsp            48 89 e5
   lea   eax, [rdi + 1]      8d 47 01
   imul  eax, edi            0f af c7
   imul  eax, eax            0f af c0
   shl   eax, 2              c1 e0 02
   pop   rbp                 5d
   ret                       c3
```

Example (Output assembly code)

Unoptimized Code

```
_expr:
        push    rbp
        mov     rbp, rsp
        mov     [rbp - 4], edi
        mov     eax, [rbp - 4]
        shl     eax, 2
        imul    eax, [rbp - 4]
        mov     ecx, [rbp - 4]
        add     ecx, 1
        imul    eax, ecx
        mov     ecx, [rbp - 4]
        add     ecx, 1
        imul    eax, ecx
        mov     [rbp - 8], eax
        mov     eax, [rbp - 8]
        pop     rbp
        ret
```

Optimized Code

```
_expr:
    push  rbp
    mov   rbp, rsp
    lea   eax, [rdi + 1]
    imul  eax, edi
    imul  eax, eax
    shl   eax, 2
    pop   rbp
    ret
```

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

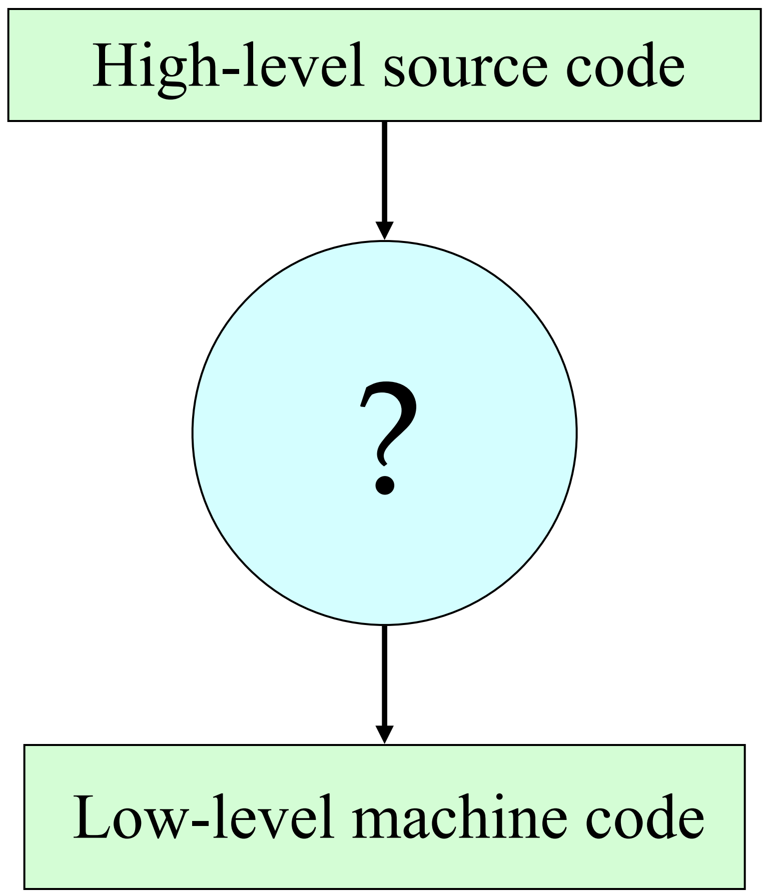
How to translate?

- 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

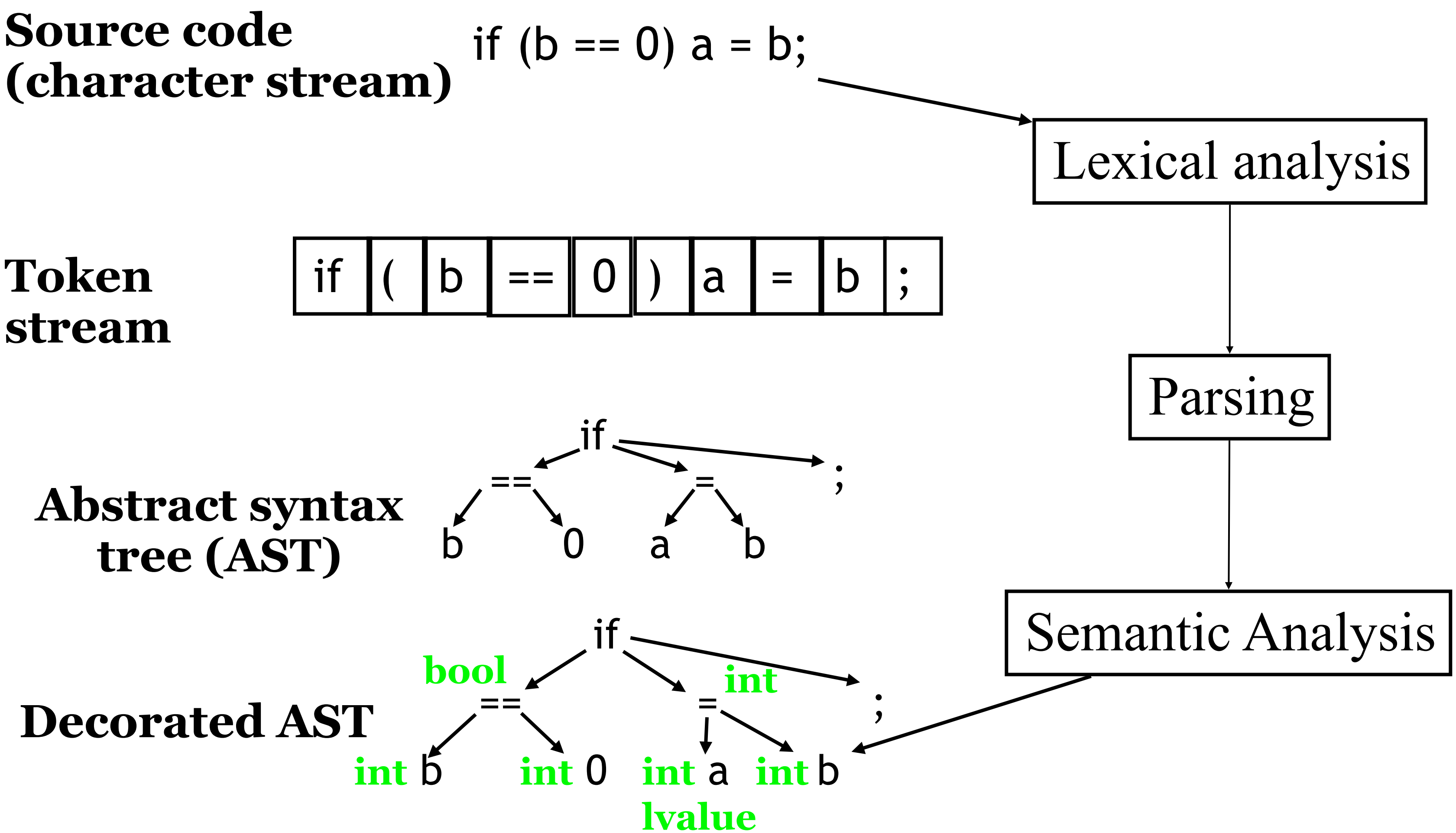
How to translate effectively?



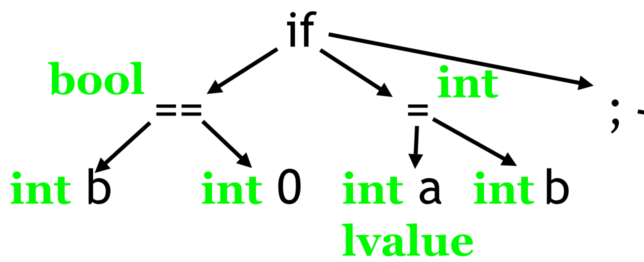
Idea: small easy pieces

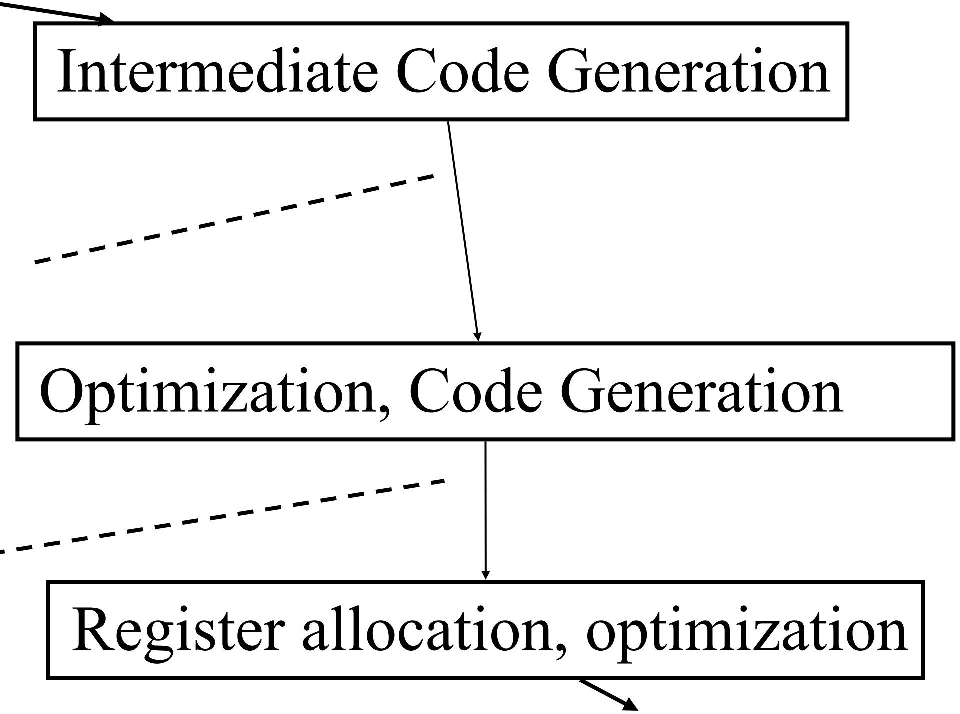
- Compiler translates via a series of different **program representations**.
- Intermediate representations designed to support the necessary program manipulations:
  - type checking
  - analysis
  - optimization
  - code generation

Compilation in a Nutshell 1



Compilation in a Nutshell 2



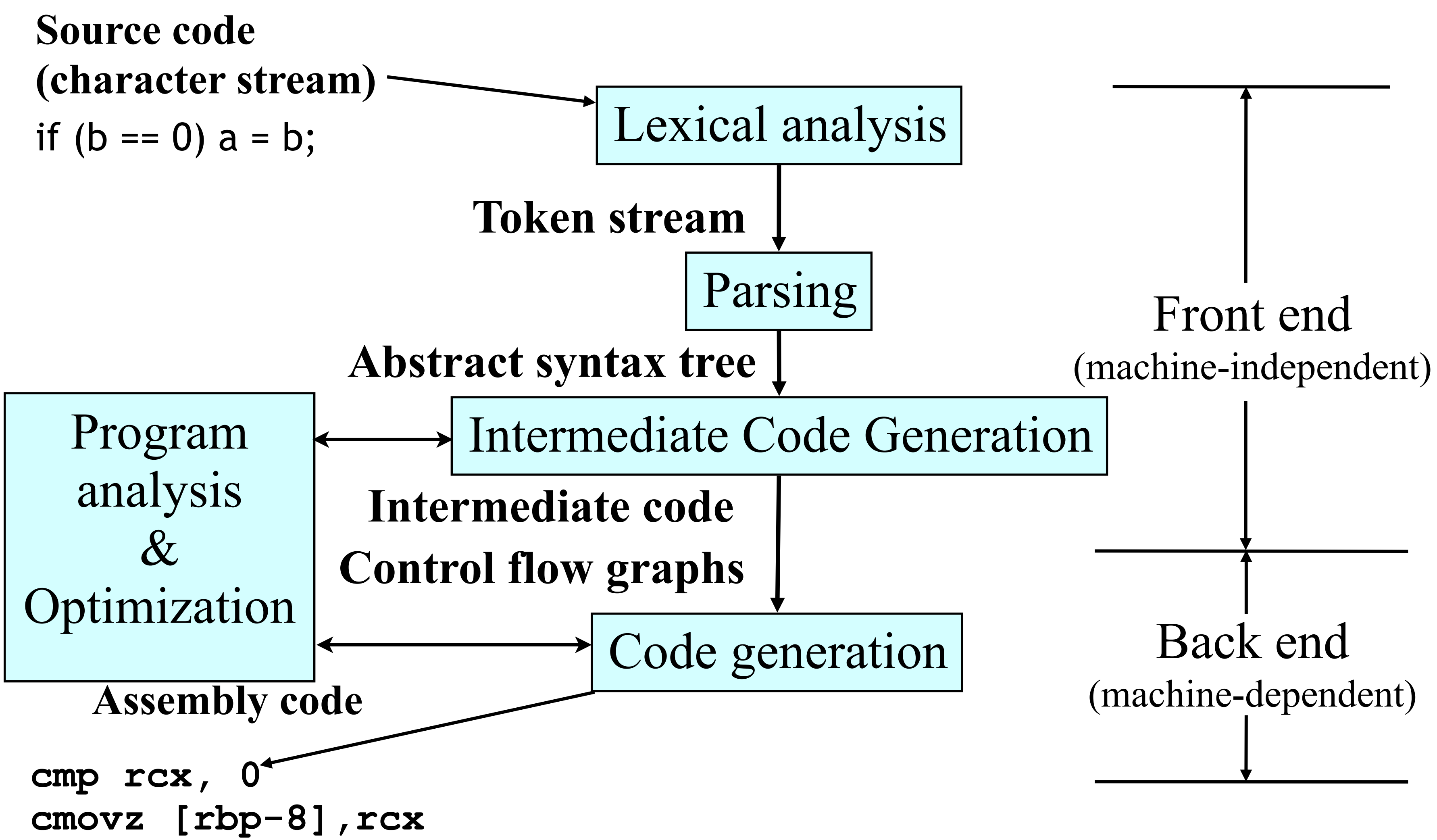



```
if b == 0 goto L1 else L2
L1: a = b
L2:
```

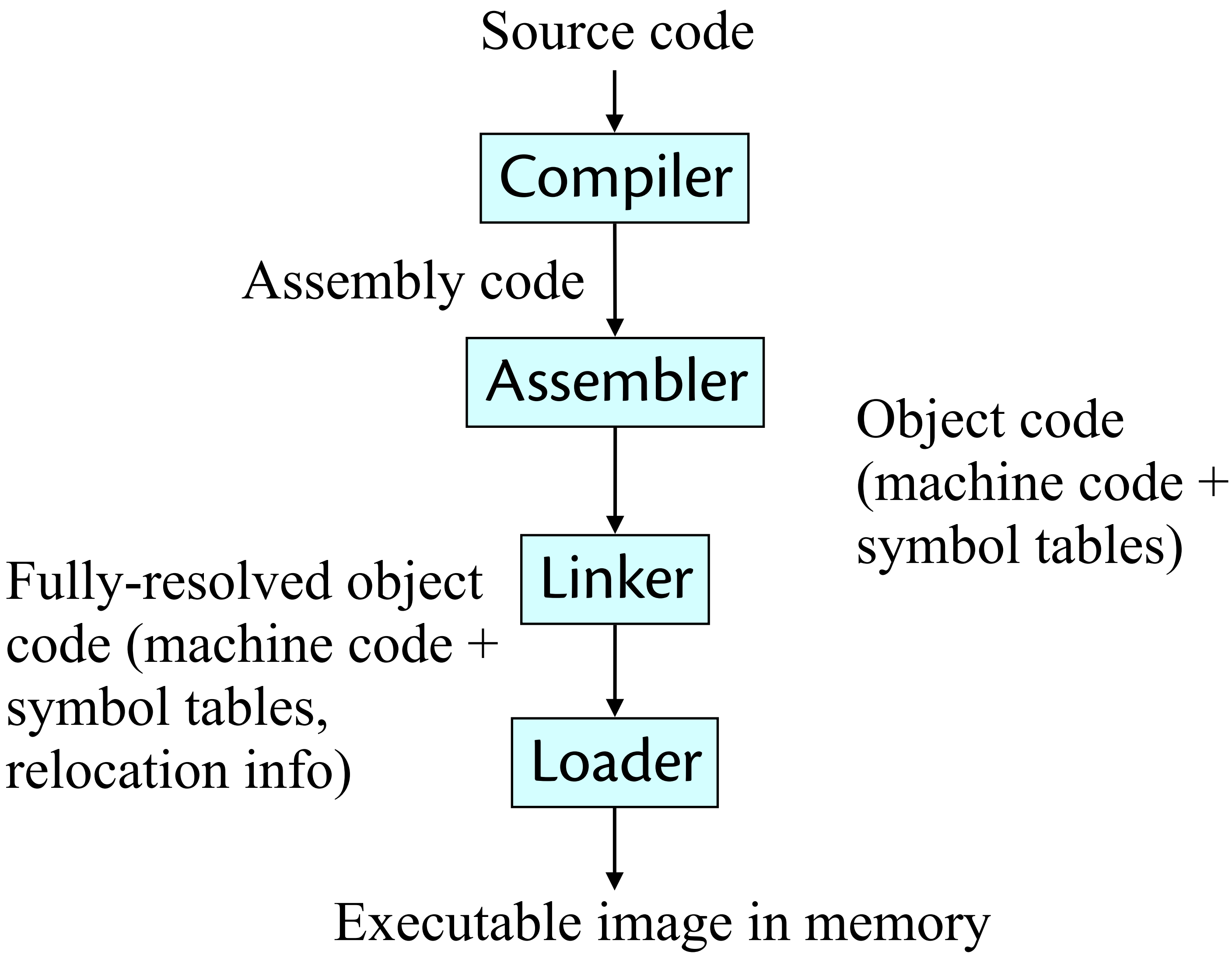
```
cmp r_b, 0
jnz L2
L1: mov r_a, r_b
L2:
```

```
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:	4
Code generation:	2
Prelim 1: March 17–19	(any 24 of 48 hours)
Program analysis and optimization:	12
Advanced language features:	7
Run-time support:	2
Prelim 2: May 3–5	(any 24 of 48 hours)

No final exam, final project only (due date TBD)

4 = 5 & 0 = 1

- CS 4120 and 5120 are really the same course
  - same lectures
  - mostly same assignments, some extra work in 5120
  - 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, 20% total
- Programming Assignments: 6, 45%
  - 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

- Six programming assignments
- Implementation language: usually Java
  - talk to us if your group wants to use something else (e.g., OCaml, Scala, Kotlin, TypeScript, Swift, Rust, Haskell, …)
- Groups of 3–4 students
  - same group for entire class (ordinarily)
  - same grade for all (ordinarily, but peer review will be used)
  - 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.
- create your group on CMSX for assignment “Project”
  - Use the discussion forum to find group members with a matching working style
  - contact us if you are having trouble finding a group.

Assignments

- Due at midnight on due date
  - 10% deduction per late day
- 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."

How to make your compilers project harder

- Some tips for an added challenge

1. The Scapegoat.
2. The Lone Wolf.
3. The Round Robin.
4. The Schism.
5. The Borg.
6. The Blitz.
7. The Stoic.
8. The Blank Slate.
9. The Time Machine.
10. The Combo.