CS (4|5)12(0|1)

Introduction to Compilers

Spring 2016

Andrew Myers

Lecture 1: Overview

Outline

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

Course Information

- MWF 3:35–4:25PM in Gates G01
- Instructor: Andrew Myers
- Teaching Assistants: Owen Arden, Chin Isradisaikul, Steffen Smolka
- Web page: http://www.cs.cornell.edu/courses/cs4120
- See also Piazza page

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 (Real Soon Now)
  - **Compilers—Principles, Techniques and Tools.** Aho, Lam, Sethi and Ullman (The Dragon Book) (strength: parsing)
  - **Modern Compiler Implementation in Java.** Andrew Appel. (strength: translation)
  - **Advanced Compiler Design and Implementation.** Steve Muchnick. (strength: optimization)

Work

- Homeworks: 4, 20% total
  - 5/5/5/5
- Programming Assignments: 7, 50%
  - Building a working compiler
  - 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, Scala)
- 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
- 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?

- 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 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 interpreters are 10–50× slower than compiled code
  ⇒ use up 10–50× more energy, generate 10–50× more heat.
  ⇒ 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;
}
```

Machine code

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

```
expr:
        pushl   %ebp                    55
        movl    %esp, %ebp              89 e5
        subl    $4, %esp                83 ec 04
        movl    8(%ebp), %eax           8b 45 08
        movl    %eax, %edx              89 c2
        imull   8(%ebp), %edx           0f af 55 08
        movl    8(%ebp), %eax           8b 45 08
        incl    %eax                    40
        imull   %eax, %edx              0f af d0
        movl    8(%ebp), %eax           8b 45 08
        incl    %eax                    40
        imull   %edx, %eax              0f af c2
        sall    $2, %eax                c1 e0 02
        movl    %eax, -4(%ebp)          89 45 fc
        movl    -4(%ebp), %eax          8b 45 fc
        leave                           c9
        ret                             c3
```

Example (Output assembly code)

Unoptimized Code

```
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
```

Optimized Code

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

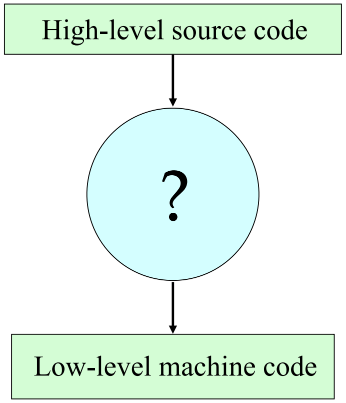
How to translate?

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

How to translate correctly?

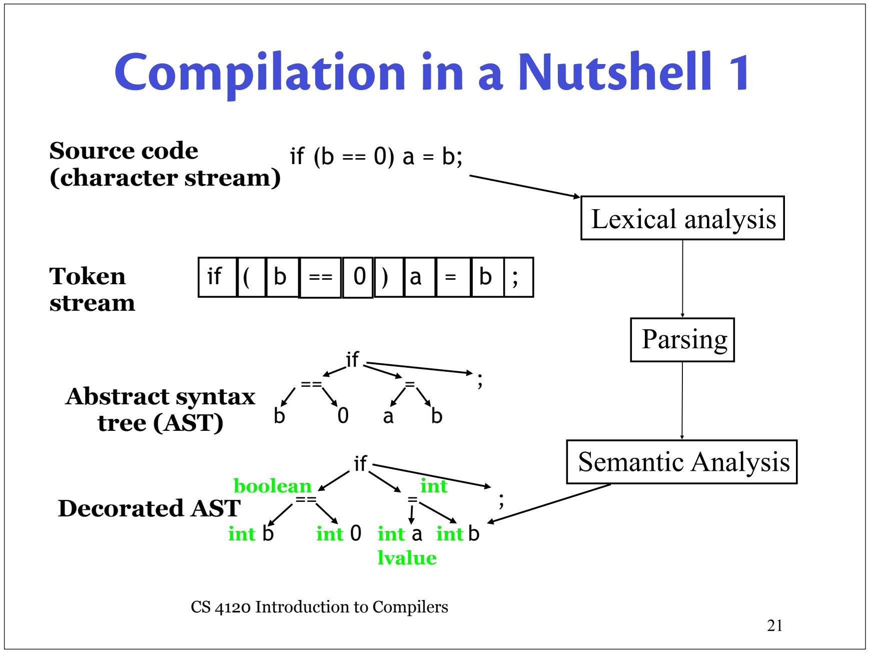
- Programming languages describe computation precisely (semantics)
- Therefore: translation can be precisely described (a compiler can be correct)
- Correctness is very important!
  - hard to debug programs with broken compiler…
  - non-trivial: programming languages are expressive
  - implications for development cost, security
  - this course: techniques for building correct compilers
  - some compilers have been **proven** correct!
    [see X. *Leroy*, Formal Certification of a *Compiler* Back End, POPL '06]

How to translate effectively?



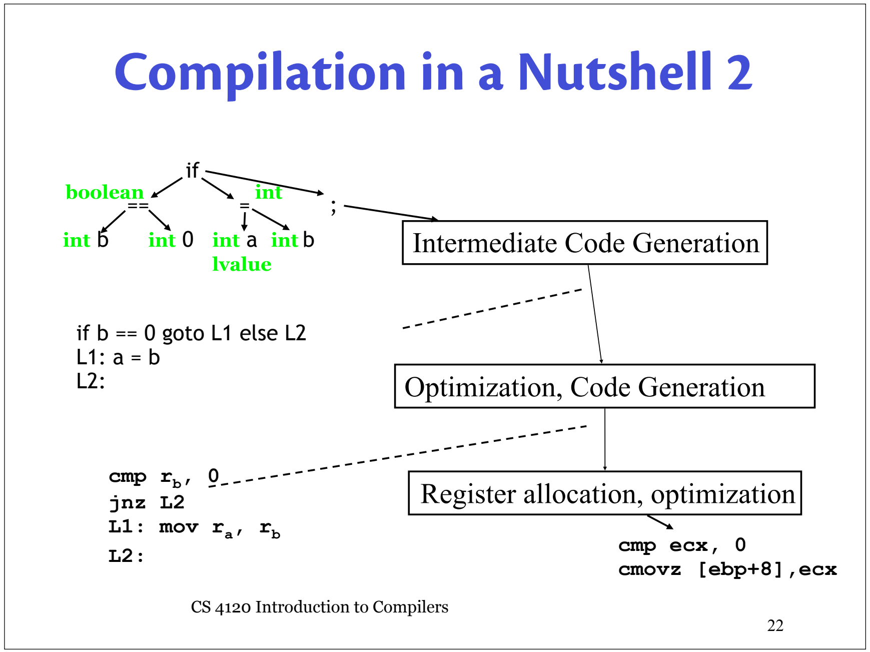
Idea: translate in steps

- 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 1

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



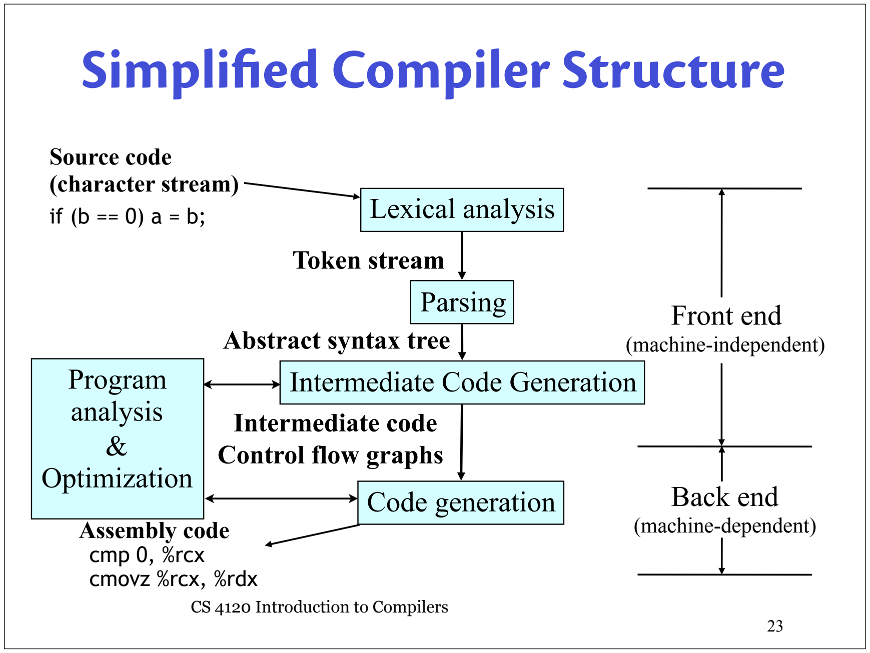
Compilation in a Nutshell 2

if
boolean
==
int b
int 0
int
=
int a
int b
lvalue
;
Intermediate Code Generation
Optimization, Code Generation
Register allocation, optimization

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

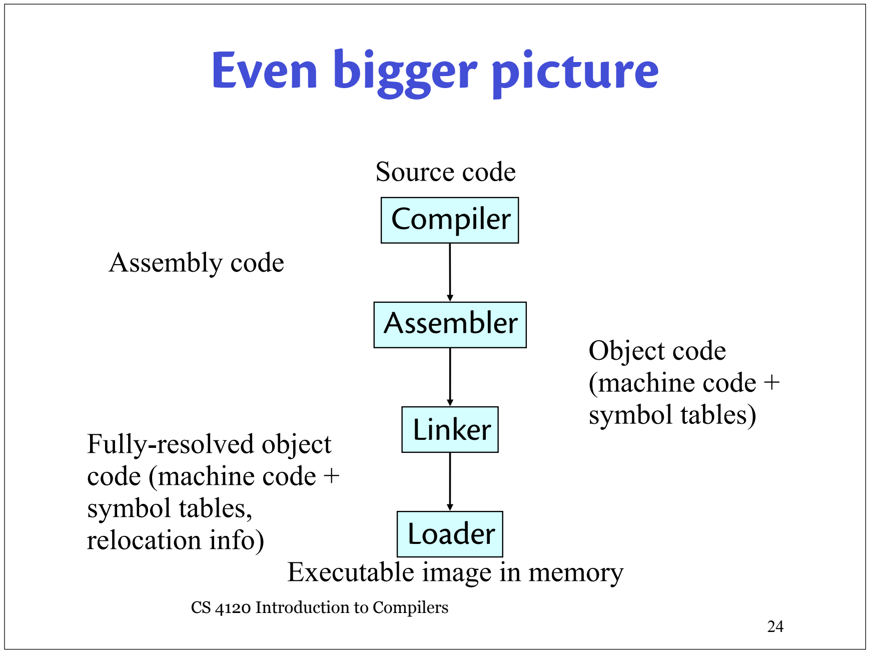


Simplified Compiler Structure

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

Assembly code

```
cmp 0, %rcx
cmovz %rcx, %rdx
```



Even bigger picture

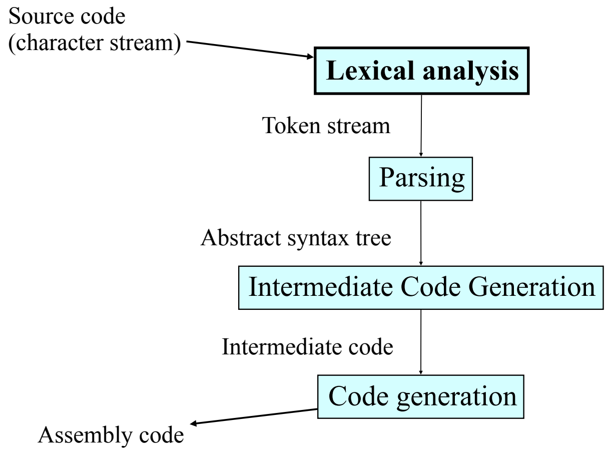
Source code
Compiler
Assembly code
Assembler
Object code (machine code + symbol tables)
Linker
Fully-resolved object code (machine code + symbol tables, relocation info)
Loader
Executable image in memory

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
Run-time, link-time support:	2
Advanced topics:	7
Prelim #2	

First step: Lexical Analysis



Source code (character stream)

Lexical analysis

Token stream

Parsing

Abstract syntax tree

Intermediate Code Generation

Intermediate code

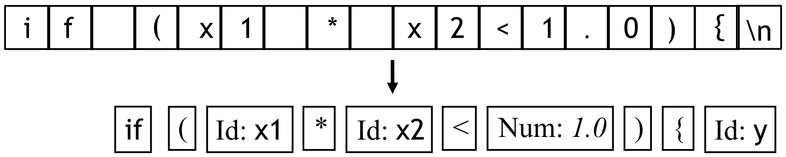
Code generation

Assembly code

What is Lexical Analysis?

- Aka tokenizing, scanning, lexing
- Converts character stream to token stream of pairs ⟨*token type, attribute*⟩

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



i f (x 1 * x 2 < 1 . 0) { \n

if (Id: x1 * Id: x2 < Num: *1.0*) { Id: y

Token stream

- Gets rid of whitespace, comments
- Only ⟨ Token type, attribute ⟩:
- ⟨ Id, “x” ⟩, ⟨ Float, *1.0e0* ⟩
- Token location preserved for debugging, run-time/compile-time error messages (source file, line number, character posn...)
- ⟨ Id, “x”, “Main.java”, 542⟩

- Issues:
  - how to specify tokens
  - how to implement tokenizer/lexer (lexer generator!)