CS 4120 / 4121
CS 5120 / 5121

Introduction to Compilers
Fall 2013
Ross (Tate)

Lecture 1: Overview

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

Course Information

- MWF 1:30- 2:15PM in Olin 245
- Instructor: Ross Tate
- Teaching Assistants:
  Fabian Mühlböck, Sam Hopkins, Lee Gao
- E-mail: cs4120-l@cs.cornell.edu
- Web page:
  http://www.cs.cornell.edu/courses/cs4120
- Newsgroup:
  https://piazza.com/class#fall2013/cs4120

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:
  - **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: 5, 35% total
  - 6-8% each
- Programming Assignments: 7, 65%
  - 7-13% each
- Exams: None

Academic integrity

- Taken seriously.
- Do your own (or your group's) work.
- Report who you discussed homework with (whether student in class or not).
- Feel free to share test inputs on Piazza.

Homeworks

- Three assignments in first half of course; two homeworks 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
  - or anything you can compile to an executable jar
- 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
- Old: high-level source code to machine language (object code)
- Modern:
  - High-level to mid-level (Java to bytecode)
  - Mid-level to low-level (bytecode to x86)

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       ret                 c9
                                        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?

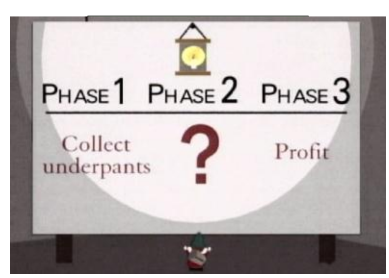
- Source-code and machine-code mismatch

- Goals:
  - source-level expressiveness for task
  - best performance for concrete computation
  - reasonable translation efficiency (< $O(n^3)$)
  - 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]

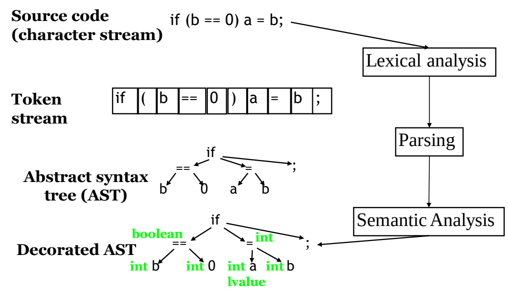
How to translate effectively?



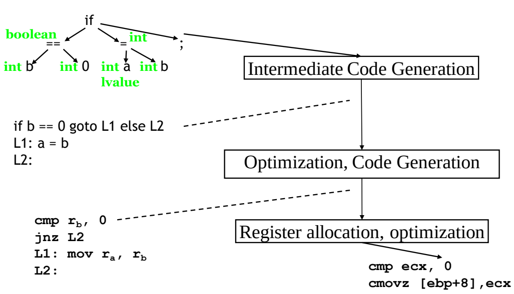
Idea: translate in steps

- Compiler uses a series of different **program representations**.
- Intermediate representations that are good for program manipulations of various kinds (analysis, optimization, code generation).

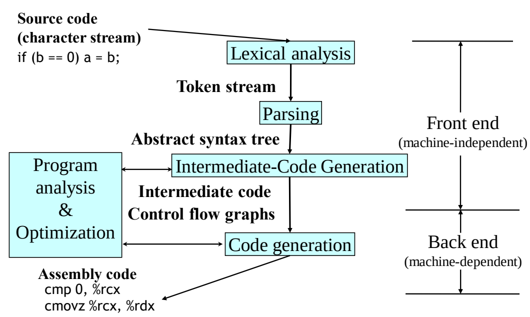
Compilation in a Nutshell 1



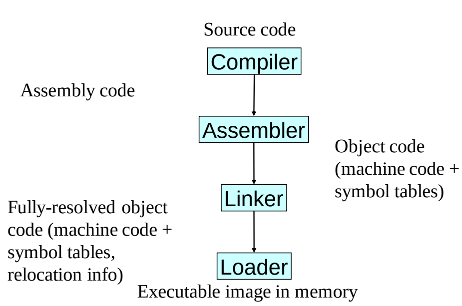
Compilation in a Nutshell 2



Simplified Compiler Structure



Even bigger picture



CubeX

```
interface List<E> {
      fun elements() : Iterable<E>;
}
class Nil() extends List<Nothing> {
      fun elements() : Iterable<Nothing>
            { return []; }
}
class Cons<E>(E head, List<E> tail)
            extends List<Nothing> {
      elems := [head] ++ tail.elements();
      fun elements() : Iterable<E> {return elems;}
}
```

CubeX

- Object-Oriented
- Generics
- Pure except for non-termination
- Memory managed

Project

- Compile CubeX to C
  - With optimizations
- Choice of Extension, such as
  - Compile to x86
  - Variance and inference
  - Recursive inheritance and F-bounded polymorphism
  - Continuation-based custom iterables
  - Iterable comprehensions and liftings