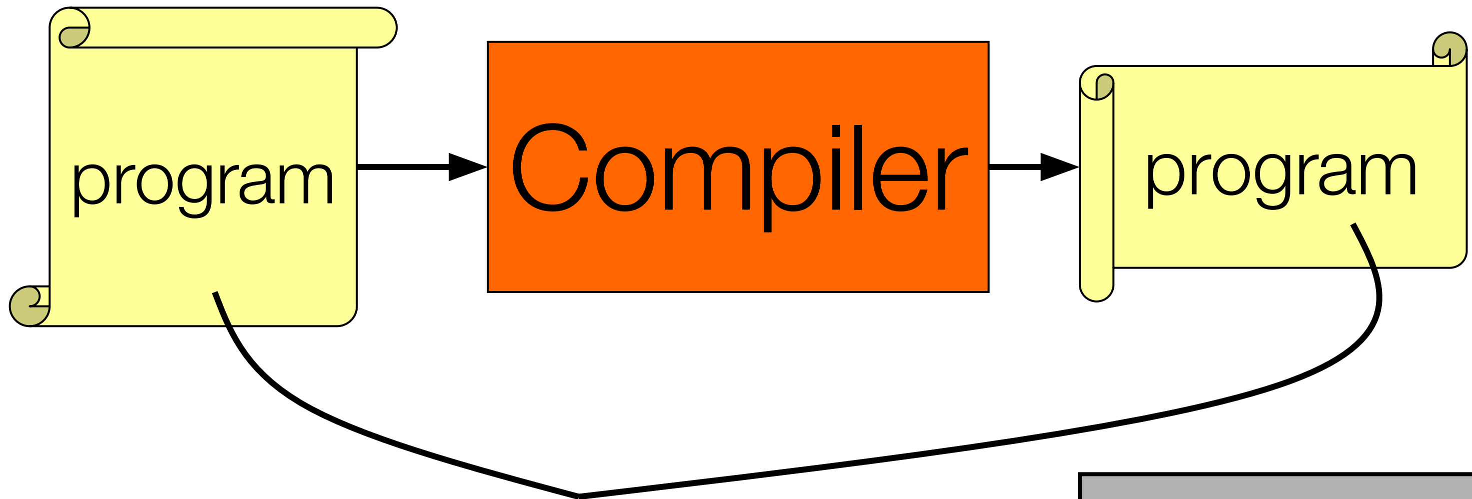
15-411/15-611 Compiler Design

Spring 2025 with Seth Copen Goldstein and Ben Titzer

http://www.cs.cmu.edu/~411

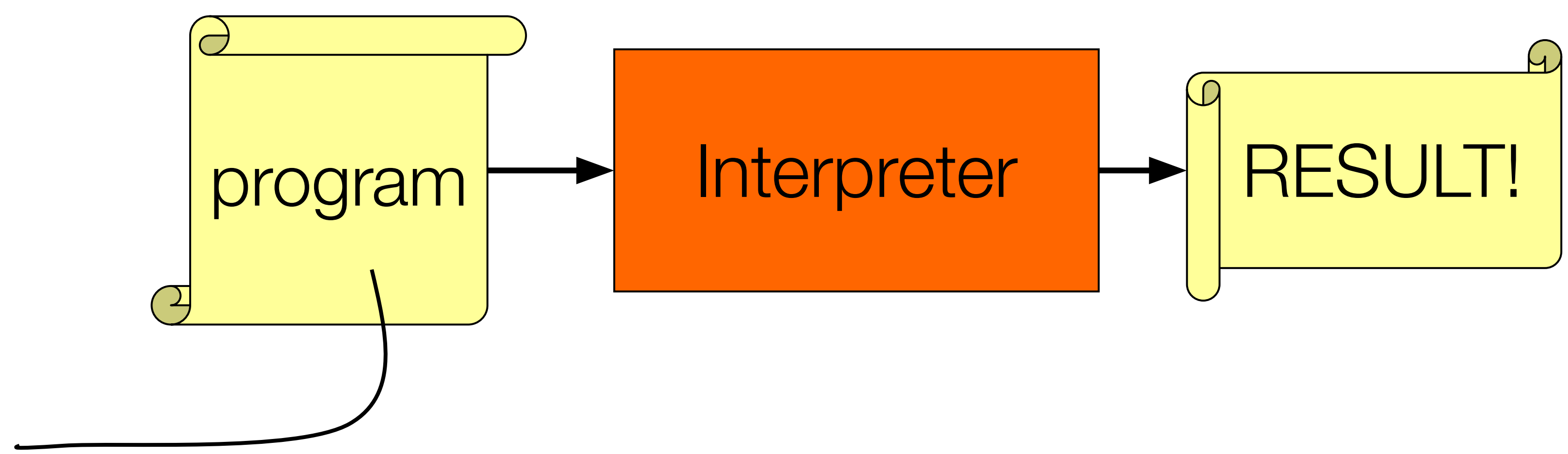
Compilers at 60K

What is a Compiler?



executable form

What did I forget?



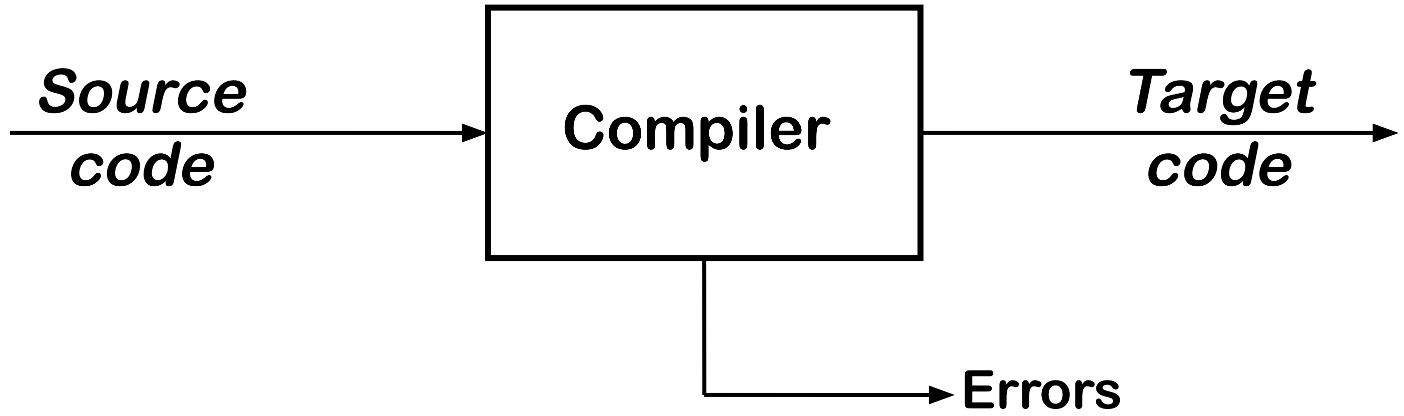
executable form

“A” Compiler is a misnomer

- Multiple sources compiled into .o files
- Linker combines .o files into .exe file
- Loader combines .exe file (with .so) into a runnable application

- But, we will mostly ignore this in class.

Better View of a Compiler



Implications

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with target on format for object code

Big step up from assembly language—use higher level notations

Executors

- Compilers transform specifications
- Interpreters execute specifications
  - (without generating new target code)
- E.g.: C++ is usually compiled
  Python is usually interpreted
  Java/JavaScript are JIT-compiled
- Many common issues
- 411 mainly focuses on compilers.

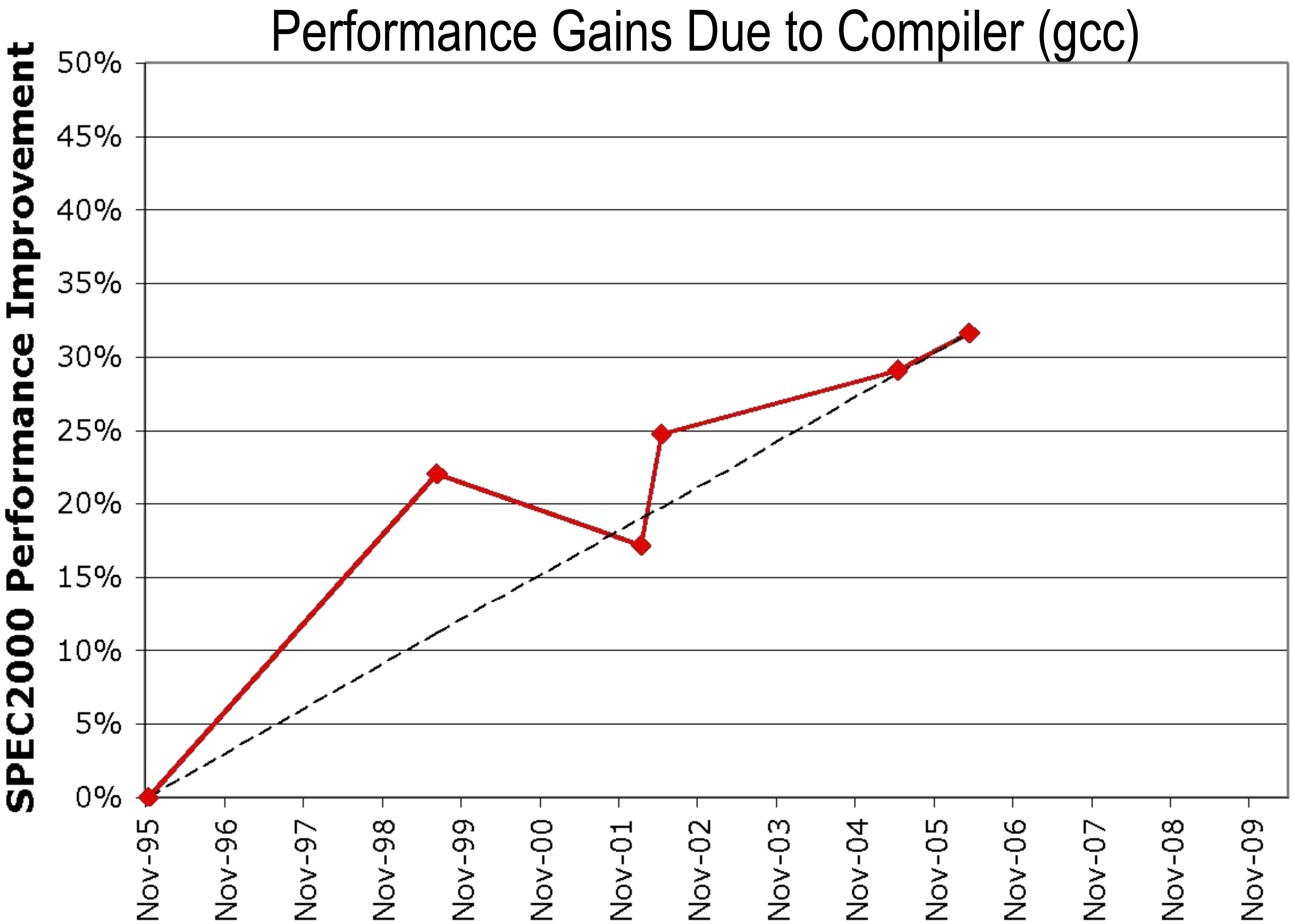
Why take this class?

- Compilers design and construction combines:
  - theory
  - algorithms
  - AI
  - systems
  - architecture
  - software engineering

Compilers Are Everywhere

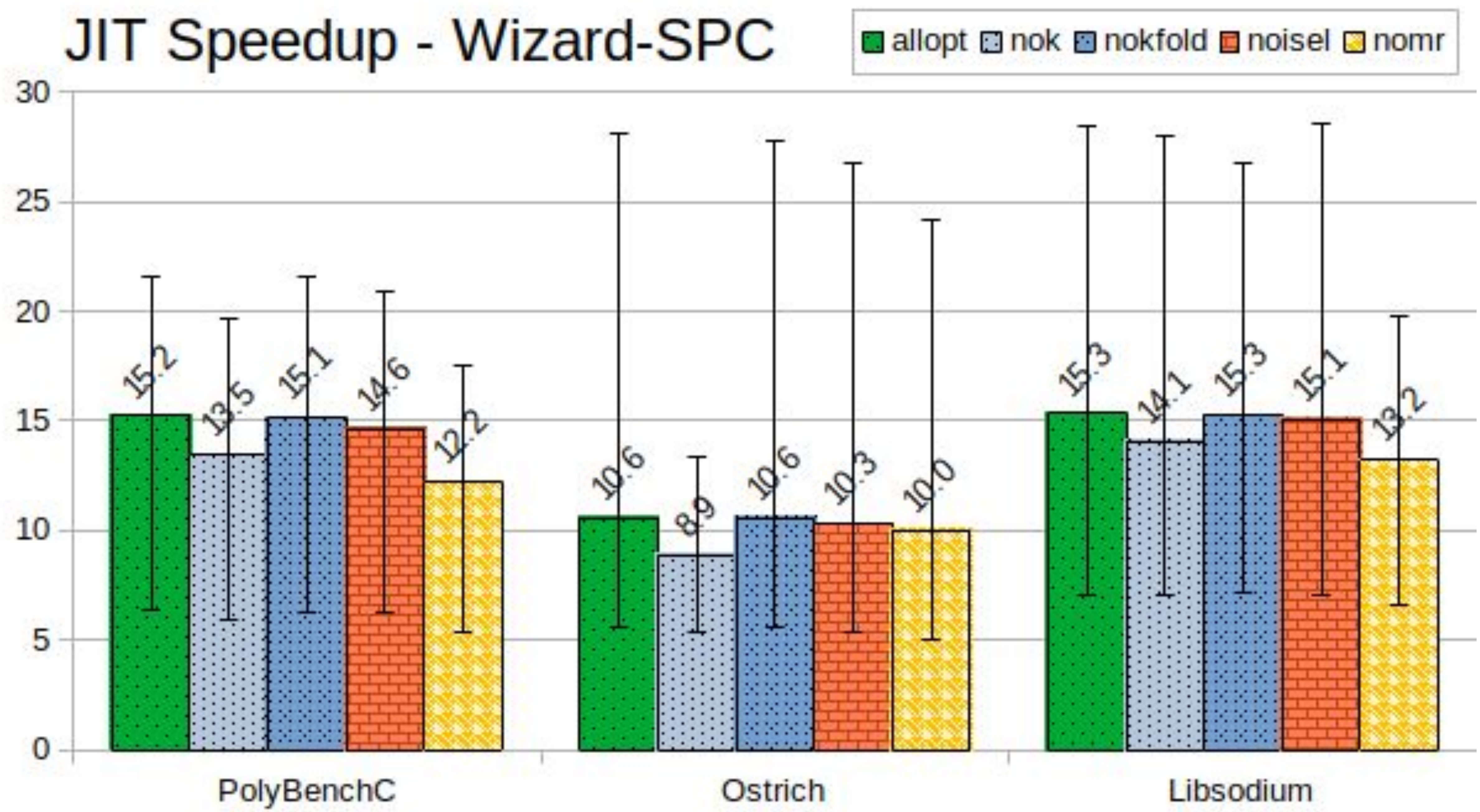
- FTP daemon
- Web browsers
- perl, sed, awk, emacs, bc
- excel, tex
- web servers (e.g., asp)
- databases (query opt)
- virtual machines
- ?

Compilers are Essential



Compilers are Essential

Virtual machines employ JITs for dramatic speedups



Compilers Are Fun

- Many very hard problems
  - Many (if not most) are NP-hard
  - So, what to do?
- Applies theory and practice
- Modern architectures depend on compilers: Compiler writers drive architectures!
- You can see the results

What makes a good compiler?

- Correctness
- Performance of translated program
  - *Predictably* small and fast code
- Scalability of compiler
  - Fast compile time
  - Separate (incremental, parallel) compilation
- Easy to modify
- Aids programmer
  - good compile time error messages
  - support for debugger

Compilers at 30K

A Simple Example

```
x := a * 2 + b * (x * 3)
```

- What does this mean? Is it valid?
- How do we determine its meaning:
  - break into words
  - convert words to sentences
  - interpret the meaning of the sentences

Lexical Analysis

```
x := a * 2 + b * (x * 3)
```

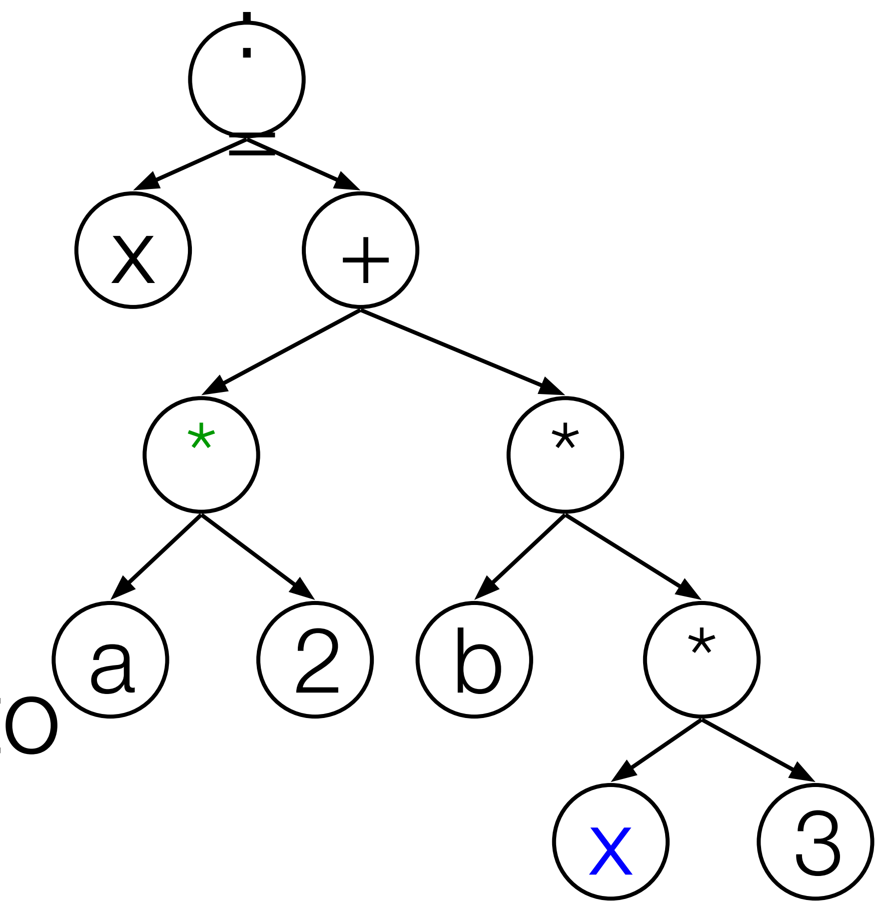
```
id<x> assign id<a> times int<2> plus id<b>
times lparen id<x> times int<3> rparen
```

- Group characters into tokens
- Eliminate unnecessary characters from the input stream
- Use **regular expressions** to specify and DFAs to implement.
- E.g., lex

Syntactic Analysis

```
x := a * 2 + b * (x * 3)
```

```
id<x> assign id<a> times int<2> plus id<b>
times lparen id<x> times int<3> rparen
```



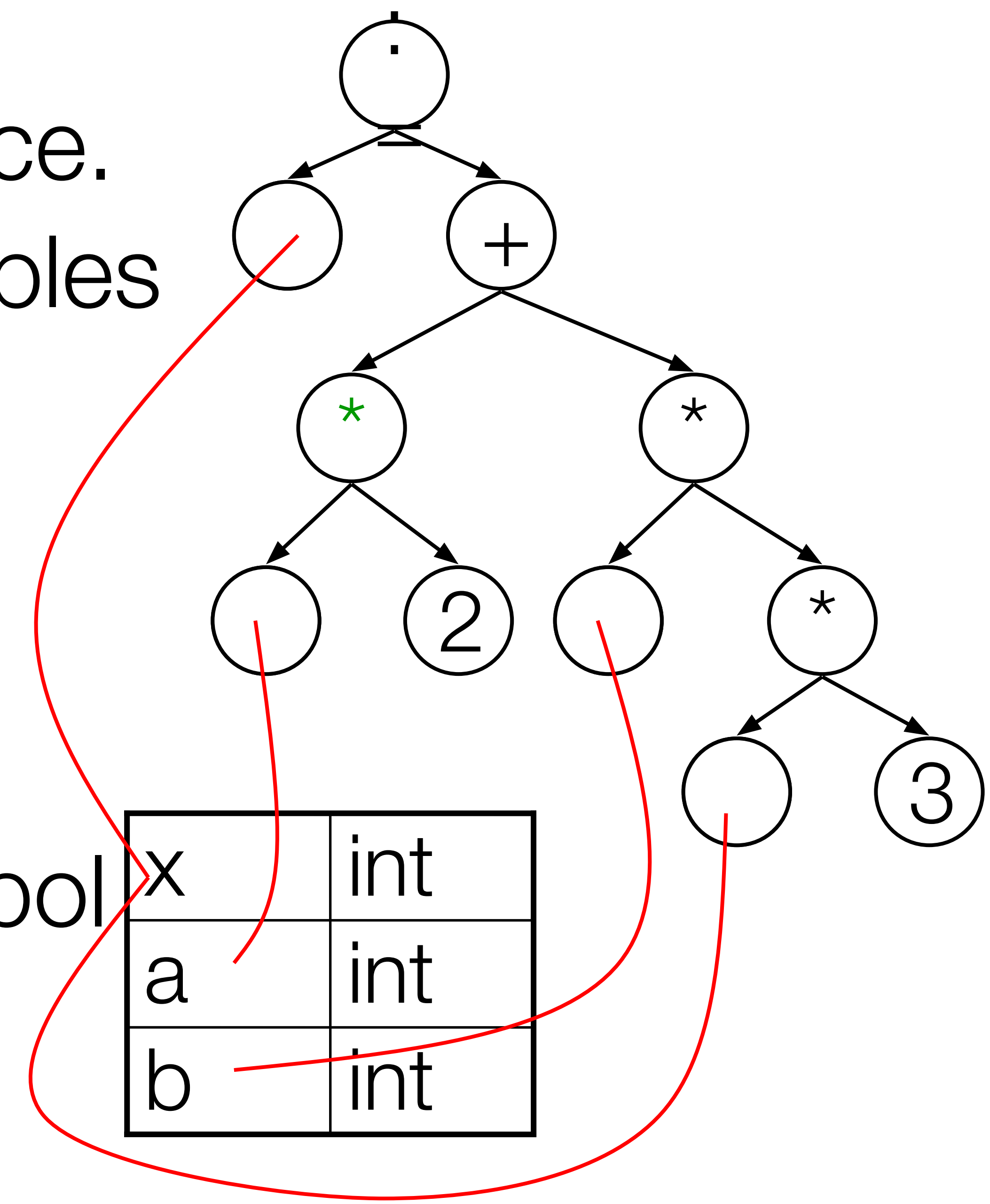
- Group tokens into sentences
- Eliminate unnecessary tokens from the input stream
- Use **context-free grammars** to specify and push down automata to implement
- E.g., bison

Semantic Analysis

```
x := a * 2 + b * (x * 3)
```

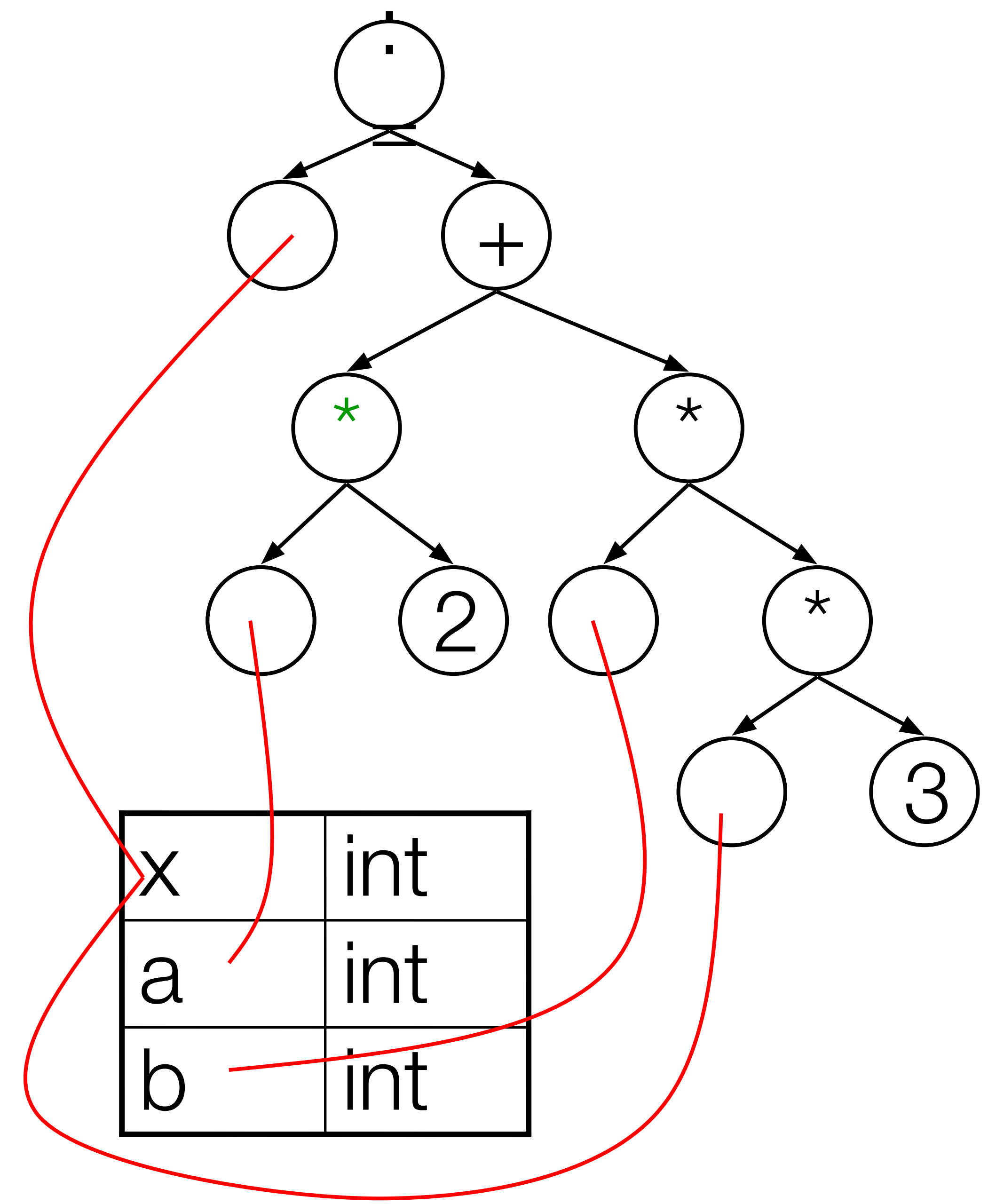
```
id<x> assign id<a> times int<2> plus id<b>
times lparen id<x> times int<3> rparen
```

- Determines meaning of sentence.
- What are the types of the variables (x, a, b)?
- Constants (2, 3)?
- Operators (*, +)
- Is it legal to read and write x?
- Use attributed grammars, symbol tables, …



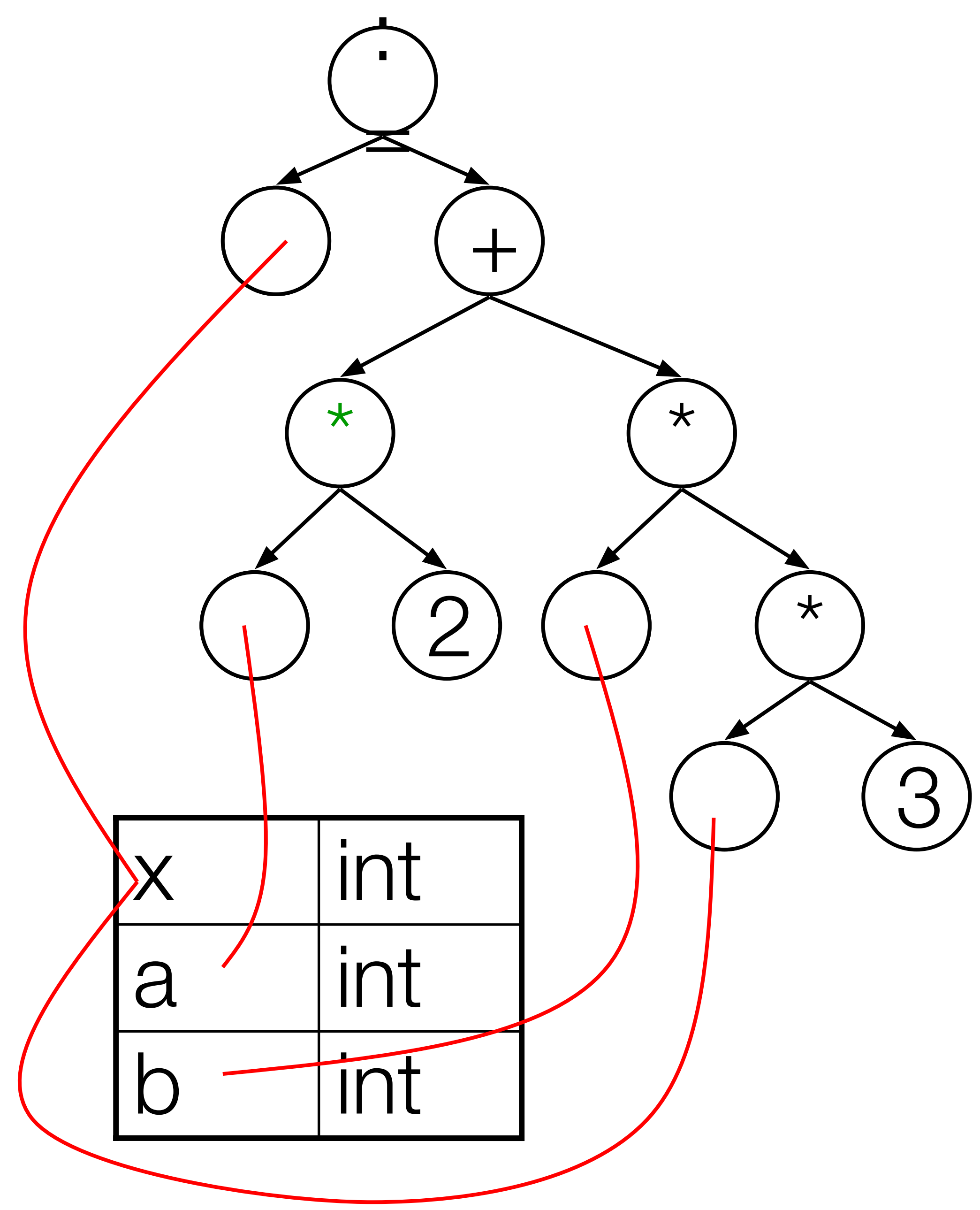
Translation

- Interface between front-end and back-end
- Many different types of IRs
  - Hierarchical
  - Linear
  - Tree based
  - Triple based



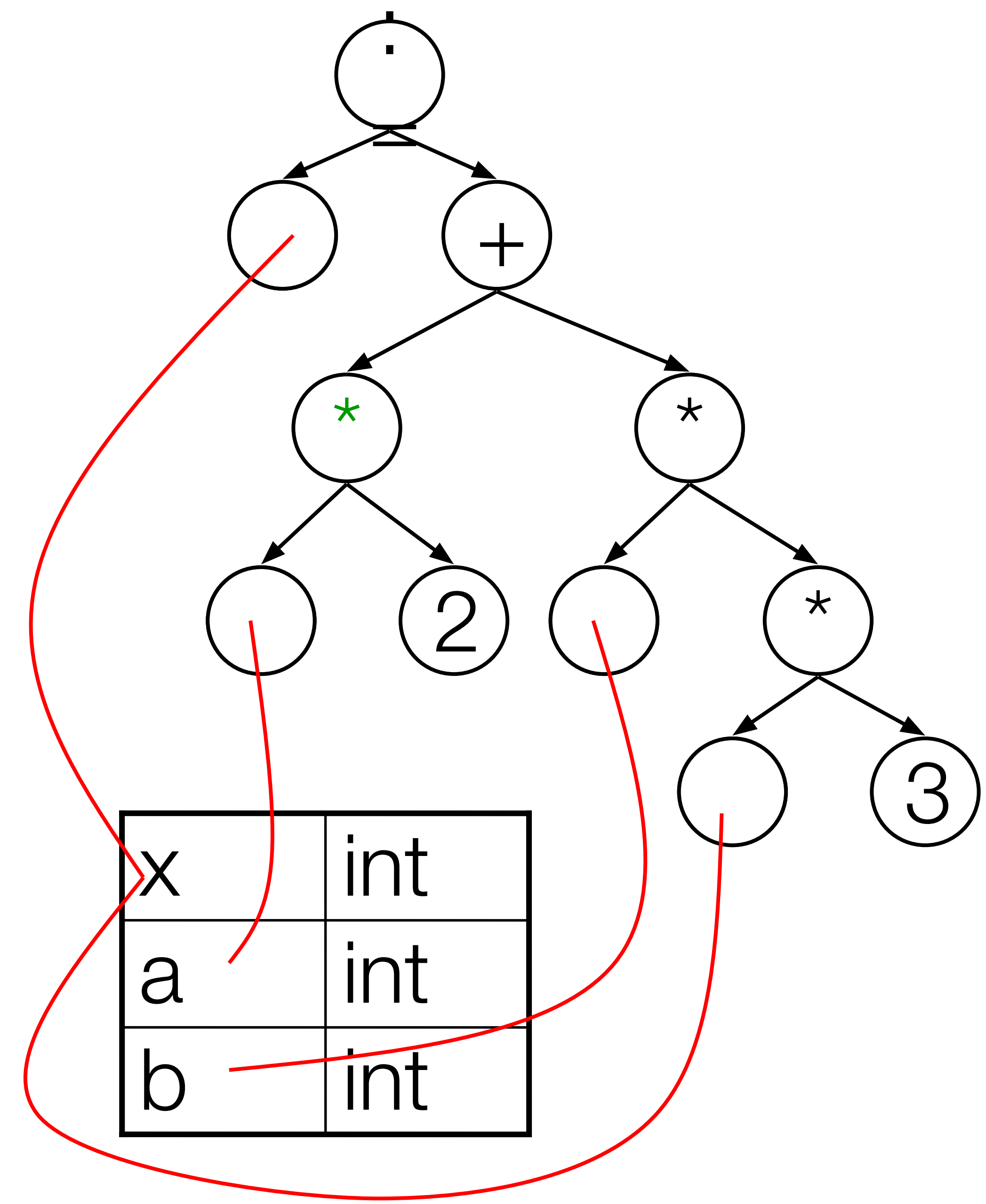
Instruction Selection

- Translates IR into target instruction set
- Choose instructions (**`smul`** or **`sll`**)
- Choose operand modes
  - immediate constants (2 or 3)
  - load immediates
  - addressing modes
- Complex instructions
- Types of branches
- Use tree grammars & dynamic programming



Instruction Selection

```
r1 ← load  M[fp+x]
r2 ← loadi 3
r3 ← mul   r1, r2
r4 ← load  M[fp+b]
r5 ← mul   r3, r4
r6 ← load  M[fp+a]
r7 ← sll   r6, 1
r8 ← add   r6, r5
     store M[fp+x] ← r8
```



Optimizations

- Improves the code by some metric:
  - code size
  - register usage
  - speed
  - power consumption
- Types of optimizations:
  - Basic block (peephole)
  - Global (loop hoisting)
  - Interprocedural (leaf functions)
  - Whole program (inlining of methods)
- Uses: flow analysis, etc.

```
r1 ← load M[fp+x]
r2 ← loadi 3
r3 ← mul r1, r2
r4 ← load M[fp+b]
r5 ← mul r3, r4
r6 ← load M[fp+a]
r7 ← sll r6, 1
r8 ← add r7, r5
     store M[fp+x] ← r8
```

Metrics Matter

Assume load takes 3 cycles, mul takes 2 cycles

$r_1 \leftarrow$ load M[fp+x]
$r_2 \leftarrow$ loadi 3
$r_1 \leftarrow$ mul r_1, r_2
$r_2 \leftarrow$ load M[fp+b]
$r_1 \leftarrow$ mul r_1, r_2
$r_2 \leftarrow$ load M[fp+a]
$r_2 \leftarrow$ sll r_2, 1
$r_1 \leftarrow$ add r_1, r_2
store M[fp+x] $\leftarrow r_1$

Cycles:14

$r_1 \leftarrow$ load M[fp+x]
$r_4 \leftarrow$ load M[fp+b]
$r_6 \leftarrow$ load M[fp+a]
$r_2 \leftarrow$ loadi 3
$r_1 \leftarrow$ mul r_1, r_2
$r_1 \leftarrow$ mul r_1, r_4
$r_6 \leftarrow$ sll r_6, 1
$r_1 \leftarrow$ add r_6, r_1
store M[fp+x] $\leftarrow r_1$

Cycles:9

Register Allocation

- Assign variables to registers and/or memory locations
- Decisions are crucial!
- Take into account
  - specialized registers (fp, sp, mul on x86)
  - calling conventions
  - number and type
  - lifetimes
- graph coloring and linear scan are the most commonly-used algorithms

$$
\begin{array}{lll}
r_1 & \leftarrow & \texttt{load M[fp+x]} \\
r_4 & \leftarrow & \texttt{load M[fp+b]} \\
r_6 & \leftarrow & \texttt{load M[fp+a]} \\
r_2 & \leftarrow & \texttt{loadi 3} \\
r_1 & \leftarrow & \texttt{mul } r_1,\ r_2 \\
r_1 & \leftarrow & \texttt{mul } r_1,\ r_4 \\
r_6 & \leftarrow & \texttt{sll } r_6,\ 1 \\
r_1 & \leftarrow & \texttt{add } r_6,\ r_1 \\
 & & \texttt{store M[fp+x]} \leftarrow r_1
\end{array}
$$

Compilers at 45K

Compilers

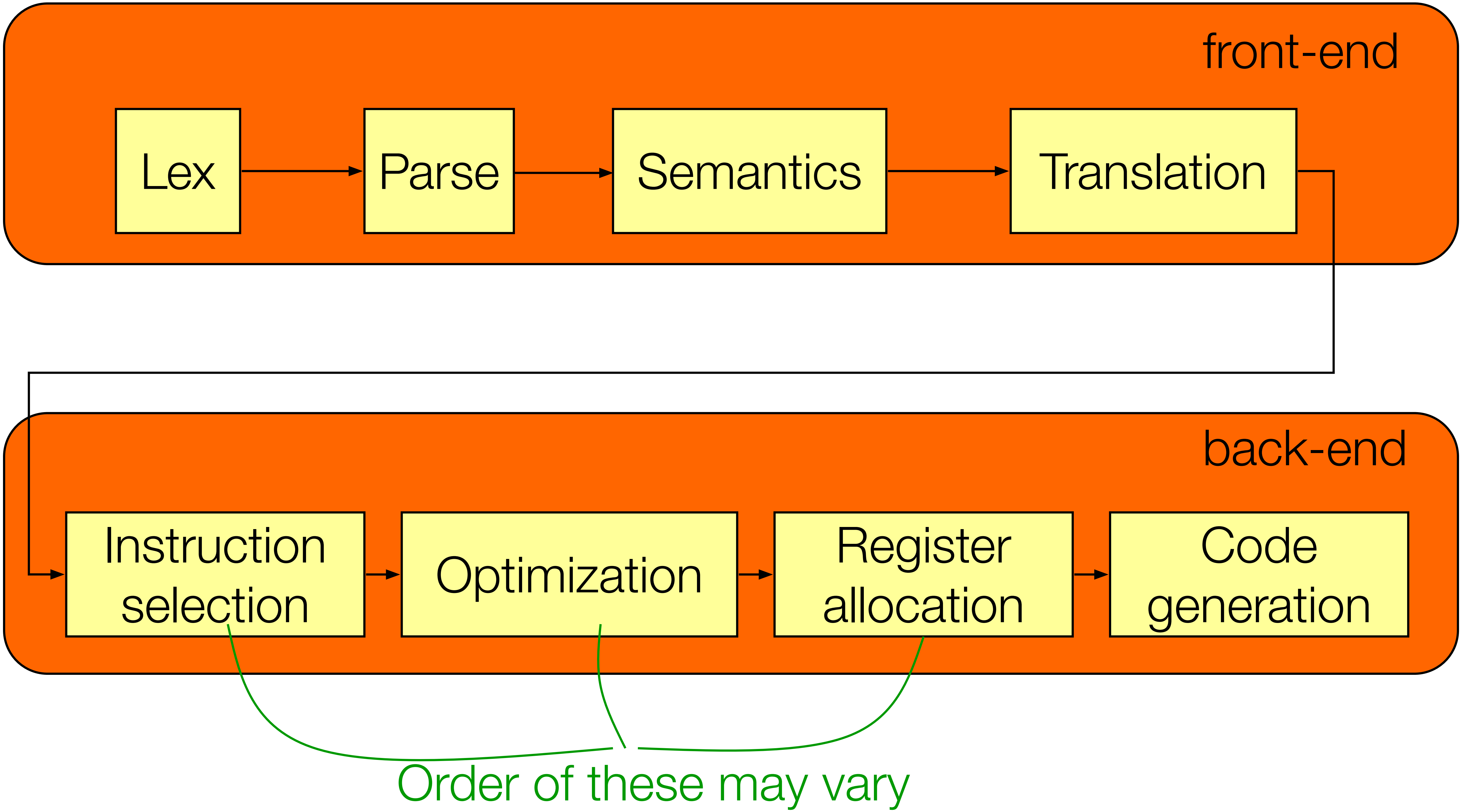
- A compiler translates a programming language (source language) into executable code (target language)

- Quality measures for a compiler
  - Correctness (Does the compiled code work as intended?)
  - Code quality (Does the compiled code run fast?)
  - Efficiency of compilation (Is compilation fast?)
  - Usability (Does the compiler produce useful errors and warnings?)

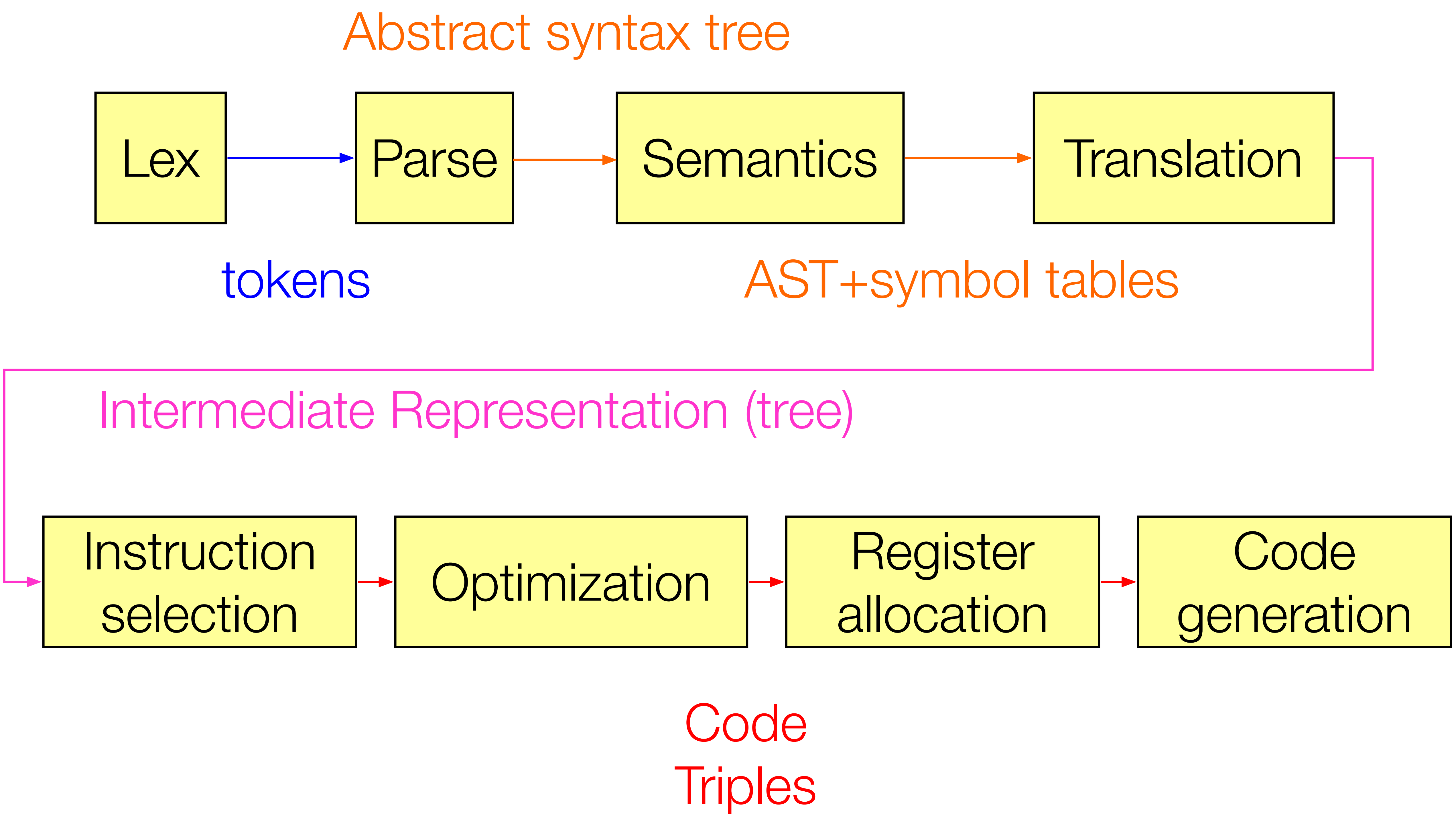
Organizing a Compiler

- Split work into different compiler phases **!!**
- Phases transform one program representation into another
- Every phase has a clear role, some more complex than others
- Phases can be between different types of program representations
- Phases can be on the same program representation

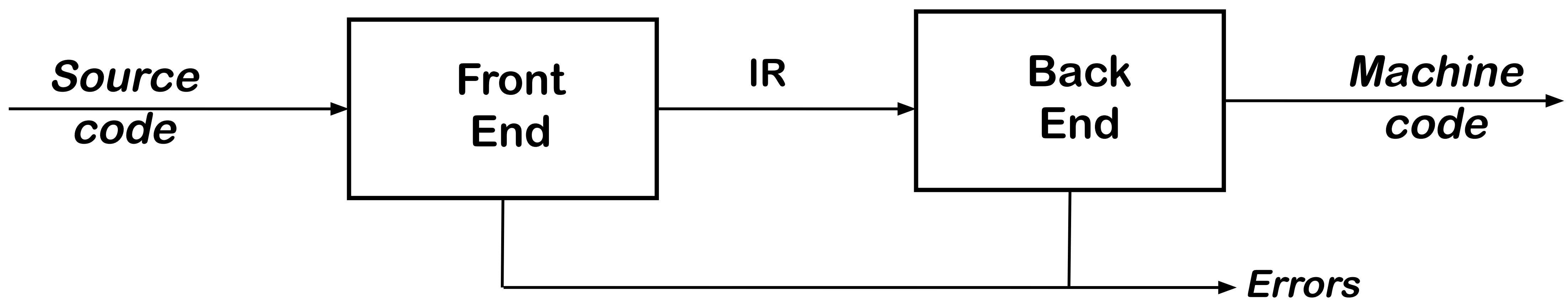
Example phases of a compiler



Many representations



Traditional Two-pass Compiler

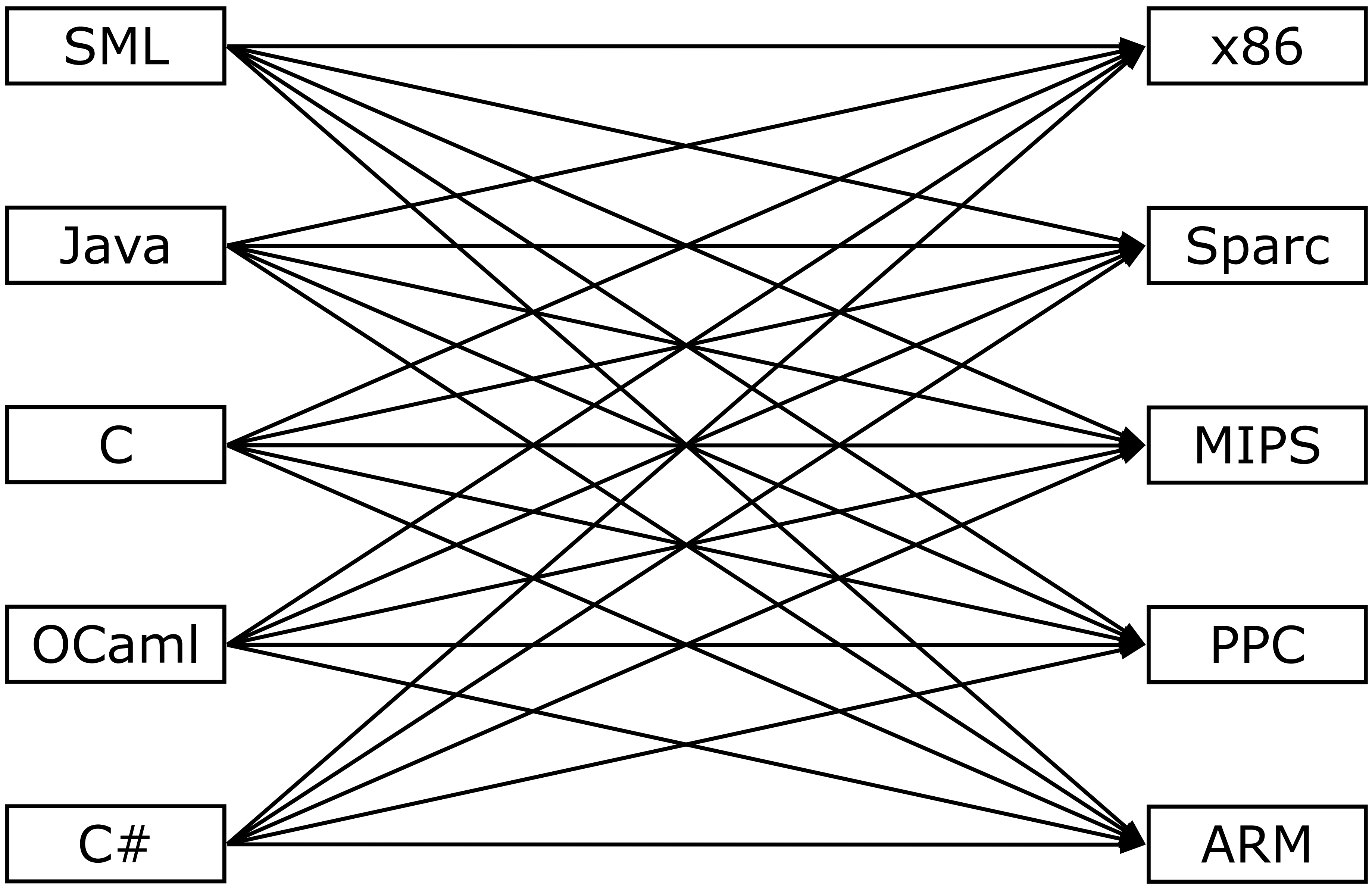


Implications

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Supports independence between source and target

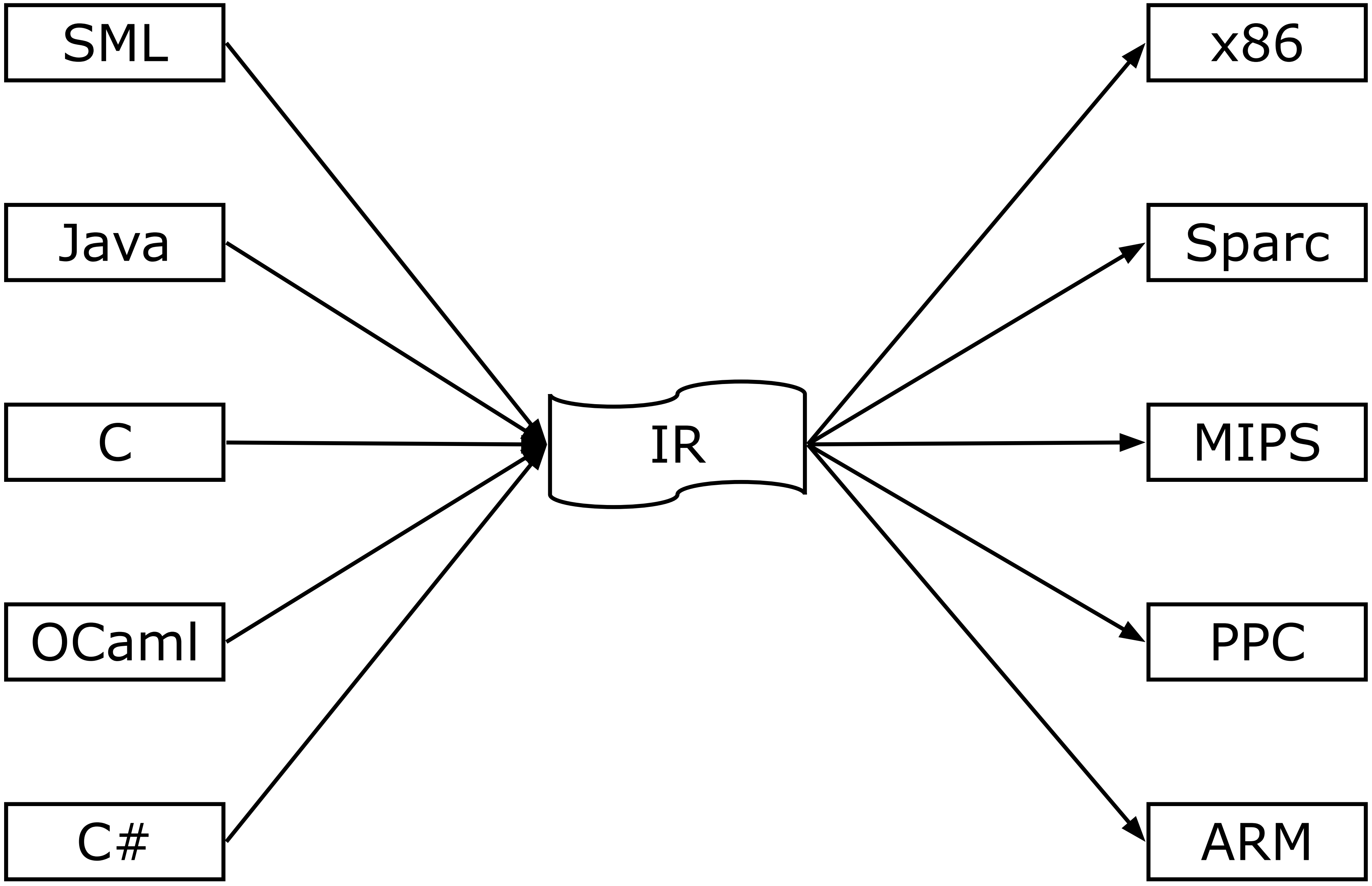
Typically, front end is O(n) or O(n log n), while back end is NP-hard

Without IR



n×m compilers!

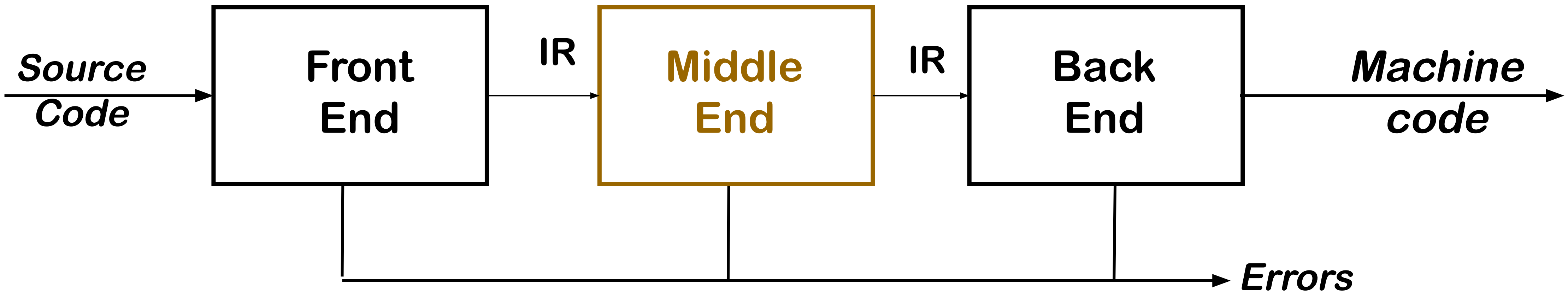
With IR



P.S. No compiler has a truly universal IR (so far).

vs n+m compilers

Traditional Three-pass Compiler



Code Improvement (or *Optimization*)

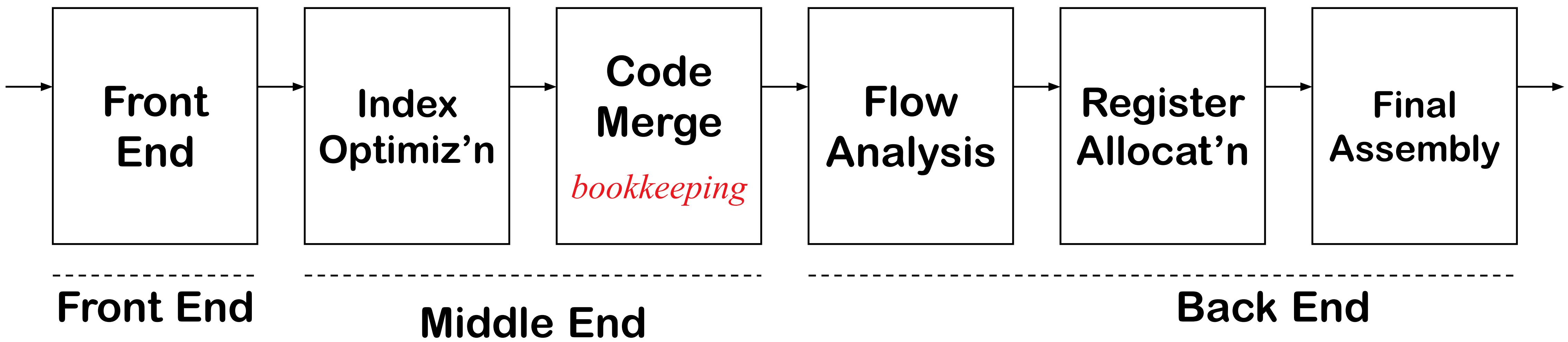
- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to improve program ("optimize")
  - Execution time space, power consumption, …
- Must preserve "meaning" of the code
  - Correct behavior, output of the program

Compilers is a “Mature” Field

- Compiler History
  - 1943: Plankalkül, first high-level language (Konrad Zuse)
  - 1951: Formules, first self-hosting compiler
  - 1952: A-0, term ‘compiler’ (Grace Hopper)
  - 1957: FORTRAN, first commercial compiler (John Backus; 18 PY)
  - 1962: Lisp, self-hosting compiler and GC (Tim Hart and Mike Levin)
- Compilers today
  - Modern compilers are complex (gcc has 7.5M LOC)
  - There is still a lot of compiler research (LLVM, verified compilation, …)
  - There is still a lot of compiler development in industry (guest lecture?)

Classic Compilers

1957: The FORTRAN Automatic Coding System



- Six passes in a fixed order
- Generated good code

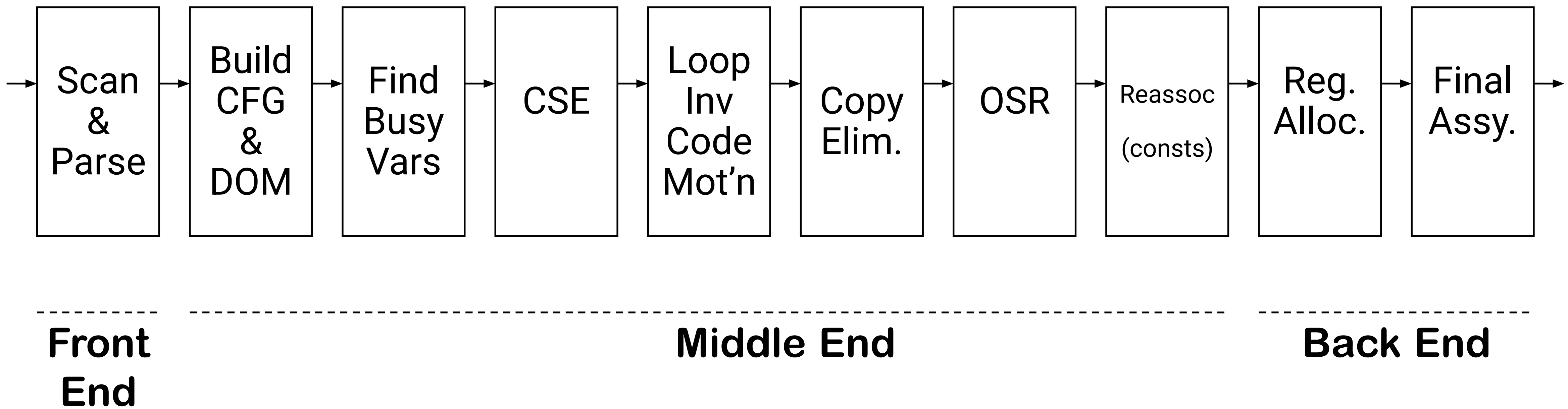
  Assumed unlimited index registers

  Code motion out of loops, with ifs and gotos

  Did flow analysis & register allocation

Classic Compilers

1969: IBM's FORTRAN H Compiler



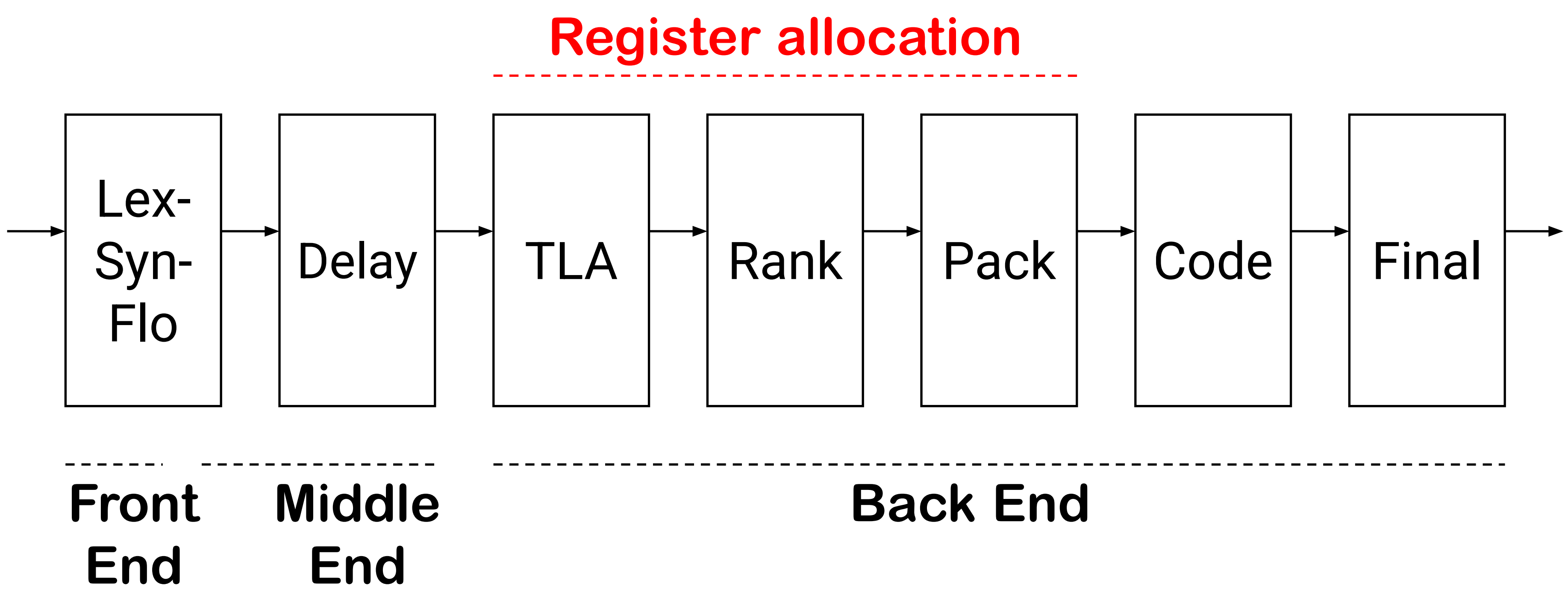
- Used low-level IR (quads), identified loops with dominators
- Focused on optimizing loops ("inside out" order)

  Passes are familiar today

- Simple front end, simple back end for IBM 370

Classic Compilers

1975: BLISS-11 compiler (Wulf *et al.*, CMU)



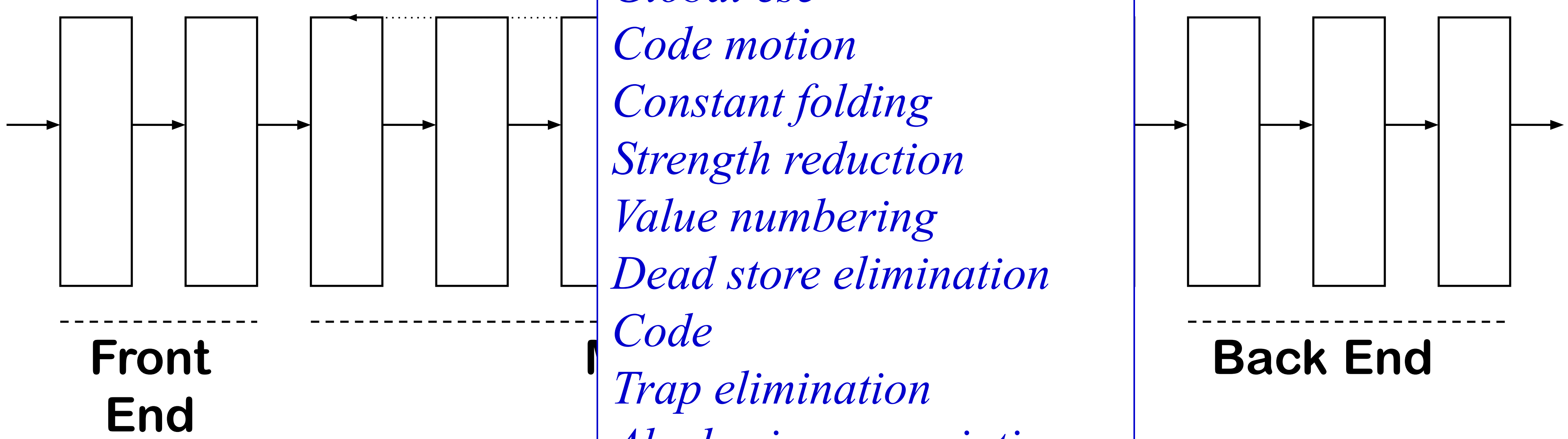
- The great compiler for the PDP-11
- Seven passes in a fixed order
- Focused on code shape & instruction selection

  LexSynFlo did preliminary flow analysis

  Final included a grab-bag of peephole optimizations

Classic Compilers

1980: IBM's PL.8 Compiler



Dead code elimination
Global cse
Code motion
Constant folding
Strength reduction
Value numbering
Dead store elimination
Code
Trap elimination
Algebraic reassociation

- Many passes, one front end, several back ends
- Collection of 10 or more passes

  Repeat some passes and analyses

  Represent complex operations at 2 levels

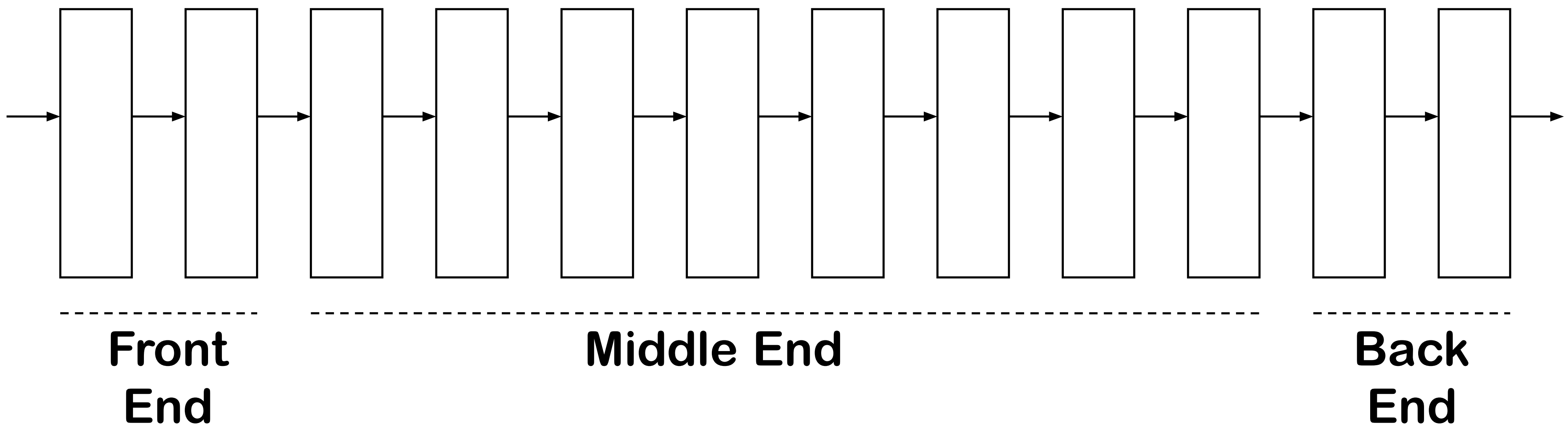
  Below machine-level IR

Multi-level IR has become common wisdom

*

Classic Compilers

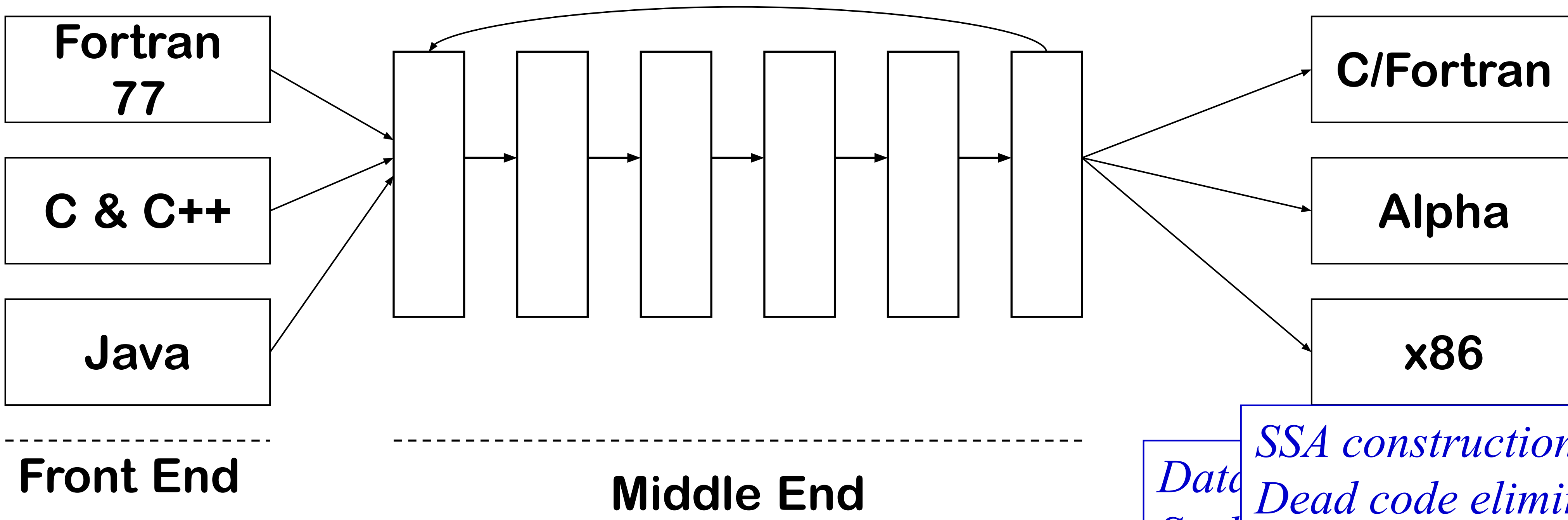
1986: HP's PA-RISC Compiler



- Several front ends, an optimizer, and a back end
- Four fixed-order choices for optimization (9 passes)
- Graph-coloring allocator, instruction scheduler, peephole optimizer

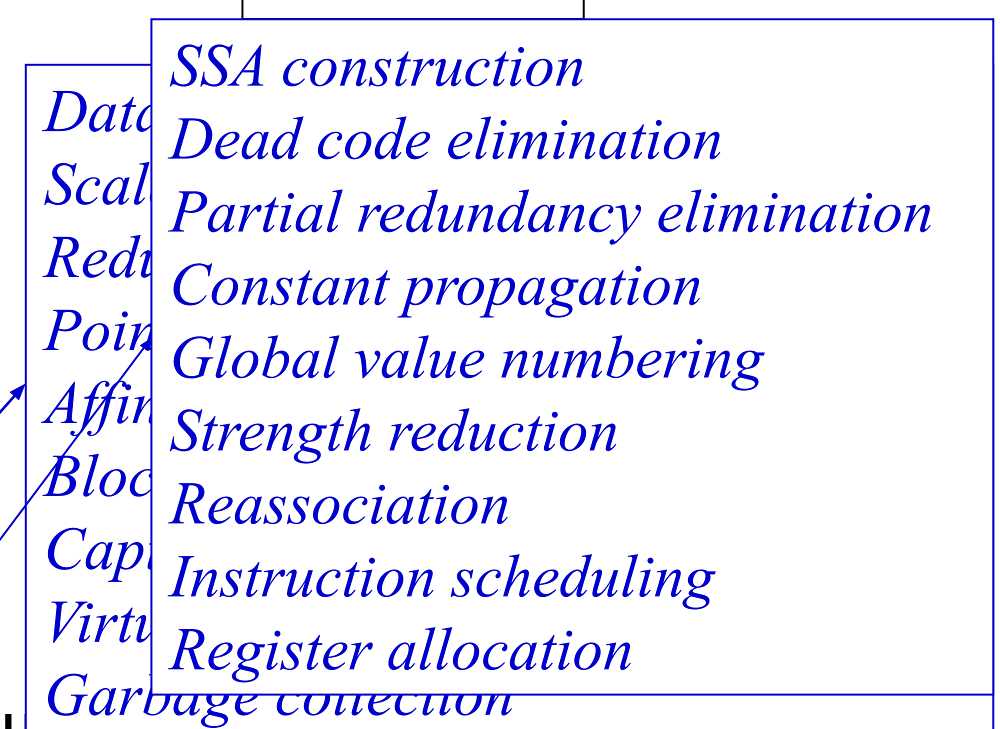
Classic Compilers

1999: The SUIF Compiler System



Another classically-built compiler

- 3 front ends, 3 back ends
- 18 passes, configurable order
- Two-level IR (High SUIF, Low SUIF)
- Intended as research infrastructure



Logisitics

Course Staff – Seth Copen Goldstein

- Office hours: Wed 1pm-3pm 7111GHC or zoom (link on piazza)

- Research
  - Concurrent Systems (Parallel, Distributed, …)
  - Architecture/Compilers
  - Monetary Systems (BoLT) & Future of Work
  - Web3

- Teaching
  - 15-411/611 Compiler Design
  - 15-319/619 Cloud Computing
  - 15-213 Introduction to Computer Systems

Course Staff – Ben L. Titzer

- Office hours: Tue 2pm - 4pm
- Research
  - Virtual machine design (Wizard Research Engine)
  - All things WebAssembly
  - Systems programming languages (Virgil)
- SG Teaching
  - 17-363 Programming Language Pragmatics (with Aldrich)
  - 17-770 Virtual Machines and Managed Languages

Communication and Resources

- **Lecture:** Tue/Thu 9:30-10:50am at DH A302
- **Recitation**

A:	Fri 1:00pm	GHC 4102
B:	Fri 2:00pm	BH 235A
C:	Fri 4:00pm	WEH 5312
D:	Fri 1:00pm	GHC 4301

- **Website:** http://www.cs.cmu.edu/~411
- **Piazza:** You should be on already
- Gradescope: Enrollment code on Piazza
- **Lecture notes:** Will be available after the lecture
- **Textbook:** Andrew Appel - Modern Compiler Implementation in ML

The Essential TAs!

Name

- Something about yourself

- Languages Prefer

Picture

Kyle Booker

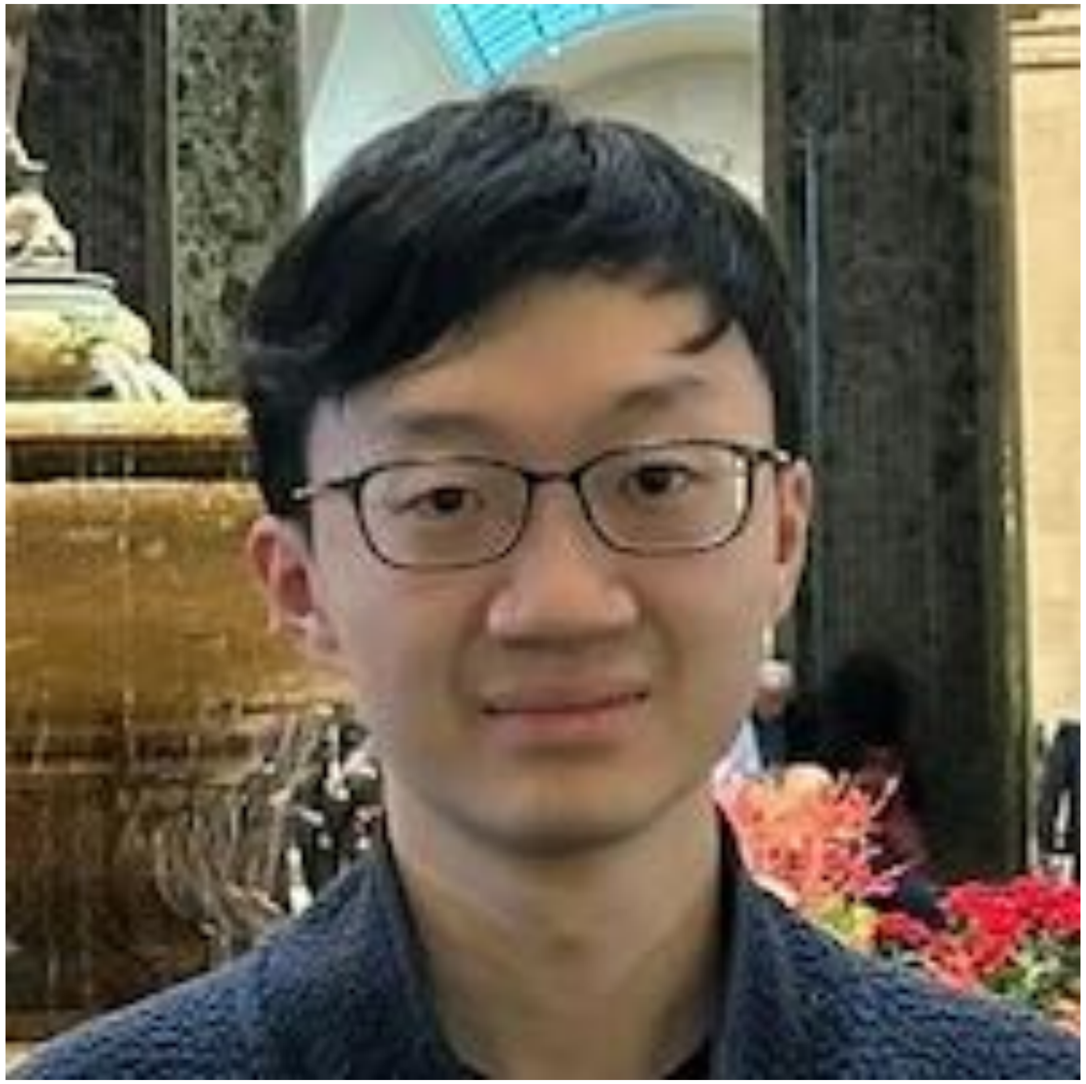
- Senior in CS
- I play in a rock band

- OCaml & Rust

Stephen Nah

- Senior in CS
- I play the drums!

- Rust



Ziqi Liu

- First-year MSCS; undergrad at CMU
- Very into volleyball

- OCaml

Iván Burgert

- Senior in CS
- I’m from Argentina 🇦🇷

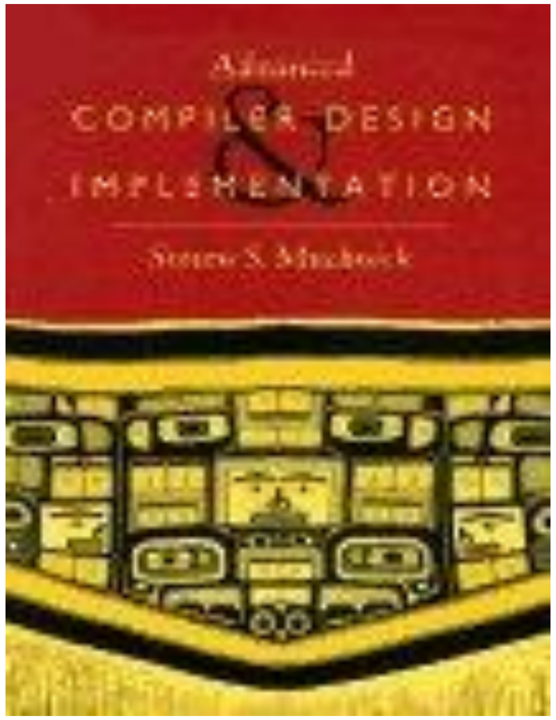
- OCaml (working on learning Rust!)

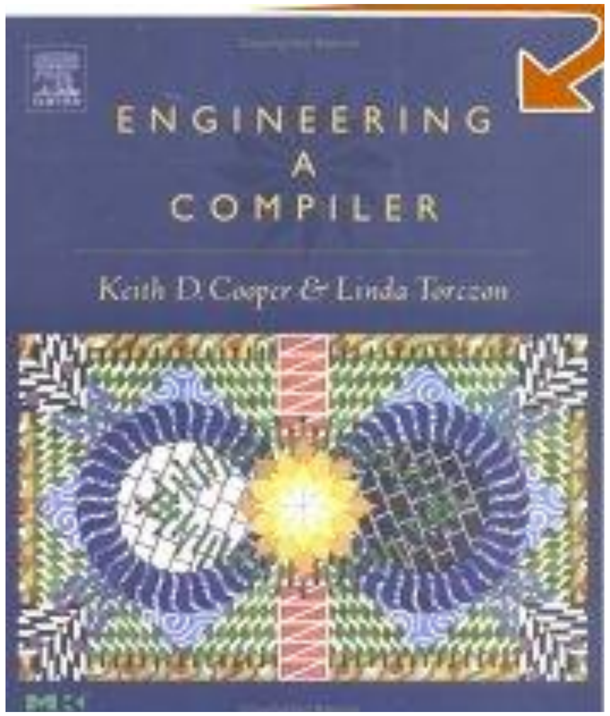
Alex Knox

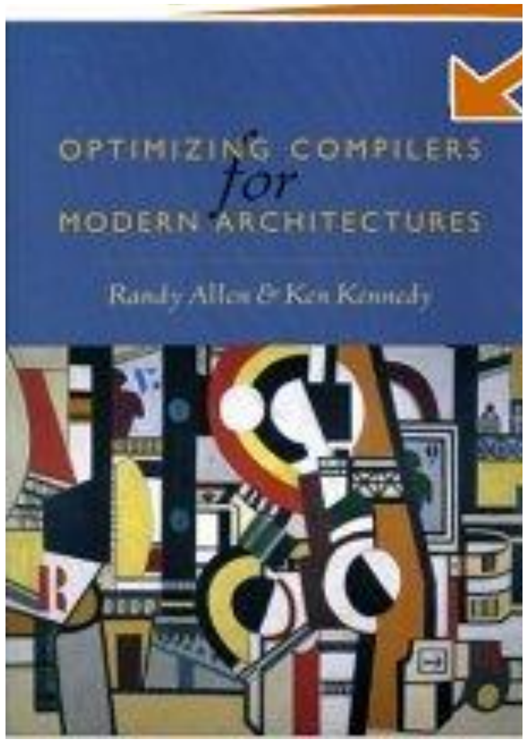
- Senior in CS
- I play the bagpipe :)

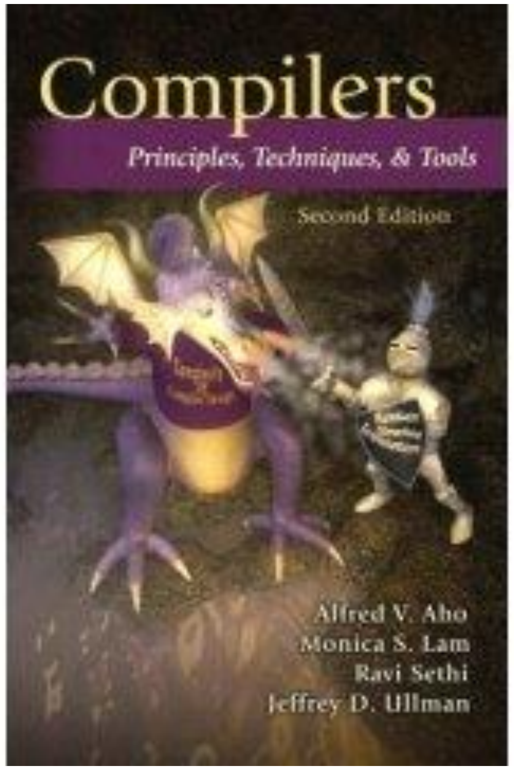
- OCaml

Other Textbooks









What will you learn?

Compiler Design

- How to structure compilers
- Applied algorithms and data structures
  - Context-free grammars and parsing
  - Static single assignment form
  - Data flow analysis and type checking
  - Chordal graph coloring and register allocation
- Focus on sequential imperative programming language
  Not functional, parallel, distributed, object-oriented, ...
- Focus on code generation and optimization
  Not error messages, type inference, runtime system, ...

Focus of the Course

- Correctness (Does the compiled code work as intended?)
- Code quality (Does the compiled code run fast?)
- Efficiency of compilation (Is compilation fast?)
- Usability (Does the compiler produce useful errors and warnings?)

Software Engineering

We won't discuss this much in lecture.

- Implementing a compiler is a substantial software project
  - Building, organizing, testing, debugging, specifying, …
- Understanding and implementing high-level specifications
- Satisfying performance constraints
- Make (and reevaluate) design decision
  - Implementation language and libraries
  - Data structures and algorithms
  - Modules and interfaces
- Revise and modify your code

Compilers are perfect to practice software engineering.

Learning Goals I

- Distinguish the main phases of a state-of-the-art compiler
- Understand static and dynamic semantics of an imperative language
- Develop parsers and lexers using parser generators
- Perform semantic analysis
- Translate abstract syntax trees to intermediate representations and static single assignment form
- Analyze the dataflow in an imperative language
- Perform standard compiler optimizations

Learning Goals II

- Allocate registers using a graph-coloring algorithm
- Generate efficient assembly code for a modern architecture
- Understand opportunities and limitations of compiler optimizations
- Appreciate design tradeoffs and how representation affects optimizations
- Develop complex software following high-level specifications

How will this work?

Your Responsibilities

No exams.

- Attend lectures
  - Lecture notes are only supplementary material
- 5 Labs: you will impl. compilers for subsets of C0 to x86-64 assembly
  - Lab1-4: each worth 100 points (total 400 points)
  - Code review after Lab 3: 60 points
  - Lab 5: 200 points + 100 points for report

With a partner or individual.

- 4 Assignments: you will complete four problem sets that help you understand the material presented in the lectures
  - Assignments 1-4: each 60 points (total 200 points)

Individual.

Labs — Overview

- Labs (700 points)
  - Lab 1: tests and compiler for L1 (straight-line code)
  - Lab 2: tests and compiler for L2 (conditionals and loops)
  - Lab 3: tests and compiler for L3 (functions)
  - Lab 4: tests and compiler for L4 (memory)
  - Lab 5: compiler and paper (optimizations)

Auto graded.

TA graded.

- Code review (60 points)
  - You show your code for Lab 3 and get feedback
  - We expect that every team member is familiar with all components
  - We expect that every team member contributes equally

TA graded.

Support for 411/611 Comes From …

Jane Street

Helps to

- Improve the grading infrastructure
- Pay for AWS cost

Source Language: C0

Subset of C

- Small
- Safe
- Fully specified
- Rich enough to be representative and interesting
- Small enough to manage in a semester

Target Language

x86-64 architecture

- Widely used
- Quirky, but you can choose the instructions you use
- Low level enough you can get a taste of the hardware

Runtime system

- C0 uses the ABI (Application Binary Interface) for C
- Strict adherence (internally, and for library functions)

Finding a partner for the labs

I strongly suggest you work in teams of two.

Labs — Finding a Partner

Don't panic.

There are two options

1. You fill out a questionnaire and we *suggest* a partner (staff selection)
   - Suggestion is not binding but it's expected that you team up

2. You team up with somebody yourself (self selection)
   - Like in previous iterations of the course

Register your team on of before Monday 1/20.

Option 1: Staff Selection

- You fill out a questionnaire about
  - Your plans and goals for the class
  - Your strengths and work style
  - And your time constraints
- We suggest a partner with complementary strengths and similar plans/goals
- You meet with your partner and (hopefully) decide to team up
- Advantages:
  - You will get a partner who is a good match
  - You will likely meet somebody new
  - Prepares you for working in a software company

Until Thursday

Friday

Until Monday 1/20

Option 1: Example Questions we Ask

- What programming language would you prefer to use?
- Are you more interested in theory or in building systems?
- Are you familiar with x86 assembly?
- How much time would be so much that you would rather drop?
- How much effort do you plan to invest in Compilers, on average?
- What grade are you aiming for in Compilers?
- Do you prefer to collaborate when writing code?

Option 2: Self Selection

- Pick your partner carefully!
- Have an honest discussion about your goals and expectations
  - What grades you are willing to accept?
  - How much time will you spend?
  - What times of day you work best?
- Find somebody who's a good match

That's not necessarily your best friend.

- Go through the questionnaire and compare your answers

Consider switching to Option 1 if there are mismatches.

Labs — Picking a Programming Language

- You can freely choose a programming language to use
- It has been suggested that you use a typed functional language
  - Writing a compiler is a killer app for functional programming
  - Most teams used OCaml last year
- We provide starter code for the following languages
  - SML, **OCaml**, Haskell, and, **Rust**
  - Also, but not recommended: C++ and Java
- When picking a language also consider the availability of parser generators and libraries

Logistics

- Assignments are submitted via Gradescope
- Labs are submitted via GitHub (on Gradescope)
  - Get a GitHub account and fill out a google form to register your team
  - Receive your group name
  - Receive an invitation to join your group on GitHub
  - Submit your code by pushing to your repository
- Local development is available using docker containers
- Auto grading with Gradescope
  - Your compiler is tested against the test cases of other groups
  - And test cases from previous years
  - You can submit as often as you like
  - Best submission before the deadline counts

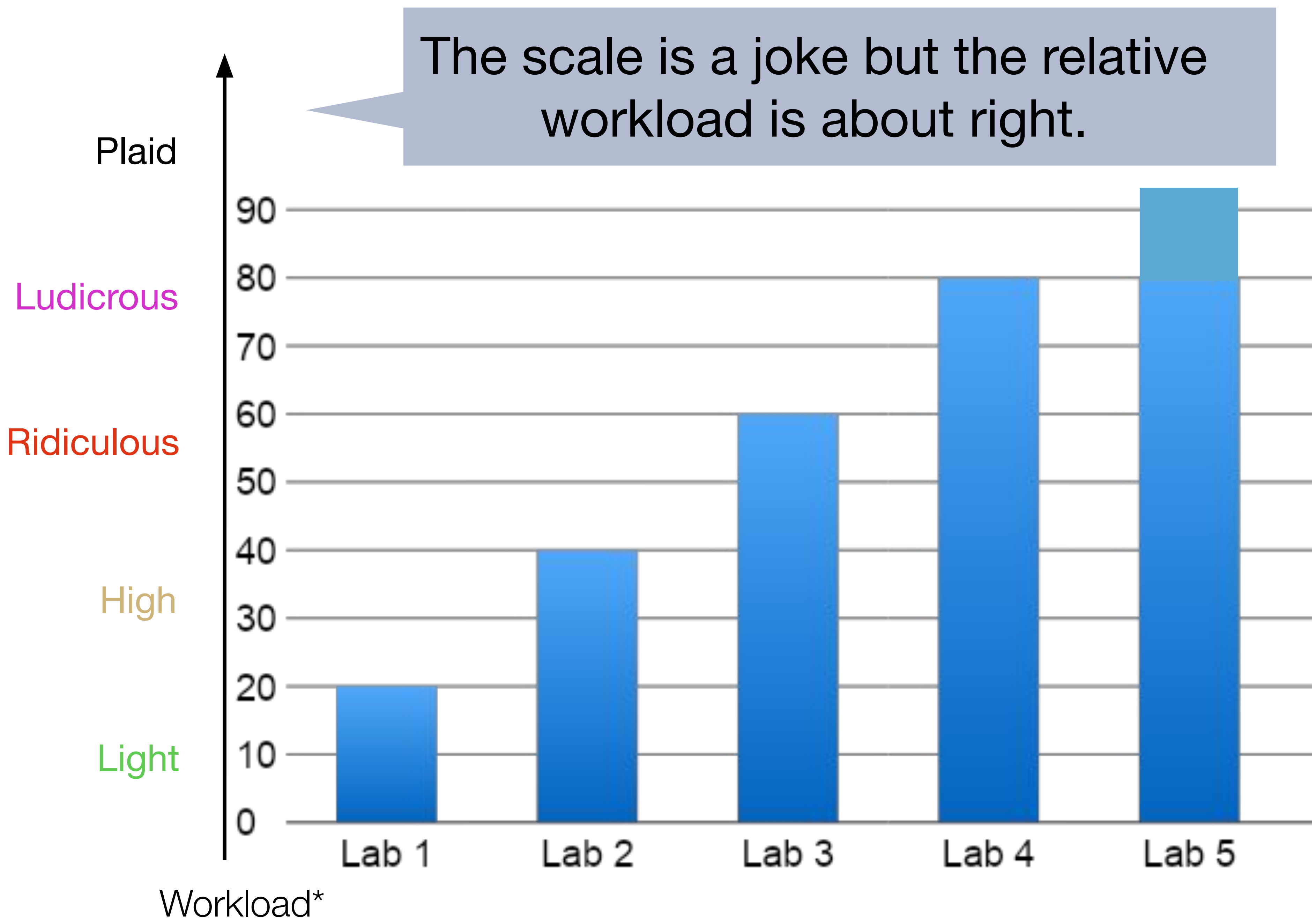
Gradescope Caveats

- You have to give Gradescope permissions to see your `15-411-s25-<groupname>` repo
- You can submit as often as you like, but …
  - Wait for each submission to complete
  - If it takes awhile, that is not because Gradescope hung
  - Submitting multiple times before previous completes will slow things down for everyone

Advice

- Labs are difficult and take time
  - Plan ahead!
  - Set up meetings with lab partners
  - Talk to us and others about design decisions
- Don’t start the compiler after the tests
- Errors carry over to the next lab
- Submit early and often
- Compilers are complex
  - That’s part of the fun
- Consider rewrites

Workload Over the Semester



* scale from the movie Spaceballs.

This Year's Theme - Pixar Characters

Deadlines and Academic Integrity

- Deadlines are midnight (after class); being late results in a late day
  - You have five (5) late days for the labs (see details online)
  - You have three (3) late days for the assignments (details online)
- Talk to Ben or me or your undergrad advisor if you cannot make a deadline for personal reasons (religious holidays, illness, …)
- Don’t cheat! (details online)
  - Use code only from the standard library, add to Readme
  - Don’t use code from other teams, earlier years, etc.
  - If in doubt talk to the instructor
  - The written assignments need to be completed individually (1 person)

Things you Should Use

- Debugger
- Profiler
- Test programs
- Standard library
- Lecture notes
- Textbooks

Well-Being

- This is only a course!
  - Take care of yourself
  - Watch out for others
  - Come speak to us. We really do care.
- Get help if you struggle or feel stressed
  - If you or anyone you know experiences any academic stress, difficult life events, or feelings like anxiety or depression seek support
  - Counseling and Psychological Services (CaPS) is here to help:
    Phone: 412-268-2922
    Web: http://www.cmu.edu/counseling/

Who should take this course?

15-411 in the Curriculum

- 15-213 Introduction to Computer Systems

Prerequisite

- 15-411 Compiler Design
  - How are high-level programs translated to machine code?
- 15-410 Operating System Design and Implementation
  - How is the execution of programs managed?
- 15-441 Computer Networks
  - How do programs communicate?

System requirement

- 15-417 HOT Compilation
  - How to compile higher-order typed languages?

Things you Should Know (Learn)

- C0 programming language
  - The source language
- x86-64 assembly
  - The target language
- Functional programming
  - Recommended?
- Git version control
  - For submitting labs

One of the Topics of this week's recitation

Reminder: inductive definitions

See: Bob Harper's "Practical Foundations for Programming Languages"