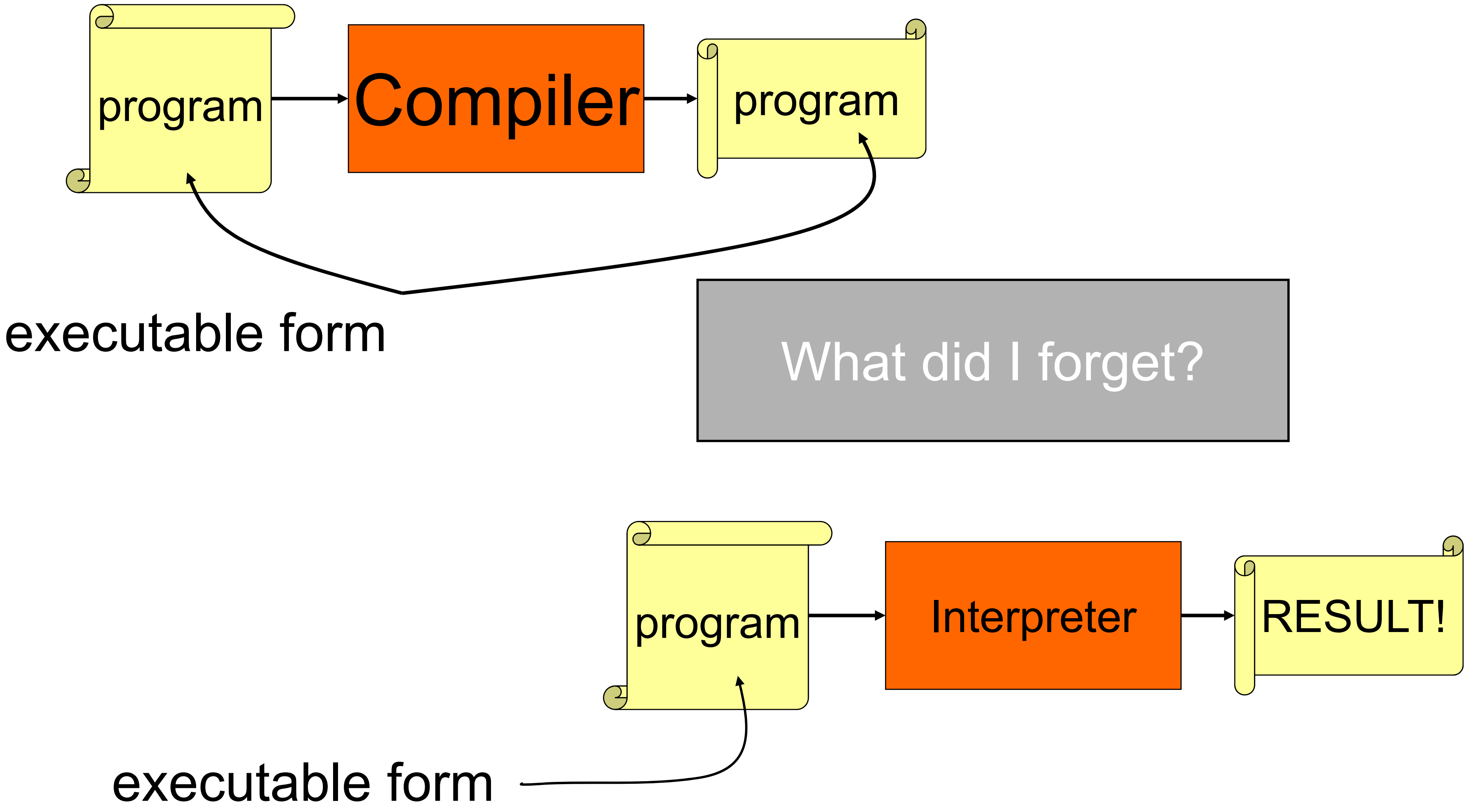
15-411/15-611 Compiler Design

Seth Copen Goldstein — Fall 2020

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

Compilers at 60K

What is a Compiler?

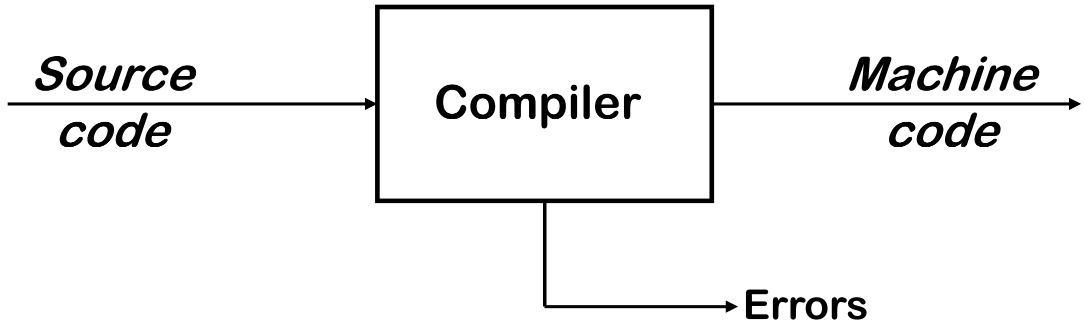


“A” Compiler is a misnomer

- 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

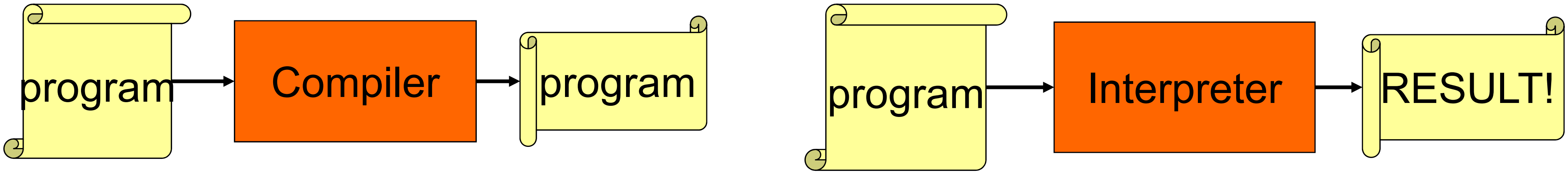


Implications

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

Big step up from assembly language—use higher level notations

Translators



- Compilers transform specifications
- Interpreters execute specifications
- E.g.: C++ is usually compiled
  Lisp is usually interpreted
  Java is not directly interpreted
- Many similarities between the two
- 411 mainly focuses on compilers.

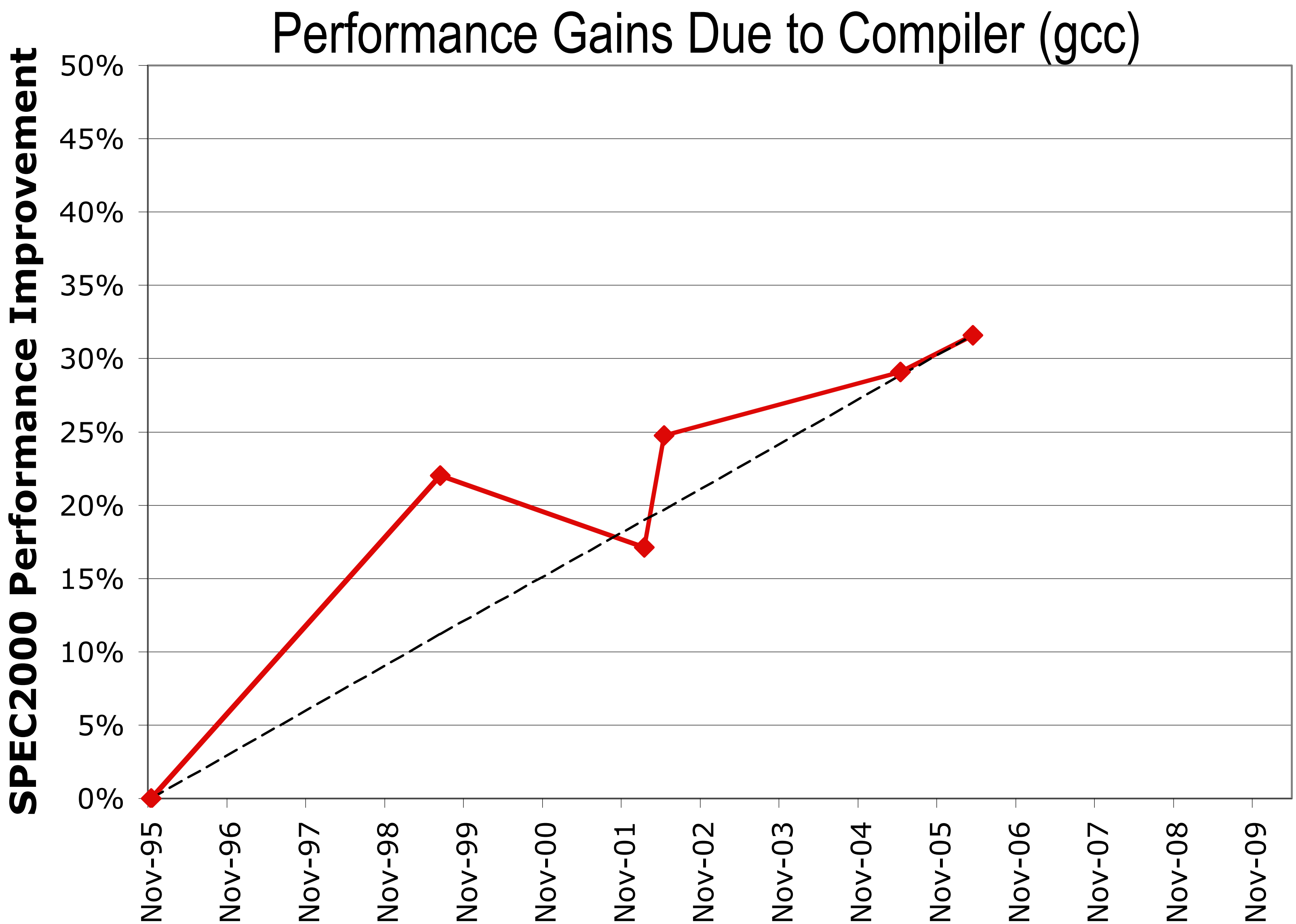
Why take this class?

- Compilers design and construction combines:
  - theory
  - systems
  - architecture
  - AI
  - Engineering

Compilers Are Everywhere

- FTP daemon
- Netscape
- perl, sed, awk, emacs, bc
- excel, tex
- web servers (e.g., asp)
- Databases (query opt)
- ?

Compilers are Essential



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
- Scalability of compiler
  - compiler itself runs quickly
  - Separate compilation
- Easy to modify
- Aids programmer
  - good compile time error messages
  - support for debugger
- Predictable performance of target program

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
```

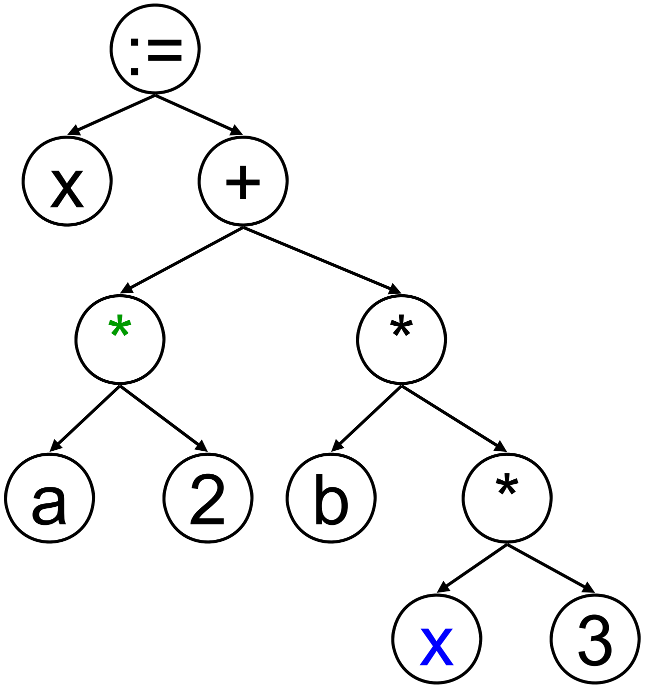
- Group characters into tokens
- Eliminate unnecessary characters from the input stream
- Use regular expressions (to specify) and DFAs to implement.
- E.g., lex

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

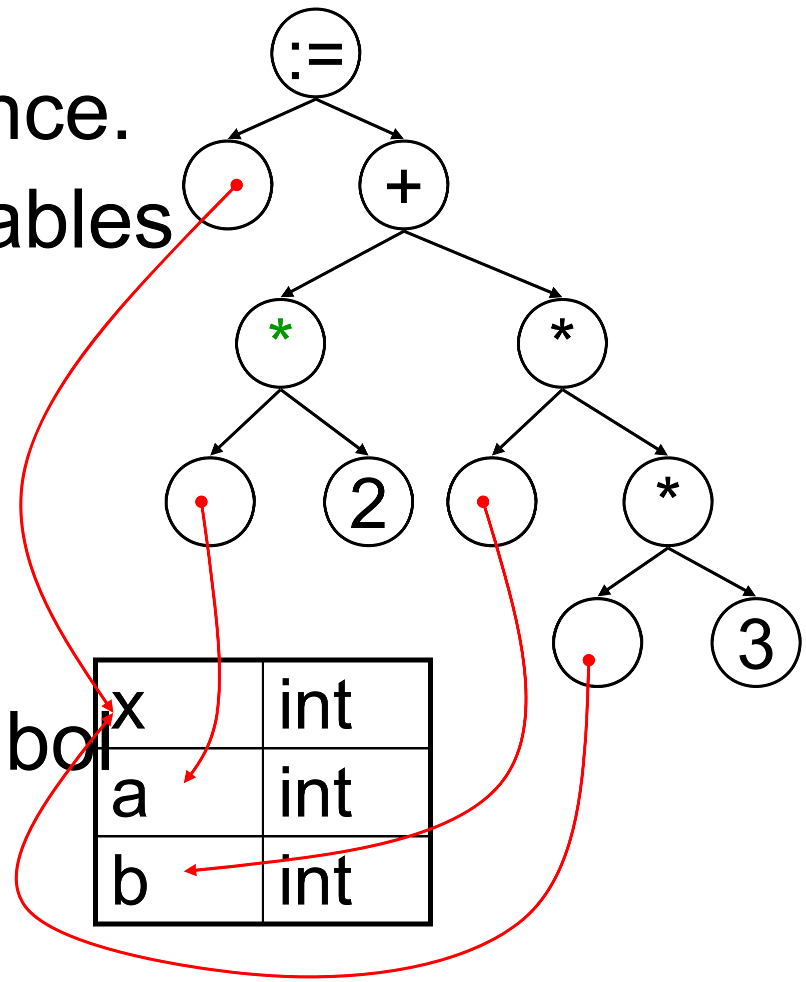


Semantic Analysis

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

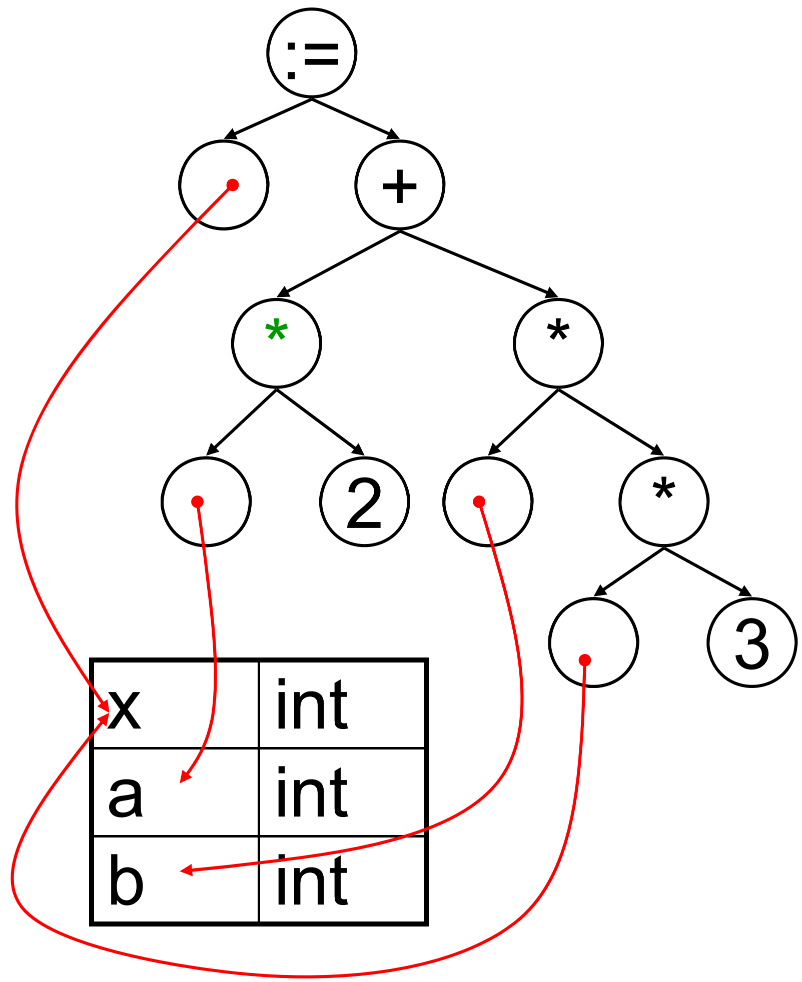
```
id<x> assign id<a> times int<2> plus id<b>
times lparen id<x> times int<3> rparen
```

- 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, ...



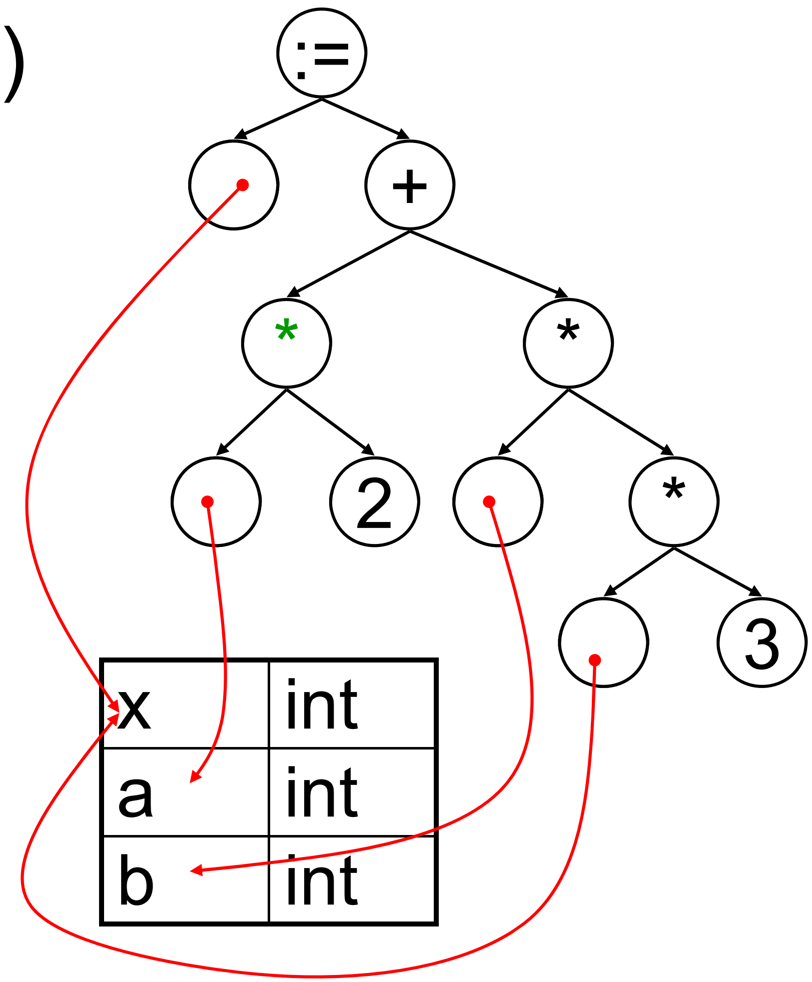
Translation

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



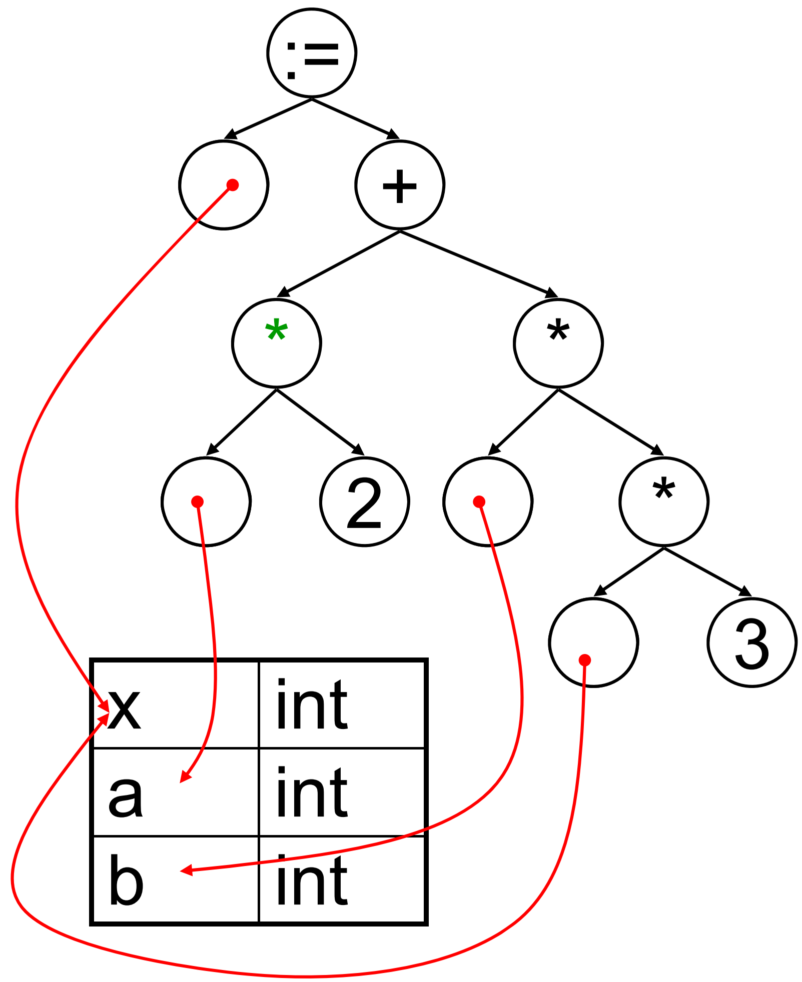
Instruction Selection

- Implements IR in target instruction set.
- Which instructions? (smul or sll)
- What operand modes?
  - immediate constants (2 or 3)
  - load immediates
  - addressing modes
- Complex instructions
- Types of branches
- Use tree grammers & 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
  - registers
  - speed
  - power
- Types of optimizations:
  - Basic block (peephole)
  - Global (loop hoisting)
  - Interprocedural (leaf functions)
- Uses: flow analysis, etc.

$$
\begin{array}{llll}
r_1 & \leftarrow & \texttt{load} & \texttt{M[fp+x]} \\
r_2 & \leftarrow & \texttt{loadi} & \texttt{3} \\
r_3 & \leftarrow & \texttt{mul} & r_1,\ r_2 \\
r_4 & \leftarrow & \texttt{load} & \texttt{M[fp+b]} \\
r_5 & \leftarrow & \texttt{mul} & r_3,\ r_4 \\
r_6 & \leftarrow & \texttt{load} & \texttt{M[fp+a]} \\
r_7 & \leftarrow & \texttt{sll} & r_6,\ 1 \\
r_8 & \leftarrow & \texttt{add} & r_7,\ r_5 \\
 & & \texttt{store} & \texttt{M[fp+x]} \leftarrow r_8
\end{array}
$$

Metrics Matter

Assume load takes 3 cycles, mul takes 2 cycles

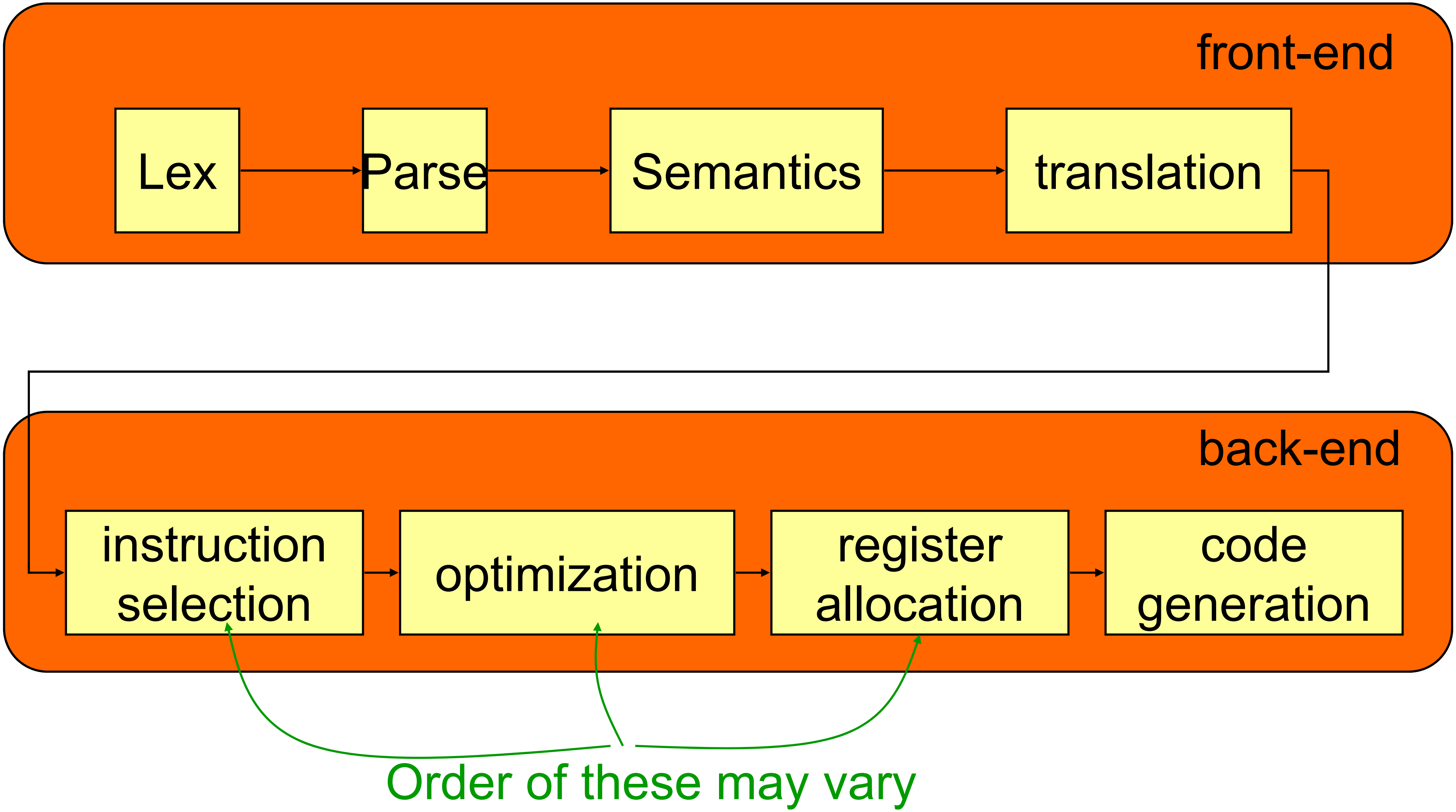
$r_1 \leftarrow$ load M[fp+x]	$r_1 \leftarrow$ load M[fp+x]
$r_2 \leftarrow$ loadi 3	$r_4 \leftarrow$ load M[fp+b]
$r_1 \leftarrow$ mul r_1, r_2	$r_6 \leftarrow$ load M[fp+a]
$r_2 \leftarrow$ load M[fp+b]	$r_2 \leftarrow$ loadi 3
$r_1 \leftarrow$ mul r_1, r_2	$r_1 \leftarrow$ mul r_1, r_2
$r_2 \leftarrow$ load M[fp+a]	$r_1 \leftarrow$ mul r_1, r_4
$r_2 \leftarrow$ sll r_2, 1	$r_6 \leftarrow$ sll r_6, 1
$r_1 \leftarrow$ add r_1, r_2	$r_1 \leftarrow$ add r_6, r_1
store M[fp+x] $\leftarrow r_1$	store M[fp+x] $\leftarrow r_1$
Cycles: 14	Cycles: 9

Register Allocation

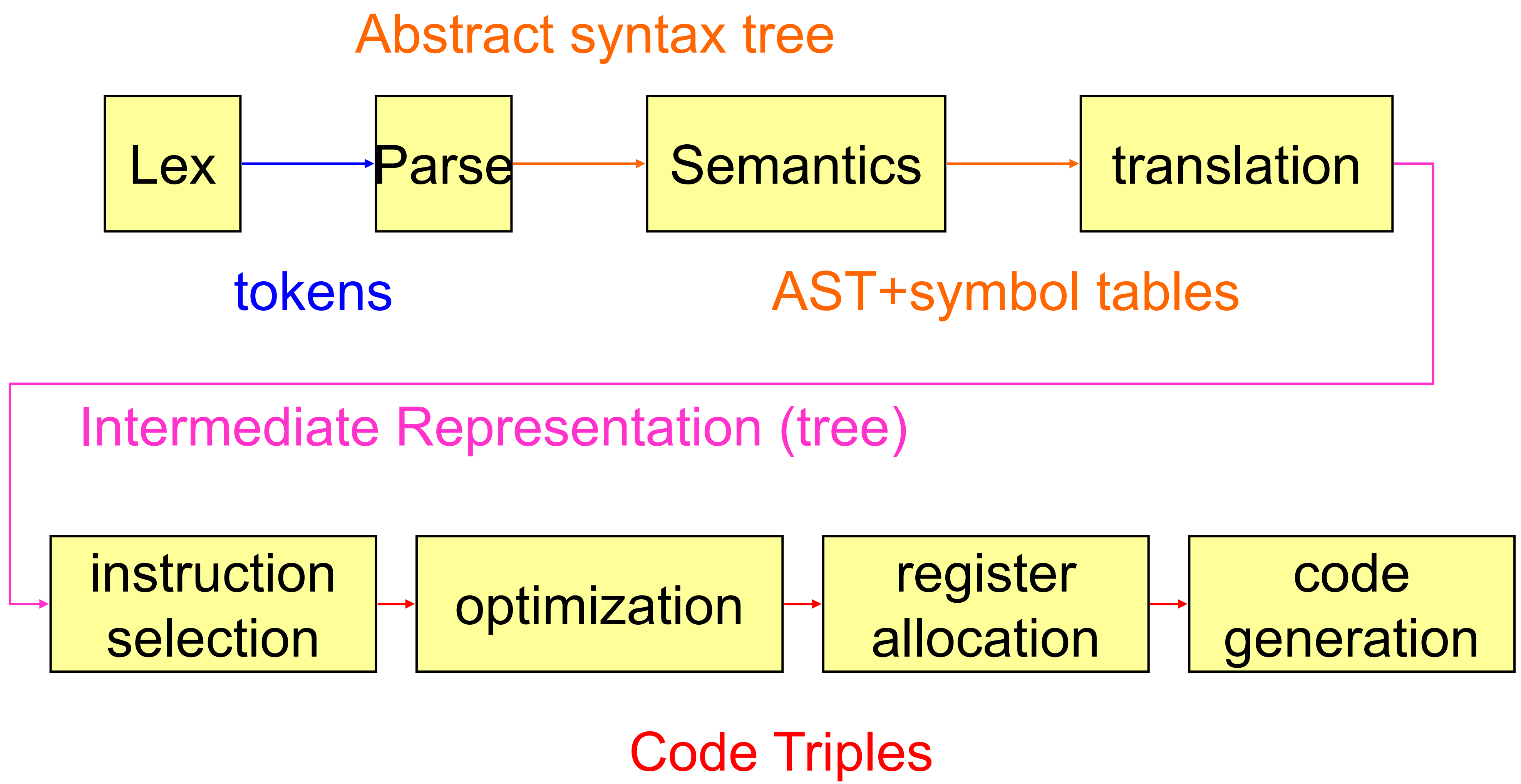
- Assign memory locations to registers
- Crucial!
- Take into account
  - specialized registers (fp, sp, mul on x86)
  - calling conventions
  - number and type
  - lifetimes
- graph coloring one method.

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

The phases of a compiler



Many representations



Compilers at 45K

Compilers

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

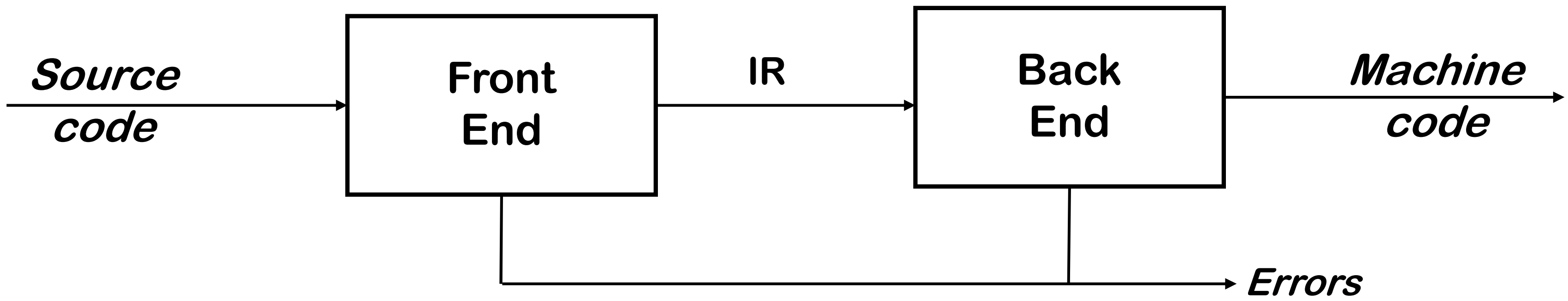
- Quality measures for a compiler
  - Correctness (Does the compiled code work as intended?)
  - Code quality (Does the compiled code run fast?)
  - Efficiency of compilation (Is compilation fast?)
  - Usability (Does the compiler produce useful errors and warnings?)

Compilers

- 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?)

Traditional Two-pass Compiler

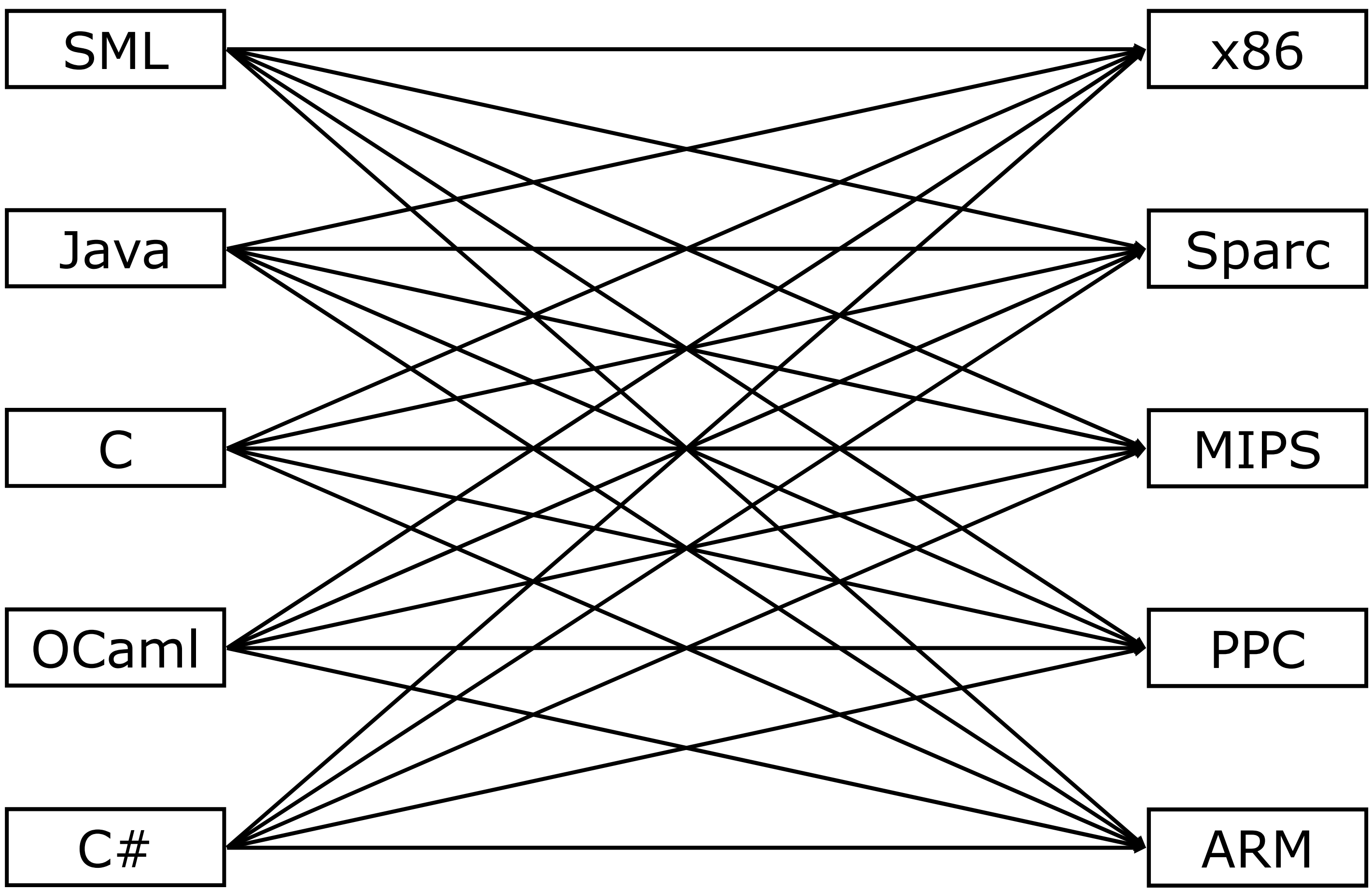


Implications

- 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

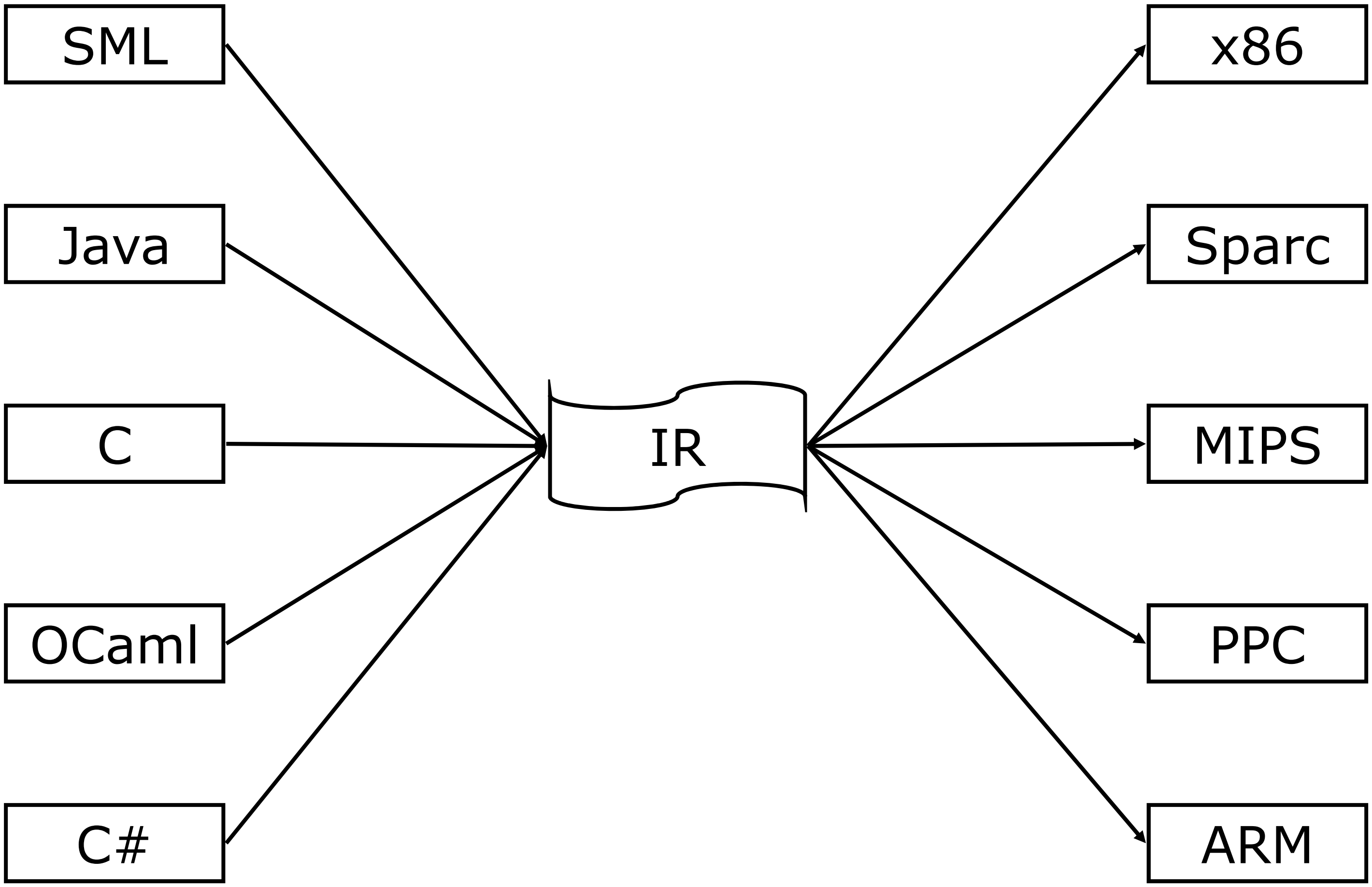
Typically, front end is O(n) or O(n log n), while back end is NP-hard

Without IR



n×m compilers!

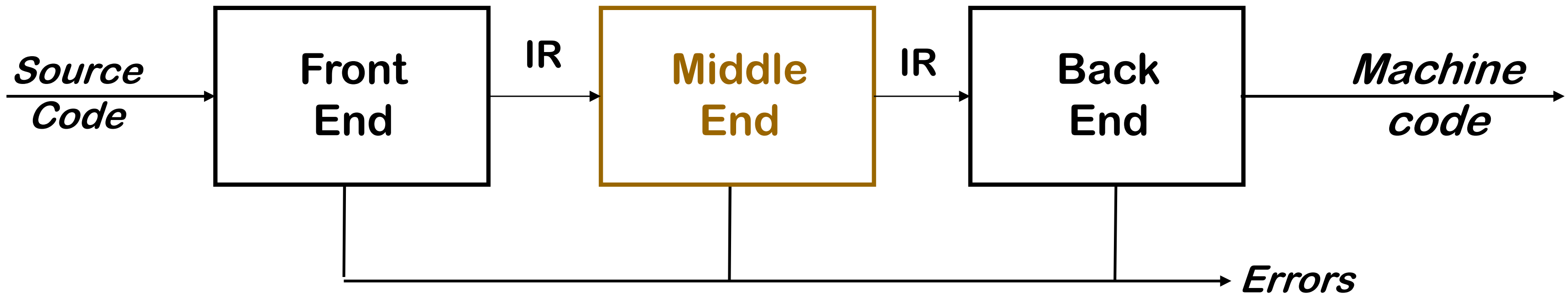
With IR



P.S. This doesn't really happen in the real world.

vs n+m compilers

Traditional Three-pass Compiler



Code Improvement (or *Optimization*)

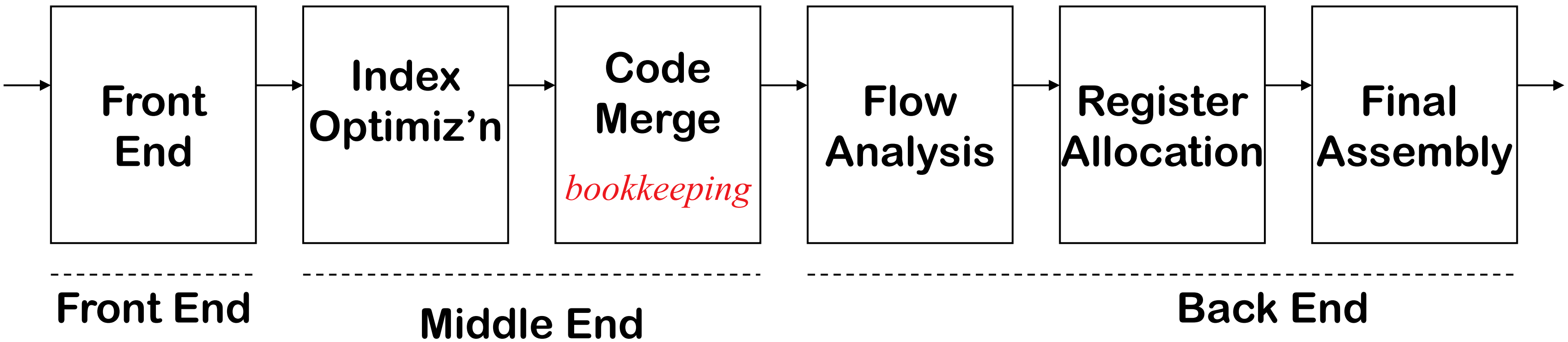
- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variables

Organizing a Compiler

- Split work into different compiler phases **!!**
- Phases transform one program representation into another
- Every phase is as simple as possible
- Phases can be between different types of program representations
- Phases can be on the same program representation

Classic Compilers

1957: The FORTRAN Automatic Coding System



- Six passes in a fixed order
- Generated good code

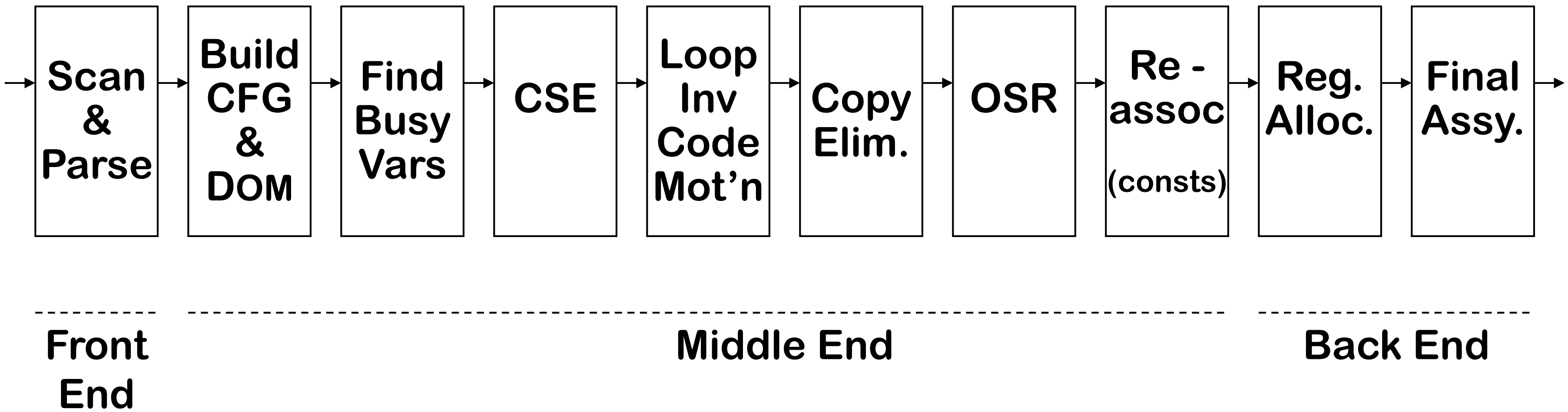
  Assumed unlimited index registers

  Code motion out of loops, with ifs and gotos

  Did flow analysis & register allocation

Classic Compilers

1969: IBM's FORTRAN H Compiler

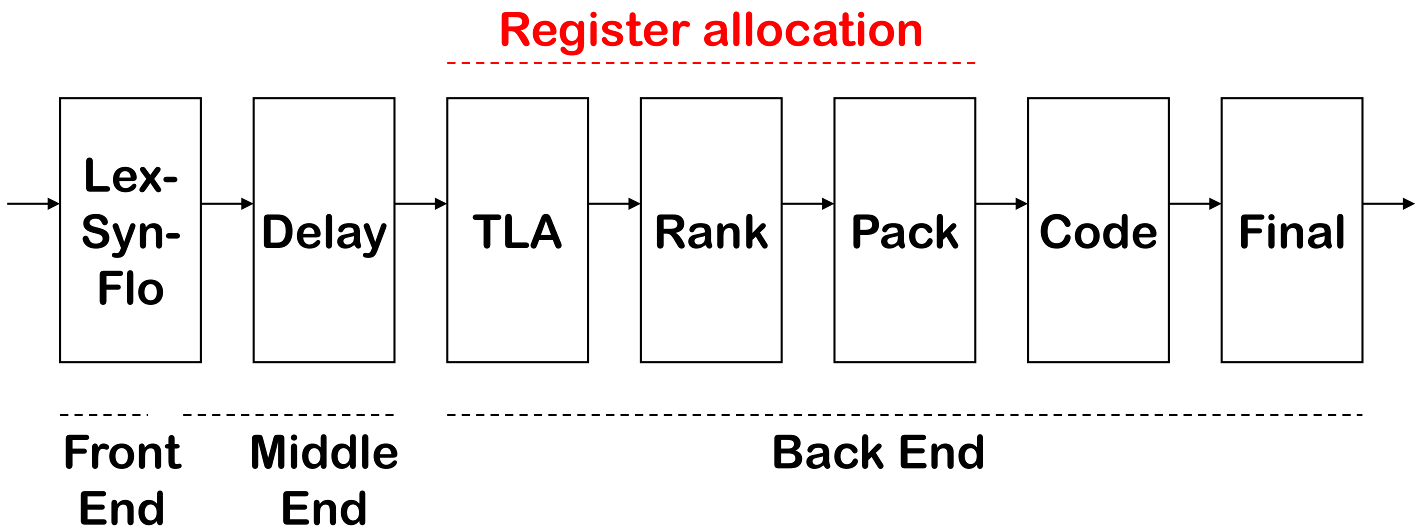


- Used low-level IR (quads), identified loops with dominators
- Focused on optimizing loops ("inside out" order)

  Passes are familiar today
- Simple front end, simple back end for IBM 370

Classic Compilers

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



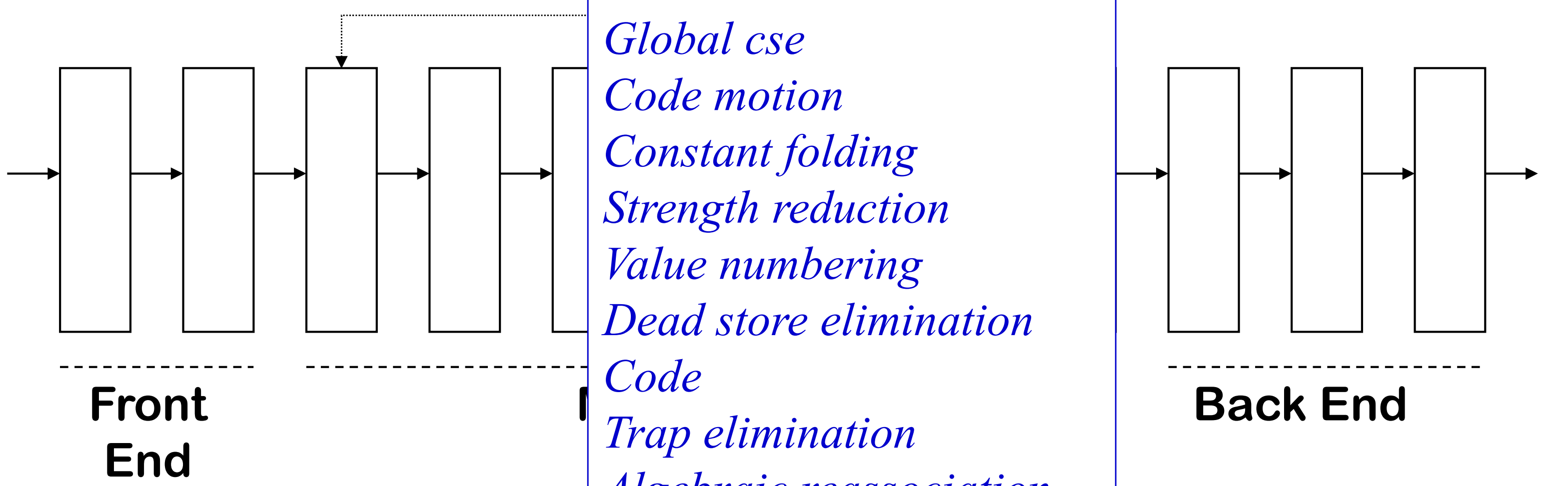
- The great compiler for the PDP-11
- Seven passes in a fixed order
- Focused on code shape & instruction selection

  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

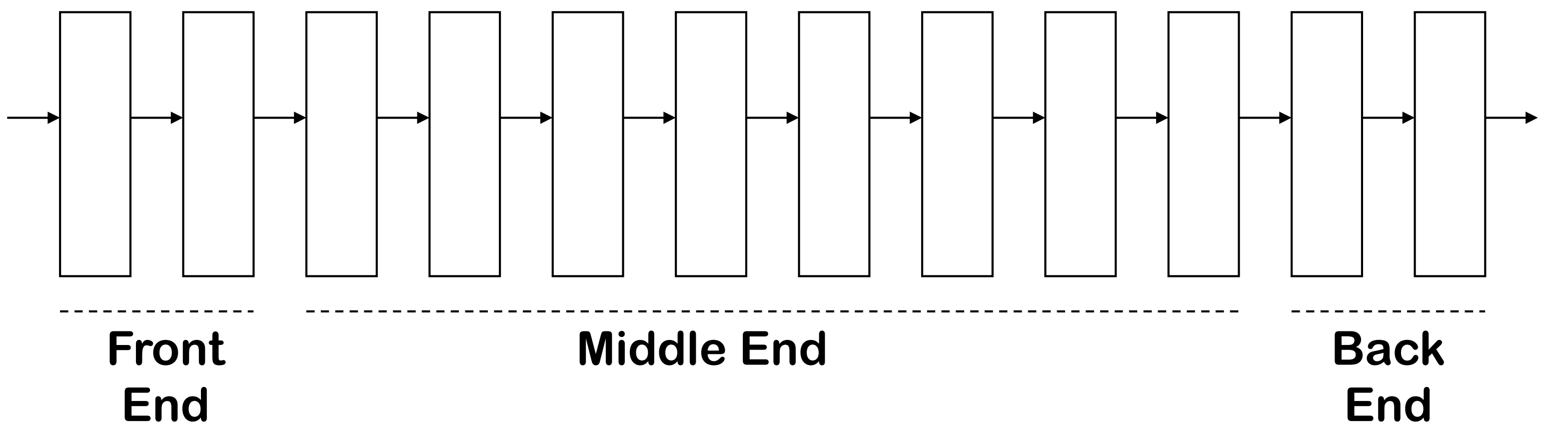
  Below machine-level IR

Multi-level IR has become common wisdom

*

Classic Compilers

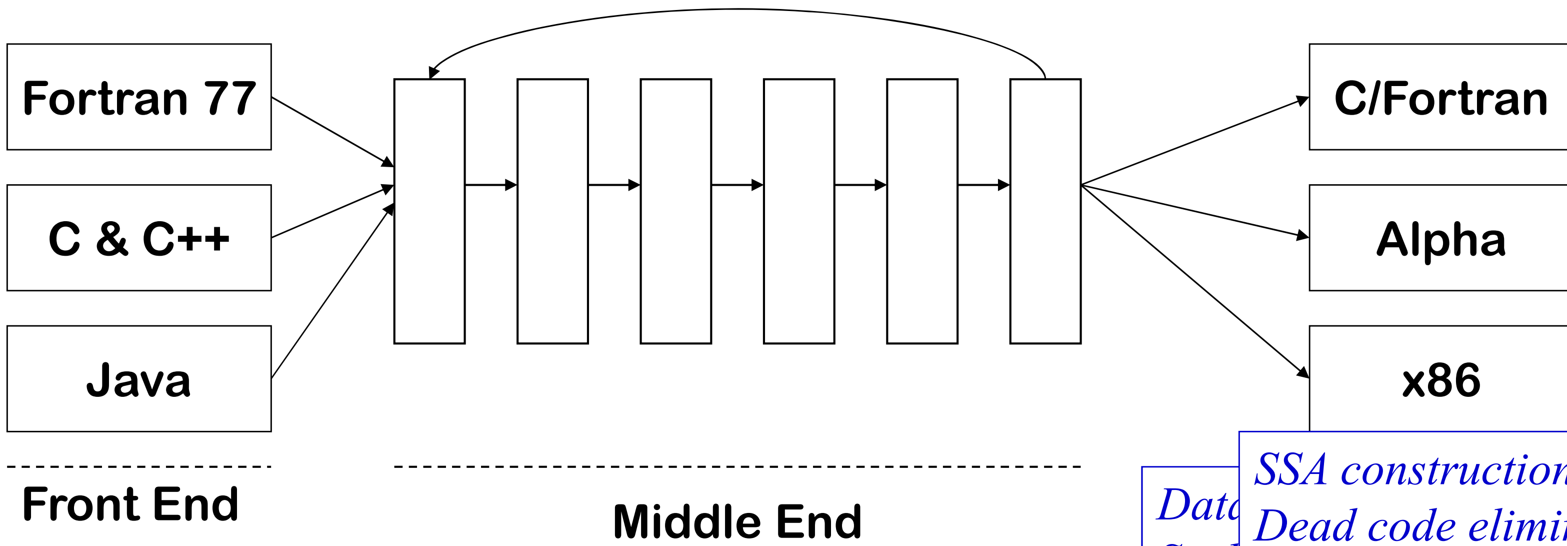
1986: HP's PA-RISC Compiler



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

Classic Compilers

1999: The SUIF Compiler System



Data
Scal
Redu
Poin
Affin
Bloc
Cap
Virtu
Garbage collection

SSA construction
Dead code elimination
Partial redundancy elimination
Constant propagation
Global value numbering
Strength reduction
Reassociation
Instruction scheduling
Register allocation

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

- Instructor: Seth Copen Goldstein

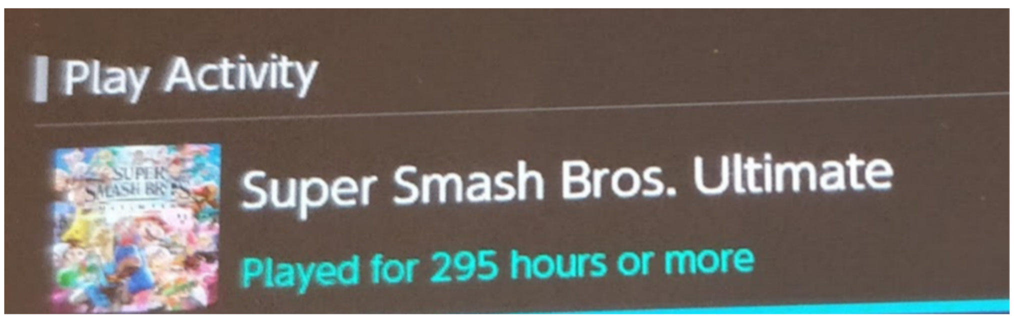
  Office hours: Thur 1pm-3pm (zoom link

- Research
  - Concurrent Systems (Parallel, Distributed, …)
  - Architecture/Compilers
  - Future of Work and Monetary Systems (BoLT)
- Teaching
  - 15-411/611 Compiler Design
  - 15-319/619 Cloud Computing
  - 15-213 Introduction to Computer Systems

Would people prefer Tue after class & Wed instead?

Pranav Kumar

- OCaml
- Super Smash Bros. Ultimate



(this is from last year dw)

Komal Dhull

- OCaml
- Likes watching sunrises/sunsets

Albert Gao

- C++
- Currently falling in love with but failing at crypto puzzles

(1, 219344805775495840321092795942973643177)

(2, 833795613474904201786431853216587596166)

(3, 1843352430976298313162458874898544815365)

Henry Nelson

- OCaml
- I like reading (hcnelson.com/books.html), board games, and tennis.

Sheng Xu

- OCaml
- big fan of Liverpool

Will Zhang

- C++
- Love traveling (particularly Switzerland

Stephen McIntosh

Rust

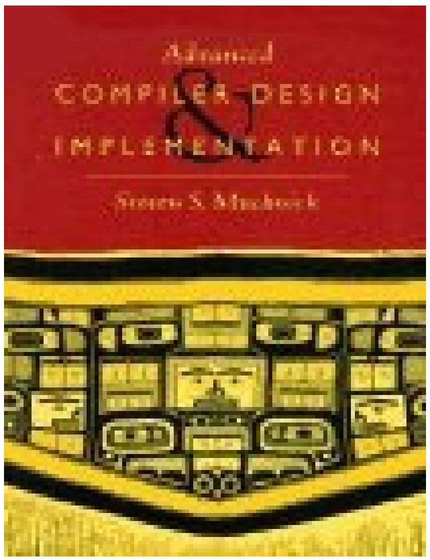
Communication and Resources

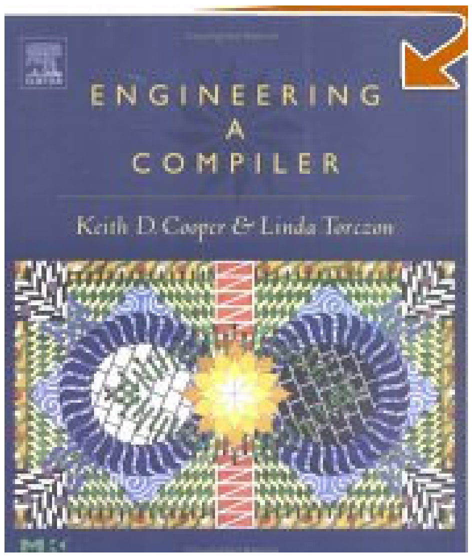
- **Lecture:** Tue/Thu 8:00-9:20am at Zoom
- **Recitation** A: Fri 2:40 - 3:20pm, Weh 7500
  B: Fri 4:00 - 4:50pm, DH 1212
  C: Fri 5:20 - 6:10pm, GHC 4401

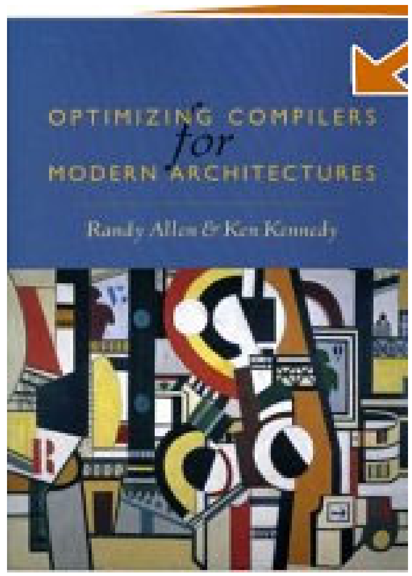
Trying to get this changed. Will update if so.

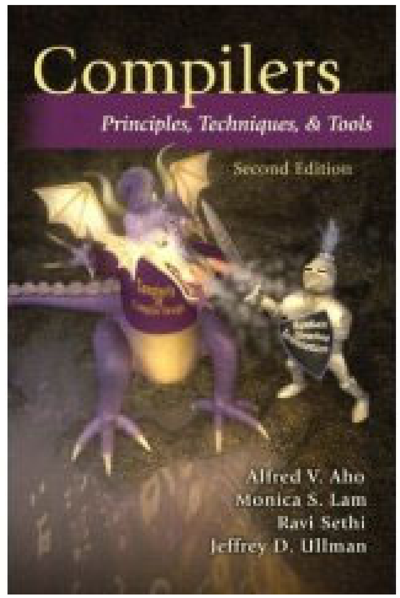
- **Website:** http://www.cs.cmu.edu/~411
- **Piazza:** Enroll from website
- **Lecture notes:** Will be available after the lecture
- **Textbook:** Andrew Appel - Modern Compiler Implementation in ML

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

- Attend lectures
  - Lecture notes are only supplementary material

No exams.

- 6 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: 50 points
  - Project proposal for a Lab 6 project: 50 points
  - Lab 5-6: each 150 points (total 300 points)

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 50 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)
  - Lab 6: code and paper (you choose)

Auto graded.

TA graded.

- Code review (50 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

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 9/7.

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 Today

Before Sunday

Until Monday 9/7

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
- Go through the questionnaire and compare your answers

That's not necessarily your best friend.

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, **Rust**, and **C++**
- When picking a language also consider the availability of parser generators and libraries

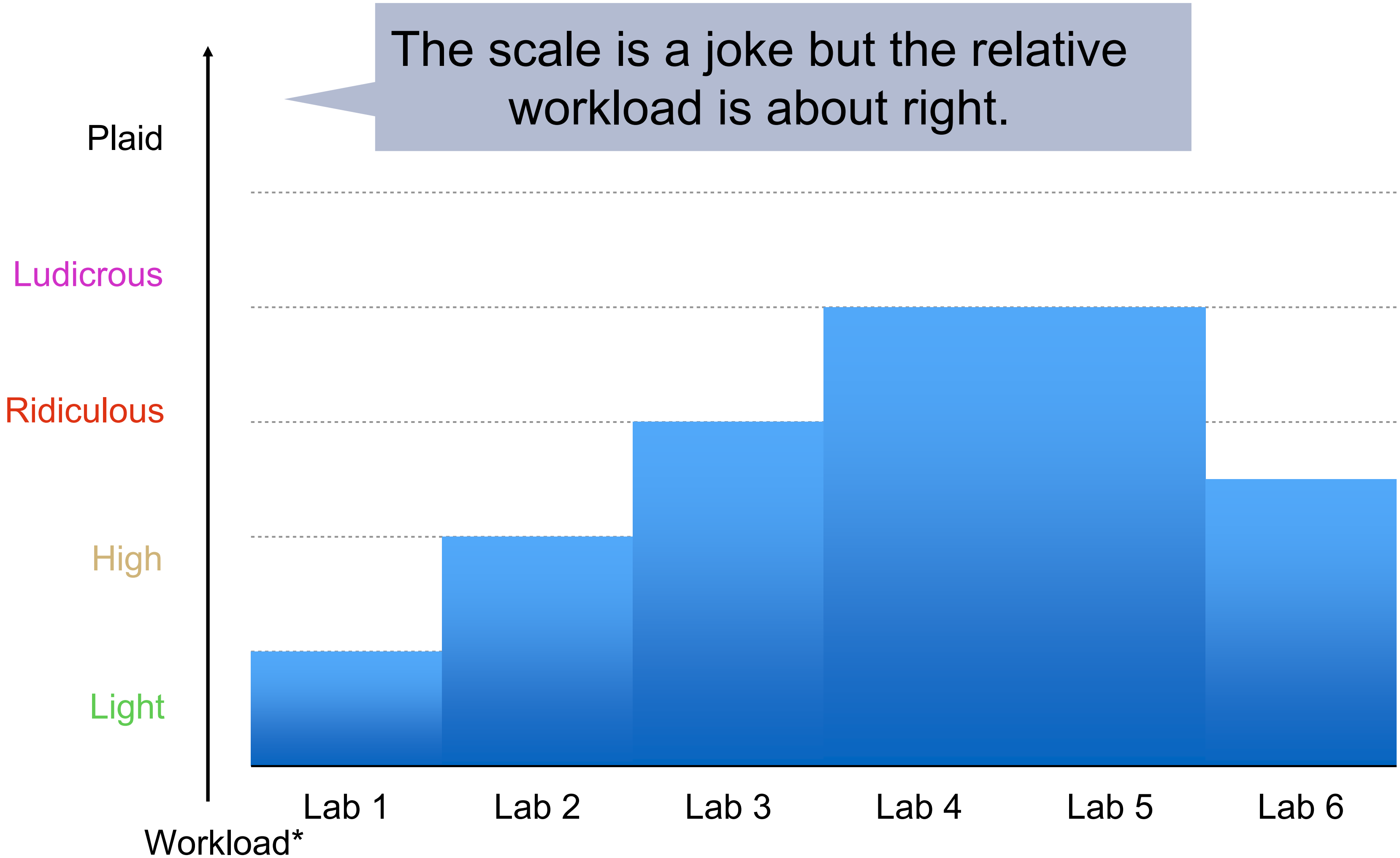
Logistics

- Assignments are submitted via Gradescope
- Labs are submitted via GitHub
  - 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 Notolab
  - 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

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

Workload Over the Semester



* scale from the movie Spaceballs.

This Year's Theme

- Solresol
- Volapük
- Spokil
- Mundolinco
- Bolak
- Mondial
- Romanid
- Poliespo
- Romániço
- Europanto
- E-Prime
- Medefaidrin
- Loglan
- Sindarin
- Nadsat
- Interlac
- Enchanta
- Trigedasleng
- …

Deadlines and Academic Integrity

- Deadlines are midnight (after class); being late results in a late day
  - You have six (6) late days for the labs (see details online)
  - You have three (3) late days for the assignments (details online)
- Talk to 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"