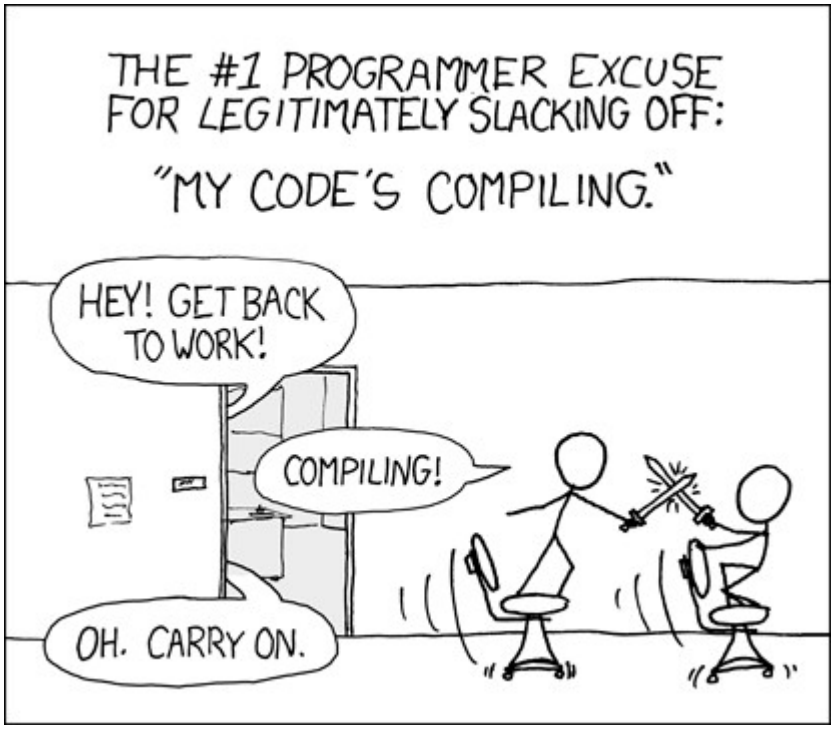
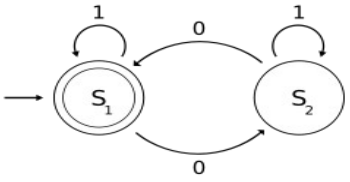
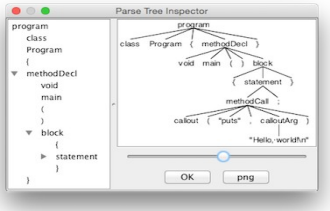
CS 432
Fall 2026

Mike Lam, Professor





Compilers



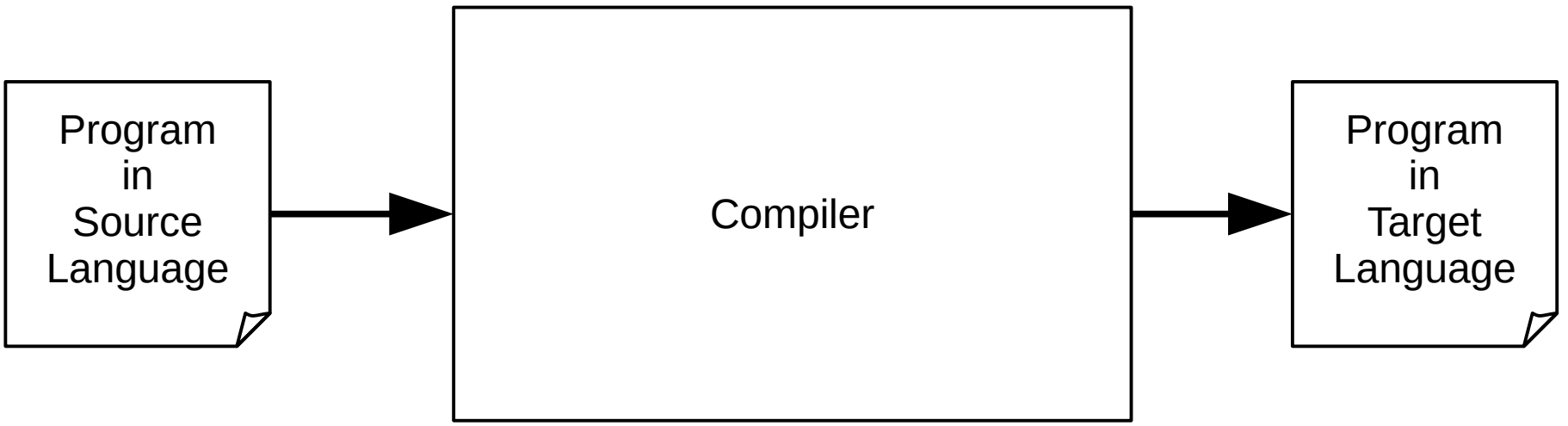
Advanced Systems Elective

Discussion question

- What is a compiler?

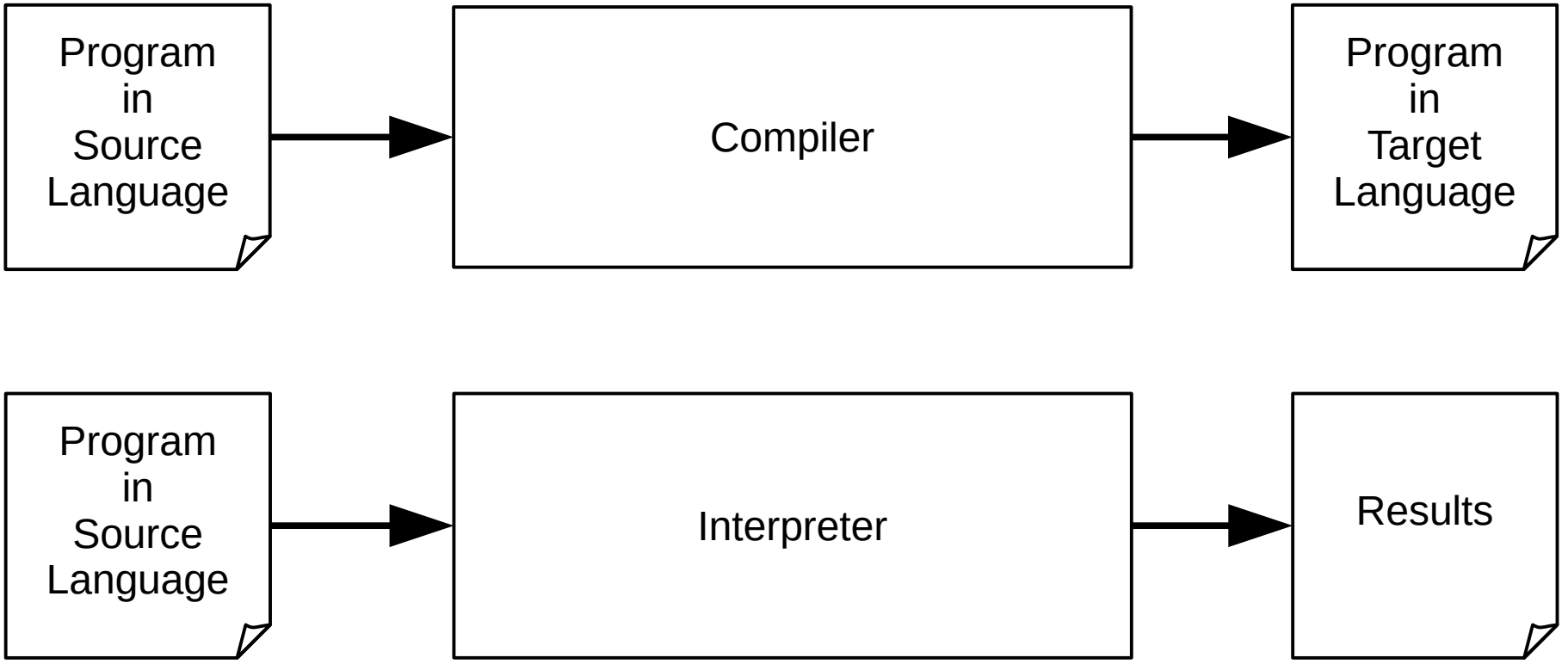
Automated translation

- A compiler is a computer program that **automatically translates** other programs from one language to another
  - (usually from a *human-readable* language to a *machine-executable* language, but not necessarily)



Automated translation

- Often contrasted with interpretation
  - A property of an implementation, not the language



Rhetorical question

- Why should we study compilers?
  - *(besides getting systems elective credit...)*

Compilers: a convergent topic

- Data structures
  - CS 240
- Architectures, machine languages, and operating systems
  - CS 261, CS 450
- Automata and language theory
  - CS 327, CS 430
- Graph algorithms
  - CS 327
- Software and systems engineering
  - CS 345, CS 361
- Greedy, heuristic, and fixed-point algorithms
  - CS 452

Reasons to study compilers

- Shows many areas of CS combining to solve a difficult and complex problem (automated language translation)
- Bridges theory vs. implementation gap
  - Theory informs system development
  - Everything supports learning how to build a compiler
- Practical experience with large(er) software systems
  - My master copy is over 5K LOC
  - You will be re-implementing it this semester
  - Need to address some software engineering concerns

Course objectives

- Identify and discuss the technical and social challenges of building a large software system such as a compiler.
- Develop and analyze formal descriptions of computer languages.
- Apply finite automata theory to build recognizers (lexers) for regular languages.
- Apply pushdown automata theory to build recognizers (parsers) for deterministic context-free languages.
- Evaluate the role of static analysis in automated program translation.
- Apply tree traversals to convert a syntax tree to low-level code.
- Discuss the limitations that an architecture or execution environment places on the generation of machine code.
- Describe common optimizations and evaluate the trade-offs associated with good optimization.

LEARN HOW TO BUILD A COMPILER

Other objectives

- Practice designing a large software system
- Practice contributing to a large code base
  - Understanding and using interfaces and data structures
  - Debugging across modules
- Practice reviewing code written by others
- Experiment with AI coding tools
- Think deeply about how theoretical formalisms inform practical implementations
- Fully understand and harness recursion (finally!)

Evolution of CS 432

- Fall 2015 - special topics (CS 480)
  - Adaptation of CS 630 (graduate course) taught in Spring 2015
- Fall 2016 - first time taught as CS 432
  - First time teaching CS 261 as well
- Fall 2017
  - Expanded test suite significantly, added type systems and lambda calculus
- Fall 2018
  - Added Y86 translator to “close the loop” with CS 261, removed lambda calculus
- Fall 2019 (two sections)
  - Removed reflection paper assignments, switched to Dragon book for LR parsing
- Fall 2020
  - Re-wrote entire project in C (w/ re-worked grading scheme)
  - Added hybrid virtual/in-person office hours
- Fall 2022
  - Official support for VS Code + Remote SSH development platform
- Fall 2023
  - First semester allowing AI-assist tools on projects
- Fall 2024
  - Split midterm into two smaller exams
- Fall 2026
  - **Complete project re-design** (minimal required framework)

Semester-long project

- Compiler for "Decaf" language
  - Implementation in C11 w/ Makefile and integrated test suite
  - Compiles Decaf programs to ILOC & Y86
  - Five **mandatory** checkpoints P1-P5: "pieces" of the full system
  - Projects must be completed in teams of 2-4
    - **Must reach consensus on AI use**
  - Include AI disclosure and prompt portfolio with every submission
  - Grading: automated + manual inspection
    - Functionality tiers (like in 261), but most test cases are NOT provided in advance
      - Grade point conversion: A = 100, B = 85, C = 70, D = 50, F = 25
    - Manual inspection of error message (P1-P3) and AI disclosure/portfolio
    - For P2-P5, **previous checkpoint's test suites are re-run** as part of the grading
  - Individual graded code reviews due a week later (P0/P2/P4)
  - Summative in-person oral examination at end of semester

F26: AI use on projects

- AI-assisted tools are allowed on the projects this semester, but each group must commit to a shared AI-use level before P1 and disclose its tool access.
- You must also include a statement that discloses the extent of your use of AI-assist technologies on that assignment, and if you used AI, a curated portfolio of logs and/or prompts you used.
- This policy is experimental, and may be revised mid-semester; changes will be announced on Canvas

AI use levels

0) None (discouraged)

1) Tutor: no code at all, just concepts or errors

2) Reviewer: reads code for debugging assistance

3) Autocomplete: line- or snippet-level code additions

4) Collaborator: larger code additions via chat

5) Agentic (discouraged)

More details are in the AI agreement document available on Canvas

AI use guidelines

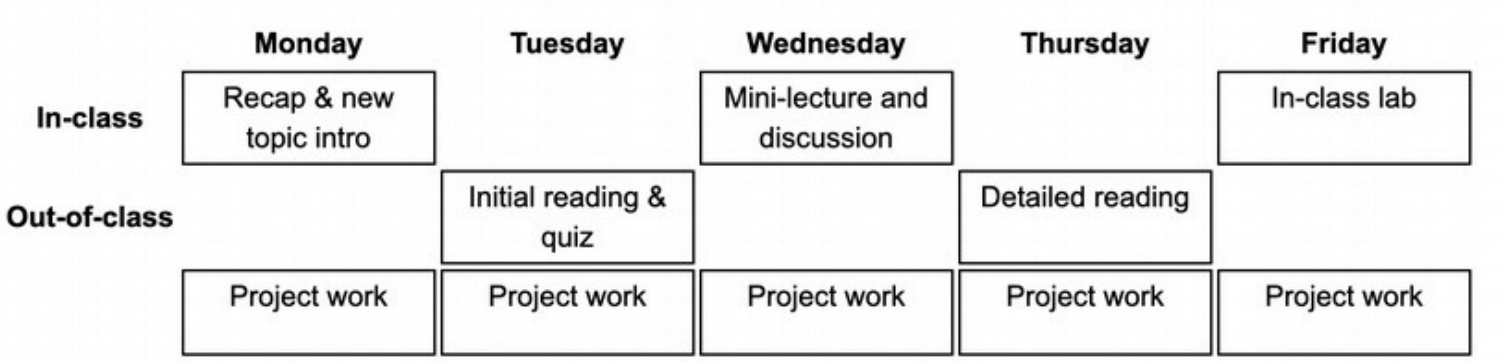
- You must be able to fully explain any work committed under your name, and own all work done by your group
- You must credit every tool and source used
  - I used Claude (mostly Opus 4.8) in preparing course materials for this semester: to refactor the project, update documentation, and audit slides and labs
- You must use a shared, private Git repository with me as a collaborator to verify individual contributions
- The goal is not just to write a compiler, the goal is to **learn how** to write a compiler
- “Who is doing your thinking for you?”

Aside: project submission

- Issue: Canvas is not a great system for code reviews
- Total of four (4) possible things to submit:
  1) **Submit project on stu (for grading, similar to 261)**
  2) Submit .c file on Canvas for P0/P2/P4 (for code reviews)
  3) **Submit code reviews on different Canvas assignment (for grading)**
  4) Send code reviews to reviewees via Canvas message (for their benefit)
- Due dates:
  - Original project deadline: **#1** and #2
    - Note: two projects (P2 and P4) also include “milestone” deadlines a week before
    - Milestone deadlines are optional; I will provide feedback to anyone who submits
  - One week after project deadline: **#3** and #4 (code reviews)
    - Note: no code reviews for (P1/P3/P5)

Course format

- Website: https://w3.cs.jmu.edu/lam2mo/cs432/
  - Make sure you're using the right year's website!
- Weekly schedule (many weeks – some may vary)
  - Labs must be completed in person (excused absences only)

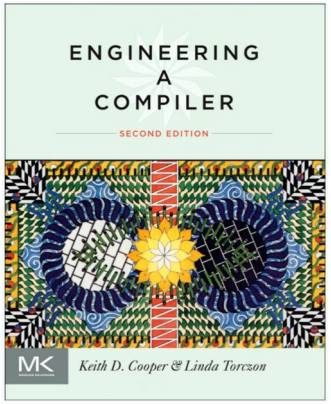


	Monday	Tuesday	Wednesday	Thursday	Friday
In-class	Recap & new topic intro		Mini-lecture and discussion		In-class lab
Out-of-class		Initial reading & quiz		Detailed reading	
	Project work	Project work	Project work	Project work	Project work

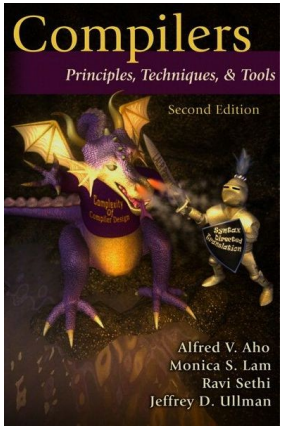
- *Formative* vs. *summative* assessment
  - Formative: quizzes and labs (together only 20% of final grade)
  - Summative: projects (25%) and in-person written exams (55%)
  - **Don't trust your Canvas grade until mid-semester! (and even then it is just an approximation)**

Course text(s)

- **Engineering a Compiler, 2nd Edition**
  - Keith Cooper and Linda Torczon
  - Available online through JMU Libraries
  - Reserve copy at Rose library
  - Caution: newer 3rd edition available (some chapters re-numbered)



- **Compilers: Principles, Techniques, & Tools, 2nd Edition**
  - Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
  - “The Dragon Book” (classic text on compilers)
  - One chapter scanned; posted under “Files” on Canvas



- Decaf/ILOC references and type systems reading
  - PDFs on website
- Design patterns reading from GoF book
  - PDF on Canvas

Communication

- Email: `lam2mo@jmu.edu`
  - Response within 24-48 hours, but no guarantees on weekends or on project deadlines
- Office hours posted on Canvas
  - **NOTE:** I have two offices this semester
    - King 227 is my CS faculty office (TuTh)
      - Drop-in office hours and appointments
    - King 365 is my CISE associate dean office (MWF)
      - Meetings by appointment only

Class policies

- If you test positive for a respiratory illness or are consistently coughing and/or sneezing, **please stay home**
  - Contact me ASAP regarding missed class
  - If you feel a bit ill but well enough to attend class (and are NOT consistently coughing and/or sneezing), please consider wearing a surgical or N95/KN95 mask to protect others
- Feel free to bring laptops to class
  - Please do not cause distractions for others
- These policies may change
  - Changes will be announced via Canvas message

Whew!

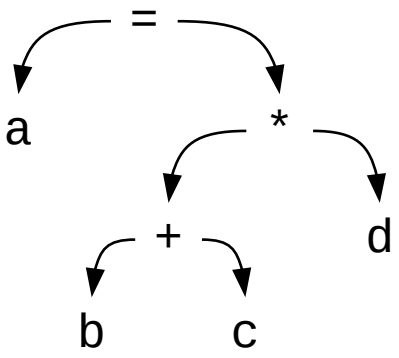
- Questions?

Compiler rule #1 (from EAC2e)

- "The compiler must preserve the *meaning* of the program being compiled."
  - What is a program's *meaning*?
  - Assume well-defined behavior (not always the case!)

Intermediate representation

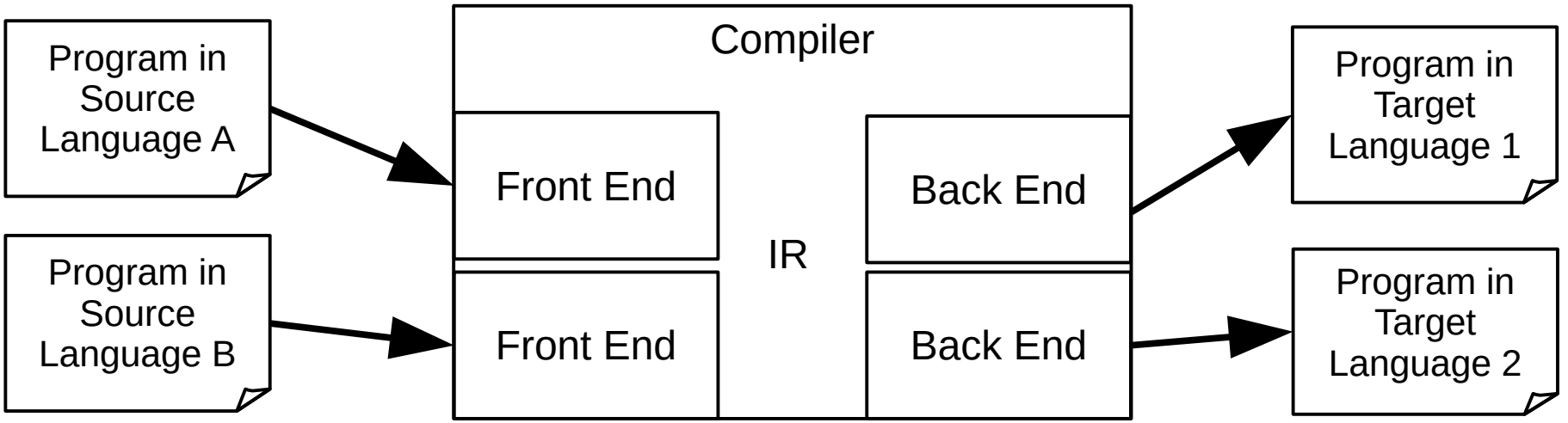
- Compilers often encode aspects of a program's meaning using an intermediate representation (IR)
  - Tree- or graph-based: abstract syntax tree (AST), control flow graph (CFG)
  - Linear: three-address code (e.g., LLVM IR), Java bytecode, intermediate language for an optimizing compiler (ILOC)




```
load b → r1
load c → r2
add  r1, r2 → r3
load d → r4
mult r3, r4 → r5
store r5 → a
```

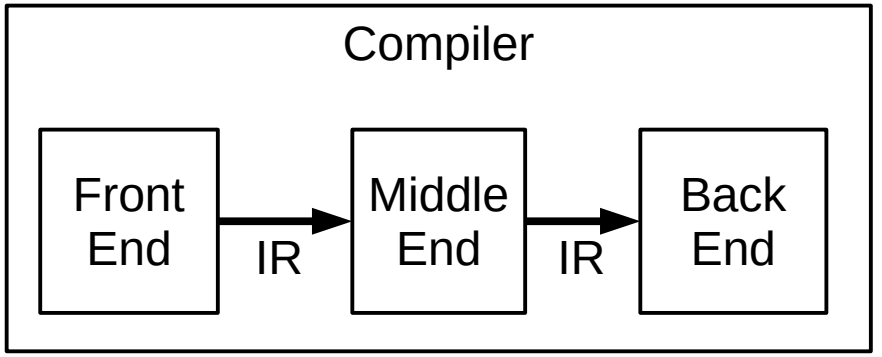
Standard compiler framework

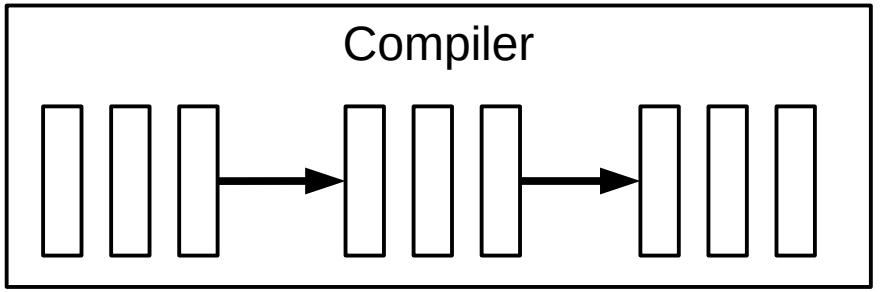
- Front end: understand the program (src → IR)
- Back end: encode in target language (IR → targ)
- Primary benefit: easier to add support for new languages or architectures



Modern compiler framework

- Front-end passes
  - Scanning (lexical analysis)
  - Parsing (syntactic analysis)
- Middle-end passes
  - Static/semantic analysis
  - IR code generation
  - IR optimization
- Back-end passes
  - Instruction selection
  - Machine code optimization
  - Register allocation
  - Instruction scheduling
- Modern many-pass / micropass approach
  - Dozens or hundreds of passes (`https://llvm.org/docs/Passes.html`)



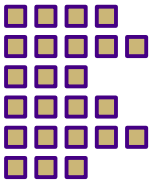


Our Decaf compiler

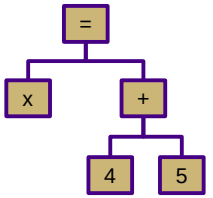
Source code

```
def int main() {
    int x;
    x = 4 + 5;
    return x;
}
```

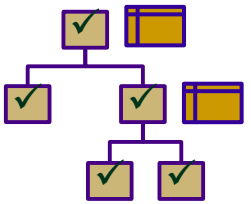
Tokens



Syntax tree



Checked AST + Symtables

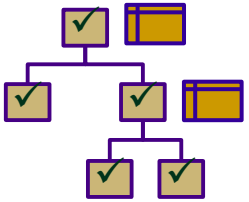


Lexing (P1)

Parsing (P2)

Analysis (P3)

Checked AST + Symtables



ILOC

```
main:
  loadI 4 => r1
  loadI 5 => r2
  add r1, r2 => r3
  i2i r3 => RET
```

Allocated ILOC

```
main:
  loadI 4 => r0
  loadI 5 => r1
  add r0, r1 => r0
  i2i r0 => RET
```

Y86

```
irmovq $4, %rcx
irmovq $5, %rdx
addq %rcx, %rdx
rrmovq %rdx, %rax
ret
```

Run via ILOC simulator

Run via yas + 261 P4

Code Generation (P4)

Register Allocation (P5)

Machine Code Generation (provided)

Compiler rule #2 (from EAC2e)

- The compiler should *help* the programmer in some way
  - What does *help* mean?

Discussion question

- What would be your design goals for a compiler?
  - E.g., what functionality or properties would you like it to have?
  - (Besides rule #1 – correct translation)

Compiler design goals

- Optimize for fast execution
- Minimize memory/energy use
- Catch software defects early
- Provide helpful error messages
- Run quickly
- Be easily extendable

Differing design goals

- What differences might you expect in compilers designed for the following applications?
  - A compiler used in an introductory programming course
  - A compiler used to build scientific computing codes to run on a massively-parallel supercomputer
  - A compiler that targets an embedded sensor network platform
  - A just-in-time compiler for running server-side user scripts
  - A compiler that targets a number of diverse systems

- Optimize for fast execution
- Minimize memory/energy use
- Catch software defects early
- Provide helpful error messages
- Run quickly
- Be easily extendable

Decaf language

- Simple imperative language similar to C or Java
- Example:

```
// add.decaf - simple addition example

def int add(int x, int y)
{
    return x + y;
}

def int main()
{
    int a;
    a = 3;
    return add(a, 2);
}
```

```
$ ./decaf-ref add.decaf >add.iloc
$ ./isim add.iloc
RETURN VALUE = 5
```

Before Friday

- Readings
  - "Engineering a Compiler" (EAC2e) Ch. 1 (23 pages)
  - Decaf reference, sections 1-4 ("Resources" page on website)
- Tasks
  - **Complete welcome survey on Canvas**
  - **Complete first reading quiz on Canvas**
  - If you have time, write some code in Decaf and test the reference compiler and simulator:
    - `/cs/students/cs432/f26/decaf-ref`
    - `/cs/students/cs432/f26/isim`
  - Look at project description for P0 (one-off project to help you learn Decaf)
  - Bring your laptop on Friday if you can, and be ready to start promptly at the beginning of class!

Upcoming college events

- September 14-15 (Mon-Tue)
  - Various career fair prep events
  - Detailed schedule sent out from CISE soon
- September 16 (Wed), 11am-3pm
  - CISE Career and Internship Fair, Festival Ballroom

Closing exhortations

- Take care of yourself
  - And if you can, someone else
  - Build (or reconnect with) a support network
  - Protect your boundaries
  - Carve out time to disconnect and rest
  - Talk to someone if things start getting overwhelming
- Have a great semester!