CIS531 — Compiler Construction

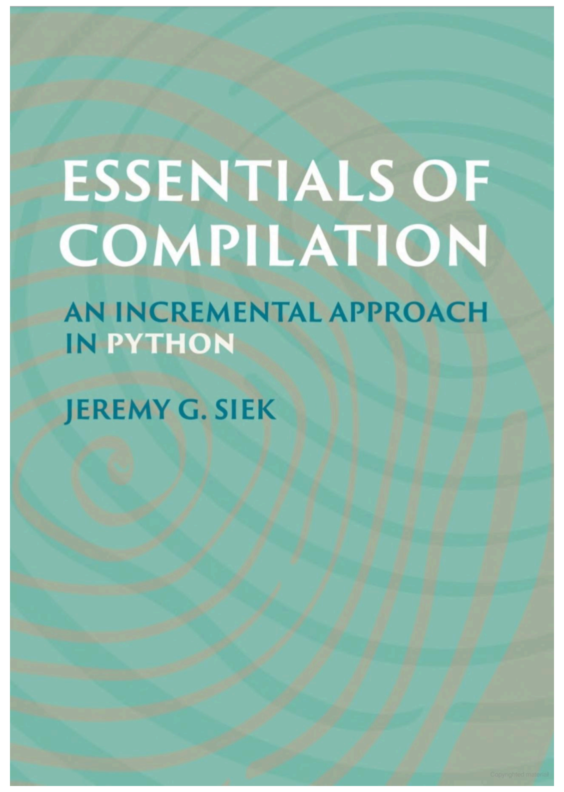
Fall 2025

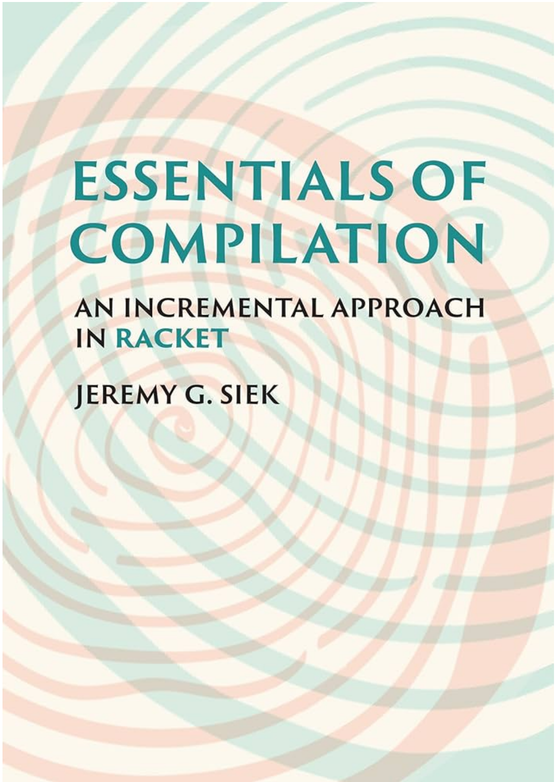
Kristopher Micinski

Syracuse University
College of Engineering
& Computer Science

Welcome to CIS531!

- Very happy to have you in class! It will be a great semester!
- This course: a tour of compiler design, along with some advanced (not *hard*!) topics
- We will be following a **free** book, which you should download:
  - There are ***several versions of the book***
    - *Project starters were rewritten (sometimes changed) by me!*
  - https://brinckerhoff.org/clements/2194-csc431/essentials-of-compilation.pdf
- Please *stay in the class* even if you think you are not prepared (talk to me if so)
  - Grading designed to be somewhat forgiving, projects done in groups (if you want only)





Why take CIS531?

- Compilers, like OS, involve nearly every aspect of computer science:
  - Algorithms (register allocation, program analysis, …)
  - Data structures (syntax trees, control-flow graphs, …)
  - Computability (grammars, automata, …)
  - Programming languages (semantics, type checking, …)
  - Systems (assembly code, ABIs, linking, …)
  - Security (stack smashing, reverse engineering, …)
  - Software Engineering (debugging, testing, specifications, …)
- ***We will talk (at least a little) about all of these!***

Why take CIS531? ...

- Great way to demonstrate to employers that you are capable of doing hard programming projects
- Distinguishing course that tests your knowledge of the rest of the curriculum
- Understand the internals of compiler design
  - Helps teach “how to learn” other languages
- Challenging (but very useful) exercise in debugging across layers
- Exposure to reading technical papers
- (Optional) learn how to work on teams to develop a large project

Grading

- There will be:
    - (50%) 4 projects — groups of up to three
    - (15%) Present a paper — I have selected papers, you will make slides
    - (30%) Exams — two written midterms
        - Not quite like traditional exams, we will talk about this
    - (5%) Participation — some incentive for you to come to class
        - Assessed via participation quizzes:
            - You will get 5% if you attend at least 2/3 of participation quizzes

Projects

- Project 1 (2 weeks) — Racket warmup, simple interpreter
- Project 2 (2 weeks) — Compiling arithmetic + variables to x86
- Project 3 (2 weeks) — Branching control-flow,
- Project 4 (~2.5 weeks) — set!, assignment conversion, procedures
- Project 5 — Final Project

Projects: Details

- You **can** work in groups of ***up to three***
  - If you don't feel confident, *please team up with someone who does*
  - If you are more confident, please consider working with someone who is a beginner
  - There are some **special rules** (more work) required for teams
- Projects coded up in either Racket or Python (starter files for both)
  - I ***strongly*** recommend Racket, and I will teach it during class
  - Also, I will give a significant amount of hints, and will code up a significant portion of the solution in class for students to follow along.
  - Racket may look a bit intimidating, but it's an easy language—I have a set of course videos which will bring you quickly up to speed if you want to learn.
- Projects graded via **autograder**

Weekly Topics (first half of course…)

- **Week 1:** Introduction, preliminaries, x86 assembly crash course
- **Week 2:** Compiling basic blocks to straight-line assembly
- **Week 3:** Register allocation, liveness analysis, interference graphs
- **Week 4:** Register allocation via graph coloring
- **Week 5:** Branching control-flow (if/cond/…)
- **Week 6:** Loops and data-flow analysis
- **Week 7:** Program analysis, lattices, abstract interpretation
    - Midterm exam here

Weekly Topics (second half of course...)

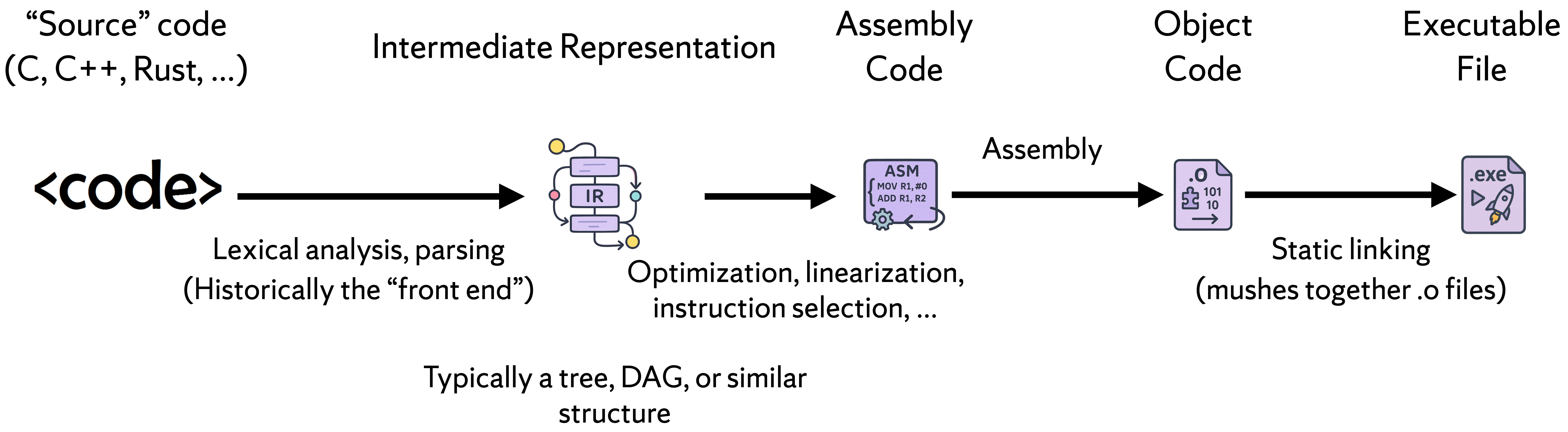
- **Week 8:** Tuples and Garbage Collection / Allocation
- **Week 9:** More GC, runtime systems
- **Week 10:** Functions, direct calls
- **Week 11:** Multiple parameters, tail calls, indirect calls
- **Week 12:** Advanced topics: Closure conversion
- **Week 13:** Advanced topics: Object orientation
- **Week 14:** Advanced topics: Security and Decompilation

Group Details

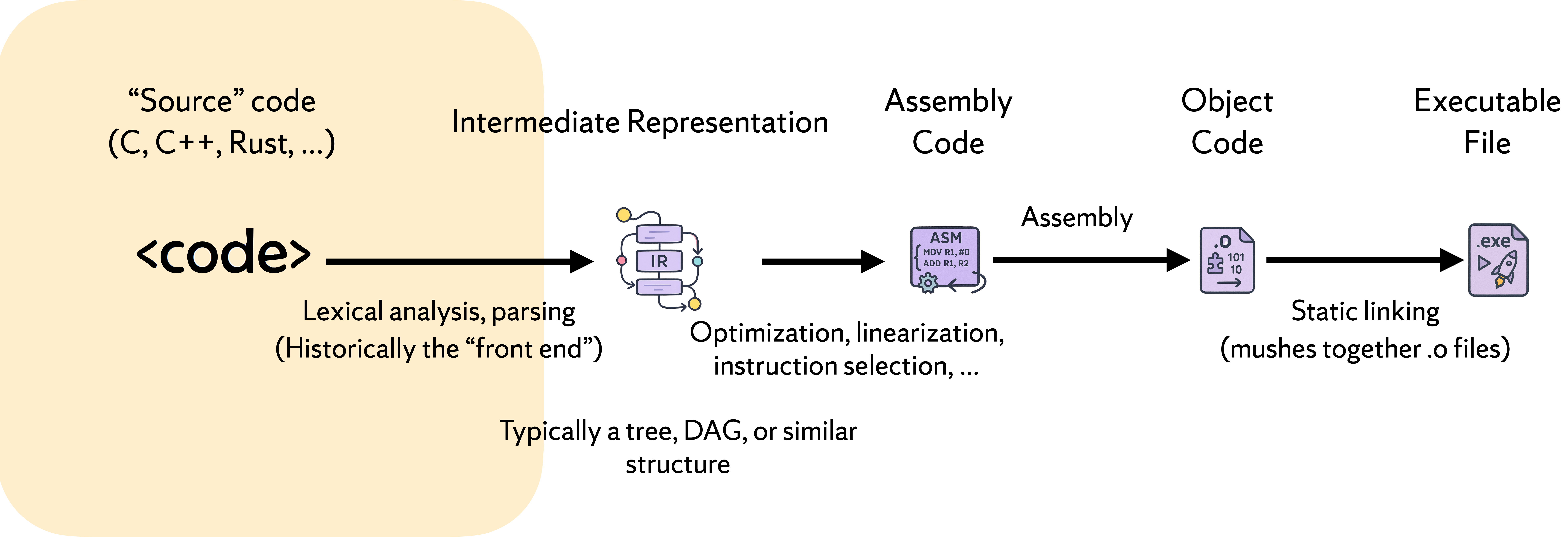
- Groups are per-project
- If you want to work in a group, you have to commit to meeting regularly
- You can work asynchronously, but you have to do the following:
  - You must spend **half** of your time writing / reviewing code synchronously, together
    - This can include debugging, or talking through concepts / white boarding
- For each group where you work as a group, each group member must submit an email to me at the end of the project saying:
  - What they personally contributed to the project?
  - Were they satisfied enough with their group's performance?
  - What things could they have done personally to improve group performance?
  - What could others in their group do to improve group performance?
  - After each project, the group must discuss: what are the top two things we can commit to doing to improve our collaboration on the *next* project (if we worked together again)?

What is a compiler?

Historically, compiler separated into “frontend” (lexing, parsing, etc…), “middle-end” (semantic analysis, IR construction, some optimization), and “backend” (dumping to machine-specific IR, assembly, linking)

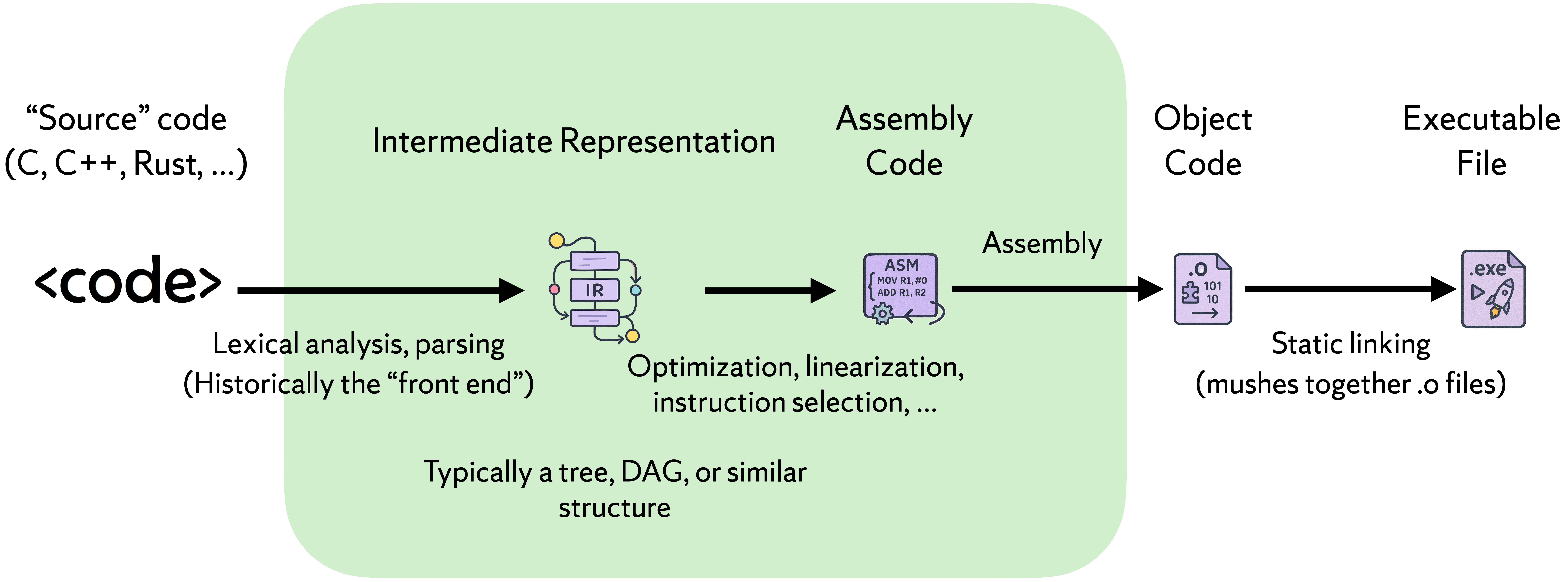


We will spend only a little time on the frontend, talking about parsing/lexing for only a few lectures



We will spend **almost all of our time** here. Going from the assembly code to object code is tedious and not very intellectually interesting: lots of assemblers / linkers exist already

Linking is interesting and hard, but we will not talk about it too much (not my expertise)



Compilers are designed in *passes*

Closeness to machine code

- Lexical analysis — separate out individual *tokens* from a stream
- Syntactic analysis (parsing) — build AST / parse tree
- Semantic analysis (name resolution, typing, etc…)
- Intermediate Representation (IR) construction
- Machine-independent optimization passes (possibly many) on IR
- Machine-dependent lowering (addressing modes, vector ops, etc.)
- Register allocation, spilling, etc.
- Instruction scheduling, peephole optimization, etc.
- Assembly / object code emission
- Linking and relocation
- Metadata emission (DWARF, …) — GC metadata emission, debugging symbols, etc.

The Waterfall (Dragon Book) Approach: Pros/Cons

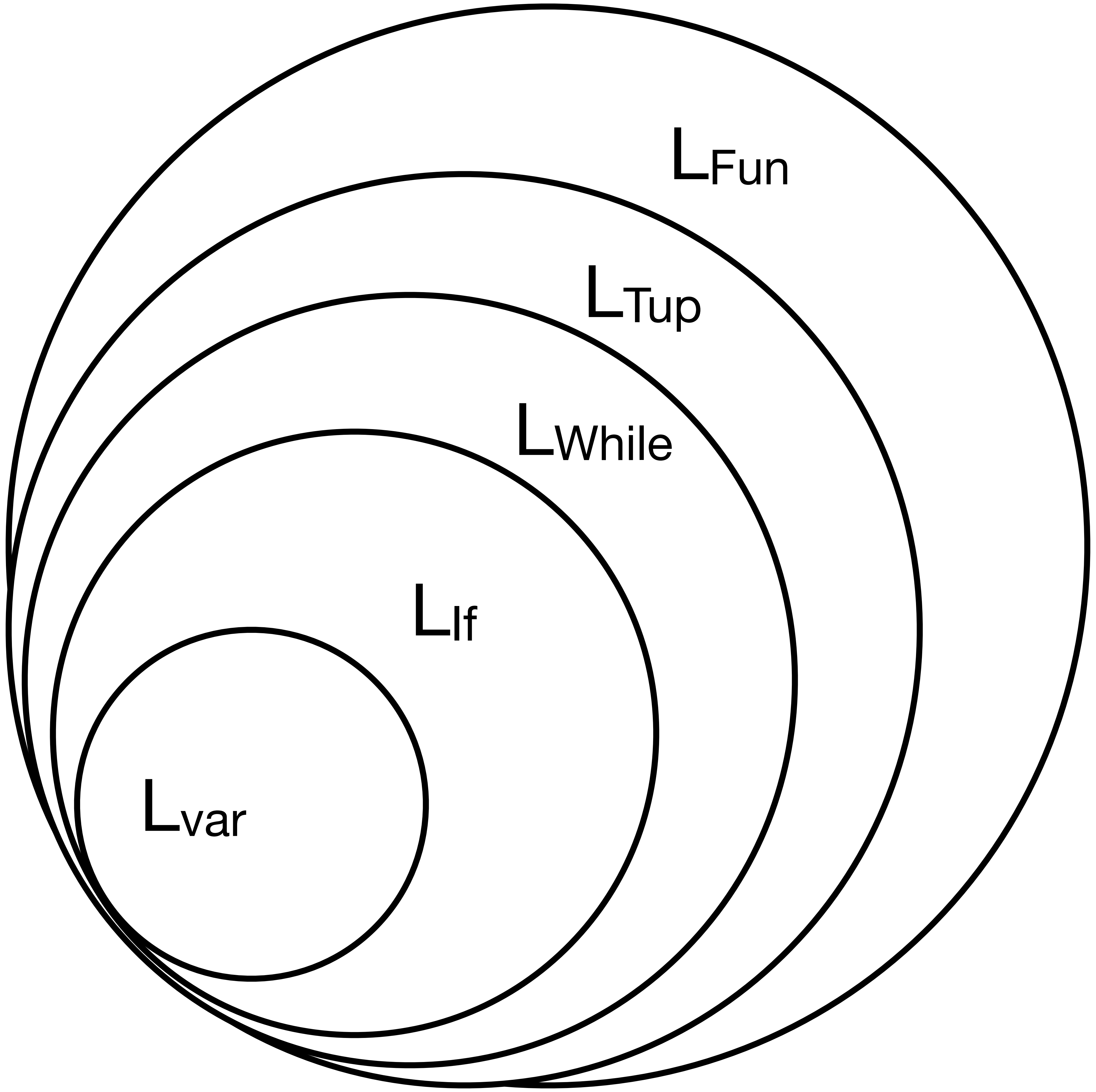
- Many classes take a top-down approach, but this has several drawbacks:
  - Lots of work, need to know whole language up-front, redoing lots of preexisting passes
  - Hard to beat modern C/C++/Rust compilers in practice (compile to C/C++/Rust?)
  - *The payoff is a long time down the road: can't run* ***any*** *program until the very end*

- Several possible solutions:
  - Write interpreters for intermediate IRs (unsatisfying, seeing x86 is exciting!)
  - Compile to higher-level language (C/C++/Rust)
    - Modern compilers (clang, etc.) are ***amazing*** in practice
    - Downside: you don't learn to truly compile to assembly, which is still relevant
    - But a very popular solution in practice (e.g., Soufflé Datalog)
  - Start in the middle (ignore parsing) by using an easily-parsable thing (e.g., JSON)
    - Parsing is interesting but orthogonal to the semantics

We take an *incremental* approach

- My course follows Jeremy Siek's approach (replicated at many other unis!)
- Unlike the traditional approach, we build a **whole** compiler in **every** project
- This way, you can use the compiler (completely) at every point
  - The **language** is the thing that grows, rather than the compiler
- Also, you **have** to develop a good debugging strategy (for assembly) early
  - It is crucial to acquire the skills necessary to debug the IR at multiple stages!

A nested tower of increasingly-powerful languages



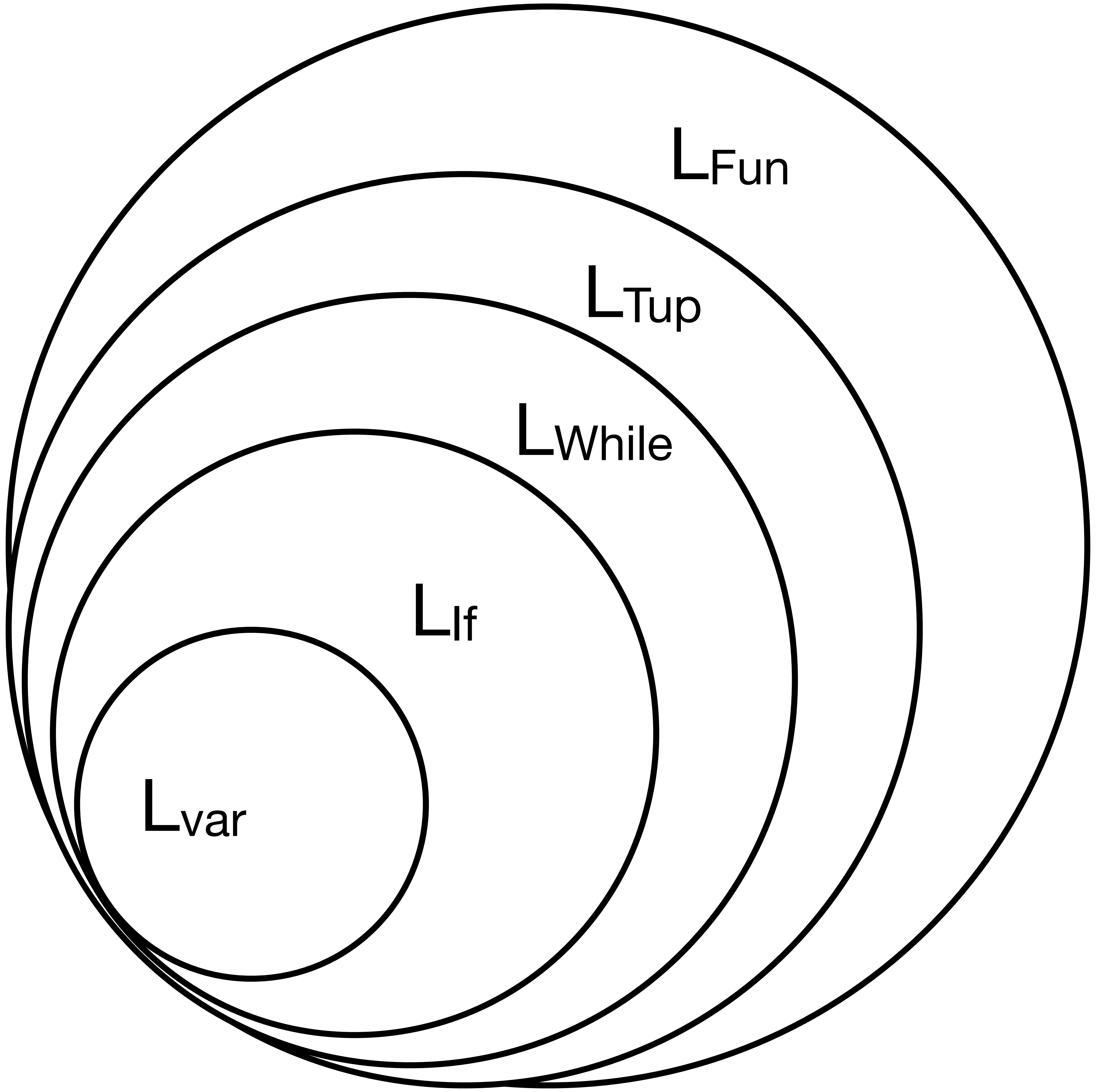
(Stolen from Jeremy Siek's slides)

Order of language the book presents:

- LVar — integer +/-, straight-line variables
- LIf — branching control
- LWhile — loops
- LTup — heap-allocated data, GC
- LFun — functions
- Some more…

We will follow roughly this structure, some variation toward the end (final project)

We have Racket (Scheme) and Python variants



Racket variant…

```
(let ([x (* 5 3)])
 (let ([y (+ x (* x x))])
   (+ (read) 5)))
```

Python variant…

```
x = 5 * 3
y = x + x*x
read_int() + 5
```

The key insight is that the ***specific syntax doesn't really matter***. We give Racket (Scheme) and Python variants, but only the surface syntax differs

Racket variant…

```
(let ([x (* 5 3)])
 (let ([y (+ x (* x x))])
   (+ (read) 5)))
```

We provide starter code and test harnesses in Python and Racket

Python variant…

```
x = 5 * 3
y = x + x*x
read_int() + 5
```

I strongly recommend coding in Racket! I will offer substantial help (both in class / OH)!

So what is the *language*?

- The focus of the course is ***not functional programming***!
  - If you are concerned that you will only be learning useless FP stuff, do not worry—the principles apply across languages / paradigms!
  - We have test cases in Python ***and*** Racket (just syntactic differences!)
  - I am *also open to you doing the class in other languages* if you have a strong preference—though we **need to have a conversation beforehand** so I can grade your solutions
- I will program in Racket—not because it is a functional language—because it is the ***best tool for the job***
  - *Do not worry* if you don't know Racket—I will keep it very simple, and we will go slow at first
  - *All of the topics in the course translate directly to Python, Rust, Java, C++, …!*

Check your Knowledge

A few questions after this lecture:

- Is attendance required? If so, how is it measured?
- What language will be used for projects?
- Are we using a book? If so, do you have to pay for it?
- Can you work in groups? If so, what are the parameters?

Ask me (kkmicins@syr.edu) if you want to discuss answers after class