Lecture 26

CS 131: COMPILERS

Final Exam

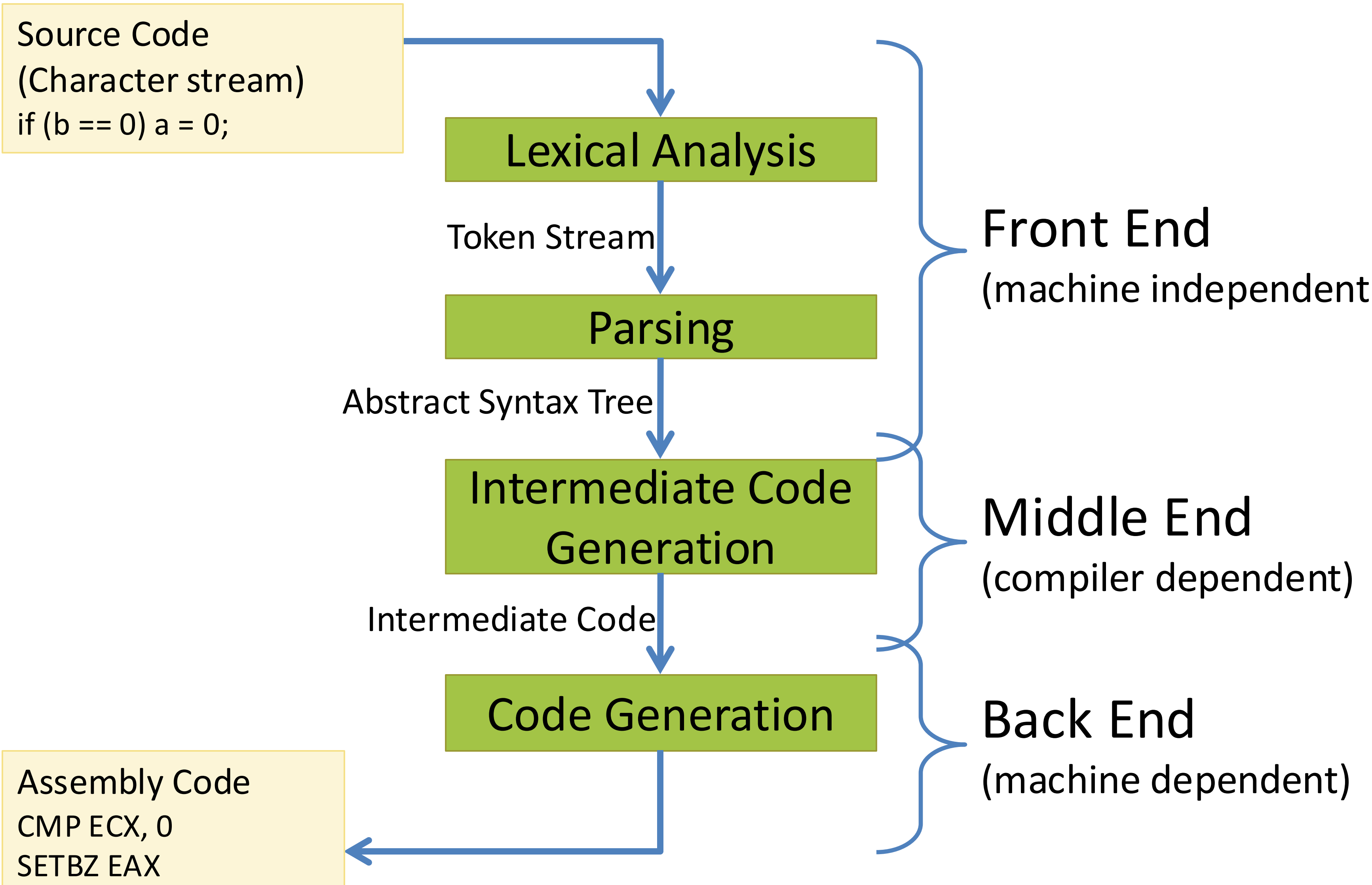
- In class, Jan 2nd

- Will mostly cover material since the midterm
  - Starting from Lecture 14
  - Lambda calculus / closure conversion
  - Scope / Typechecking / Inference Rules
  - Objects, inheritance, types, implementation of dynamic dispatch
    (de-emphasized, since we didn't cover it thoroughly)
  - Basic optimizations
  - Dataflow analysis (forward vs. backward, fixpoint computations, etc.)
    - Liveness / constant propagation / alias analysis
  - Graph-coloring Register Allocation
  - Control flow analysis
    - Loops, dominator trees
- Basics before midterm: X86Lite, LLVMLite, lexing, and parsing.

- One, letter-sized, double-sided, hand-written "cheat sheet"

COURSE REVIEW

(Simplified) Compiler Structure

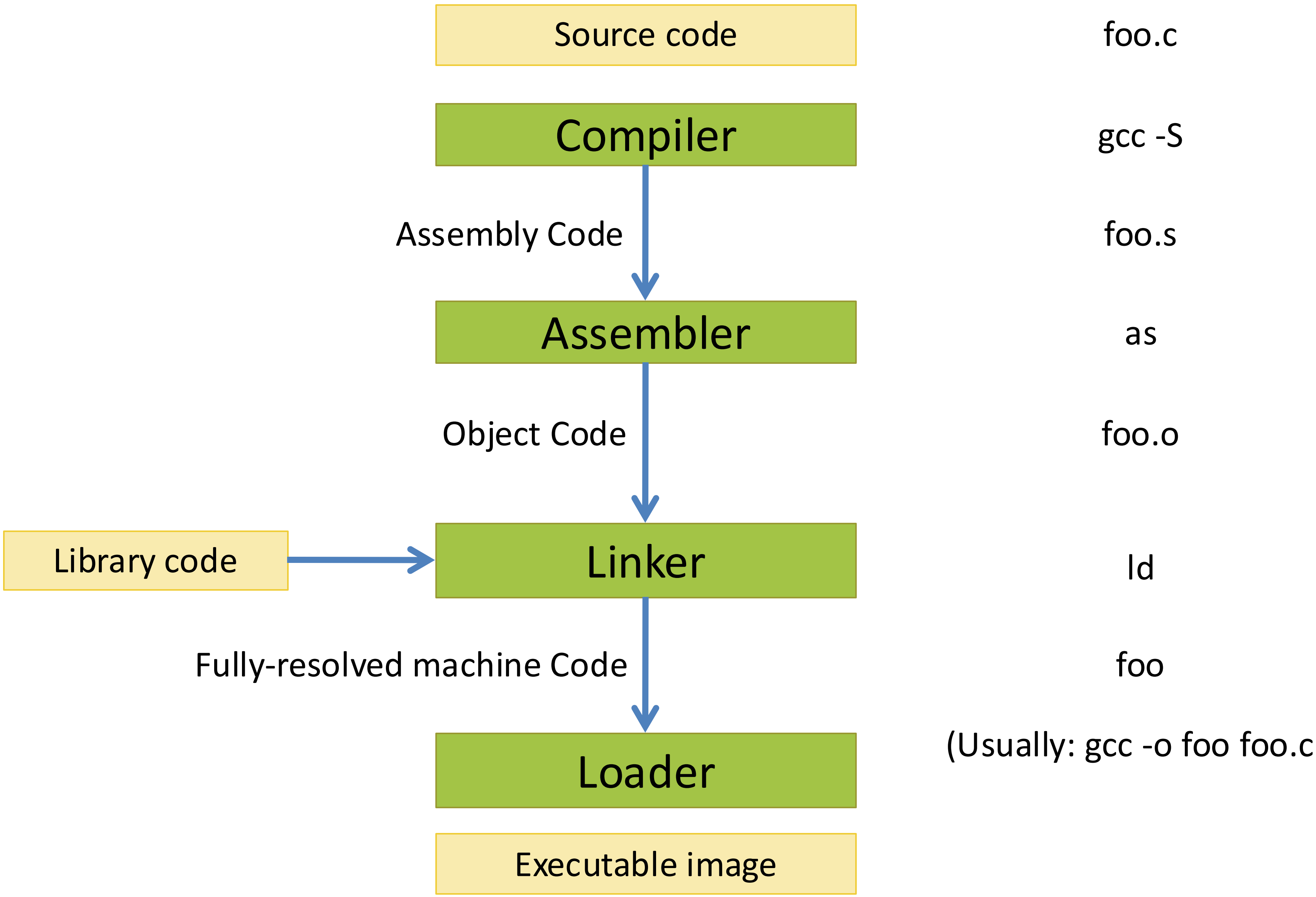


Typical Compiler Stages

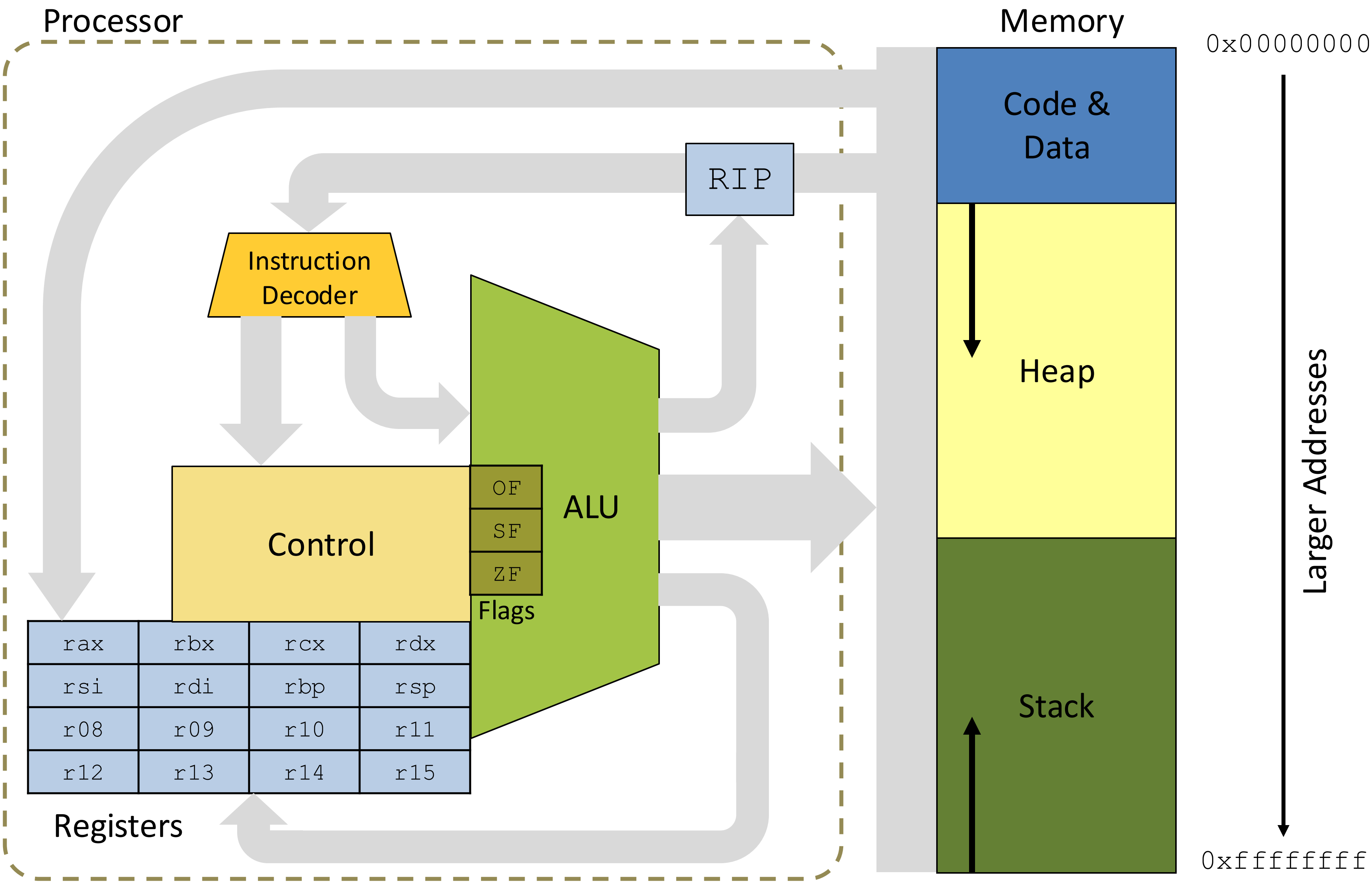
Compiler Passes		Representations of the program
Lexing	→	token stream
Parsing	→	abstract syntax
Disambiguation	→	abstract syntax
Semantic analysis	→	annotated abstract syntax
Translation	→	intermediate code
Control-flow analysis	→	control-flow graph
Data-flow analysis	→	interference graph
Register allocation	→	assembly
Code emission		

- Optimizations may be done at many of these stages
- Different source language features may require more/different stages
- Assembly code is not the end of the story

Compilation & Execution

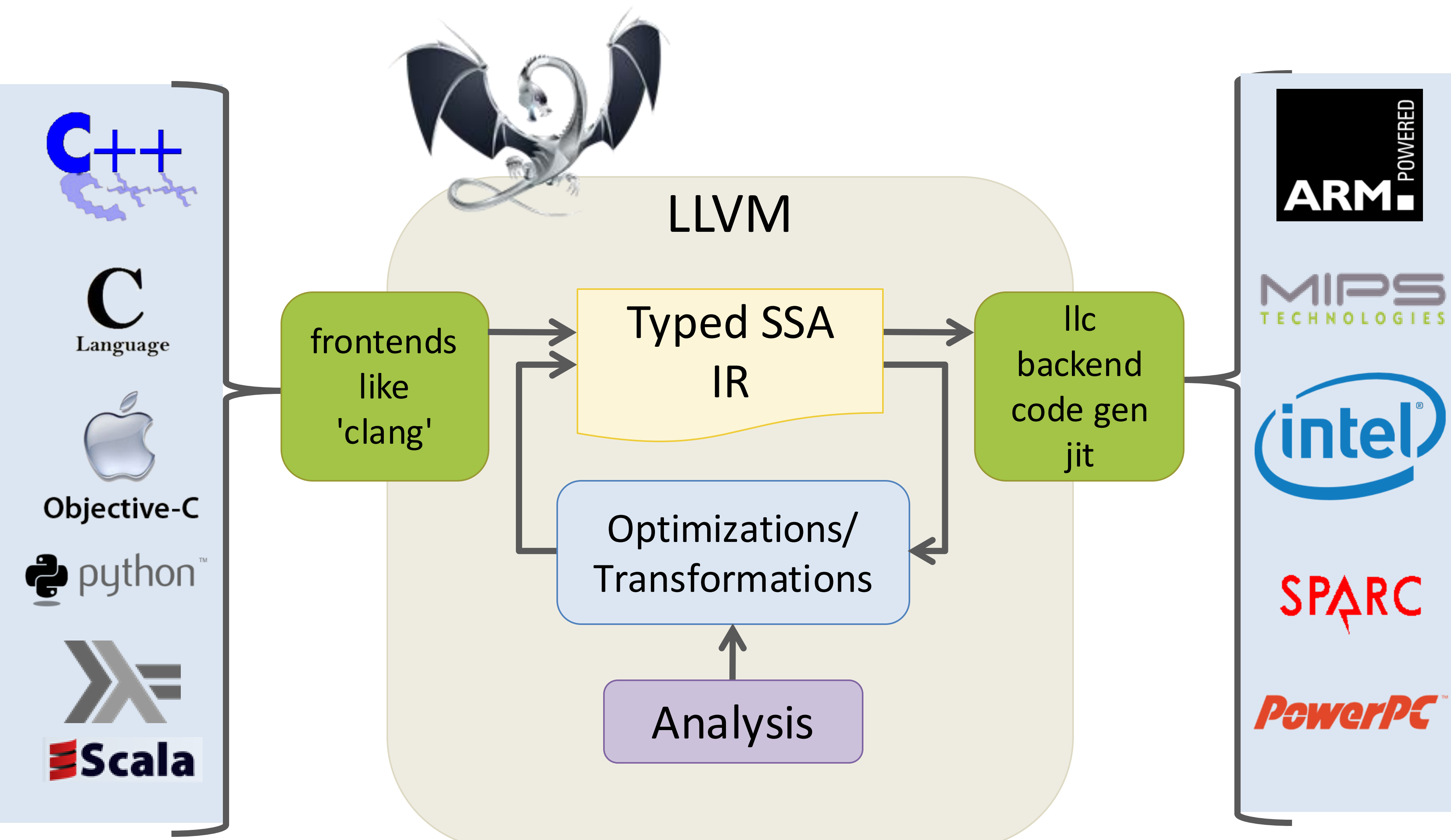


X86 Schematic



LLVM Compiler Infrastructure

[Lattner et al.]



Untyped Lambda Calculus Syntax

Abstract syntax in OCaml:

```
type exp =
 | Var of var        (* variables            *)
 | Fun of var * exp  (* functions: fun x → e  *)
 | App of exp * exp  (* function application  *)
```

Concrete syntax:

exp ::=

| x	*variables*
| fun x → exp	*functions*
| exp_1 exp_2	*function application*
| (exp)	*parentheses*

Lambda Calculus

- Call-by-value operational semantics
- Call-by-name operational semantics

- Environment-based interpreter: thread through an ***environment***, which maps variables to their values.
  - extend the environment when doing a function call
  - lookup variables in the current environment

- To properly handle first-class functions: use closures
  - a ***closure*** is a pair of a
    (1) a datastructure representing the saved environment, and
    (2) the function body definition

Type checking rules

- Recall from the demo “tc.ml” we have five inference rules:

INT

$$\frac{}{E \vdash i : int}$$

VAR

$$\frac{x : T \in E}{E \vdash x : T}$$

ADD

$$\frac{E \vdash e_1 : int \qquad E \vdash e_2 : int}{E \vdash e_1 + e_2 : int}$$

FUN

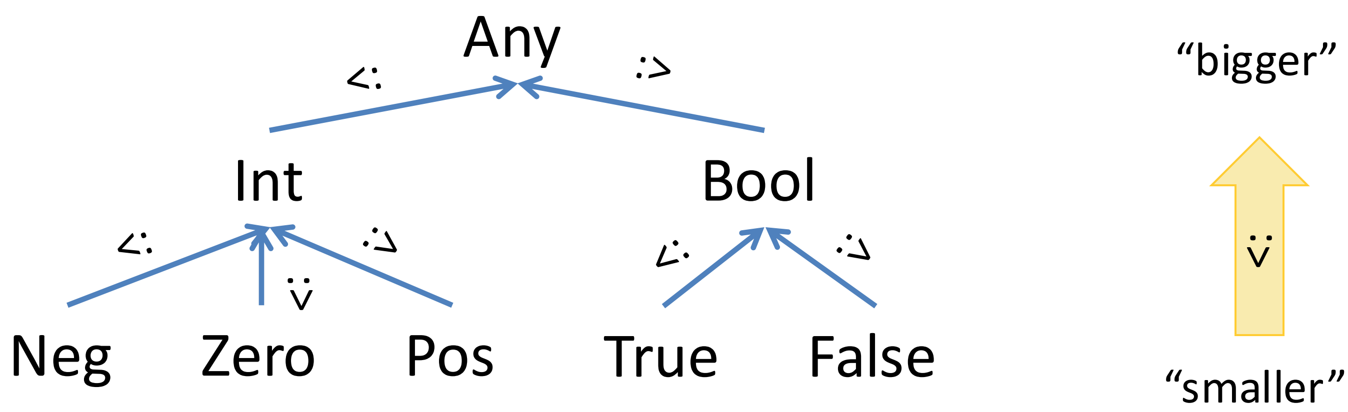
$$\frac{E, x : T \vdash e : S}{E \vdash fun\ (x{:}T) \text{ -> } e\ : T \text{ -> } S}$$

APP

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1\ e_2 : S}$$

Subtyping Hierarchy

- A *subtyping hierarchy*:



- Soundness of subtyping rules
- At compile time we don’t have full information: checked down-cast
- Subtyping other types:
  - Tuples
  - Functions: contravariant and covariant
  - Immutable records: width and depth subtyping

Code Generation for Objects

- Classes:
  - Generate data structure types
    - For objects that are instances of the class and for the class tables
  - Generate the class tables for dynamic dispatch
- Methods:
  - Method body code is similar to functions/closures
  - Method calls require *dispatch*
- Fields:
  - Issues are the same as for records
  - Generating access code
- Constructors:
  - Object initialization
- Dynamic Types:
  - Checked downcasts
  - “instanceof” and similar type dispatch

Optimizations

High level

AST

IR

Mid level

Canonical IR

Low level

Abstract assembly

Assembly

- Inlining
- Function specialization
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength Reduction
- Constant folding & propagation
- Branch prediction / optimization
- Register allocation
- Loop unrolling
- Instruction Selection
- Cache optimization

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- $out[n] := \bigcup_{n' \in succ[n]} in[n']$ (backward)
- $in[n] := gen[n] \cup (out[n] - kill[n])$

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

- $in[n] := \bigcup_{n' \in pred[n]} out[n']$ (forward)
- $out[n] := gen[n] \cup (in[n] - kill[n])$

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

- $in[n] := \bigcap_{n' \in pred[n]} out[n']$ (forward)
- $out[n] := gen[n] \cup (in[n] - kill[n])$

Register Allocation

Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   – Nodes are temporary variables.
   – There is an edge between node n and m if n is live at the same time as m
3. Try to color the graph
   – Each color corresponds to a register
4. In case step 3. fails, “spill” a register to the stack and repeat the whole process.
5. Rewrite the program to use registers

What have we learned?

Where else is it applicable?

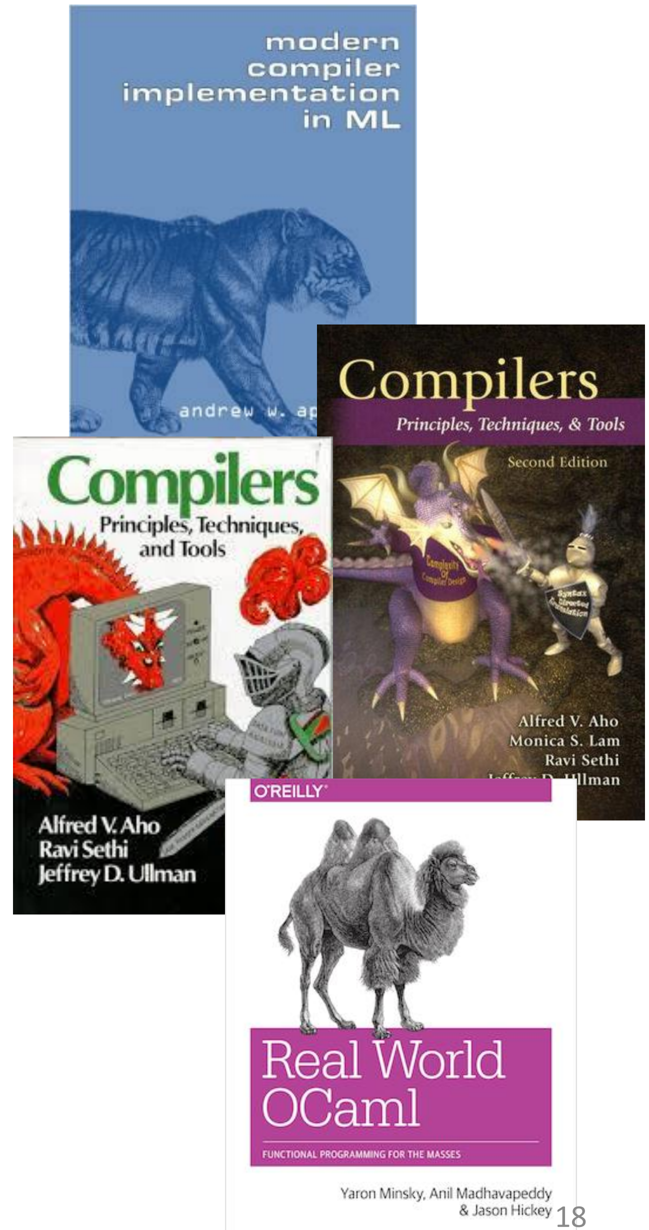
What next?

COURSE WRAP-UP

Why CS 131?

- You will learn:
  - Practical applications of theory
  - Parsing
  - How high-level languages are implemented in machine language
  - (A subset of) Intel x86 architecture
  - A deeper understanding of code
  - A little about programming language semantics
  - Functional programming in OCaml
  - How to manipulate complex data structures
  - How to be a better programmer

- Did we meet these goals?



Stuff we didn't Cover

- We skipped stuff at every level…
- Concrete syntax/parsing:
  - Much more to the theory of parsing… LR(*)
  - Good syntax is art, not science!
- Source language features:
  - Exceptions, advanced type systems, type inference, concurrency
- Intermediate languages:
  - Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
- Compilation:
  - Continuation-passing transformation, efficient representations, scalability
- Optimization:
  - Scientific computing, cache optimization, instruction selection/optimization
- Runtime support:
  - memory management, garbage collection

Lexing
Parsing
Disambiguation
Semantic analysis
Translation
Control-flow analysis
Data-flow analysis
Register allocation
Code emission

Compiler Passes

Related Courses

- CS247: Computer-aided Verification
  - I'm teaching this in Spring 2025.
  - Proving program properties with theorem prover and model checker.
- CS210 / CS211: Computer Architecture II/III
  - Prof. Chundong Wang, Prof. Shu Yin
  - Advanced topics in computer architecture, high-performance computing systems.
- CS224: Software Analysis
  - Prof. Yuqi Chen
  - Data flow analysis, pointer analysis, taint analysis, ...
- CS225: Advanced Distributed Systems
  - Prof. Jingzhu He
  - Concurrency, distributed protocols, ...

Where to go from here?

- Conferences (proceedings available on the web):
  - Programming Language Design and Implementation (PLDI)
  - Principles of Programming Langugaes (POPL)
  - Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  - International Conference on Functional Programming (ICFP)
  - European Symposium on Programming (ESOP)
  - …

- Technologies / Open Source Projects
  - Yacc, lex, bison, flex, …
  - LLVM – low level virtual machine
  - Java virtual machine (JVM), Microsoft's Common Language Runtime (CLR)
  - Languages: OCaml, F#, Haskell, Scala, Go, Rust, …?

Where else is this stuff applicable?

- General programming
  - Better understanding of how the compiler works can help you generate better code.
  - Ability to read assembly output from compiler
  - Experience with functional programming can give you different ways to think about how to solve a problem
- Writing domain specific languages
  - lex/yacc very useful for little utilities
  - understanding abstract syntax and interpretation
- Understanding hardware/software interface
  - Different devices have different instruction sets, programming models

Thanks!

- To the TAs: **You Chunhan, Zheng Jiaye**

- To *you* for taking the class!

- How can I improve the course?
  - Let me know in course evaluations!