Lecture G:

Intermediate Representations

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- Short, Mini, not that big a deal, but it's still a homework HW 3 is due Monday — 1 late day max!
- Midterm is next Friday
  - ✦ topics & old exams available on the course website
  - ✦ Closed Book! No notes except one 3x5 index card, hand-written only. Index cards available today & next week after class.
  - ✦ MiniJava BNF will be provided
  - ✦ Review for midterm during sections next week (May 8)

Outline

Compiler Architecture & IRs

Linear IRs — 3AC & Stack Machines

CFGs & Basic Blocks

Other IRs

Overview & Taxonomies

Outline

Compiler Architecture & IRs

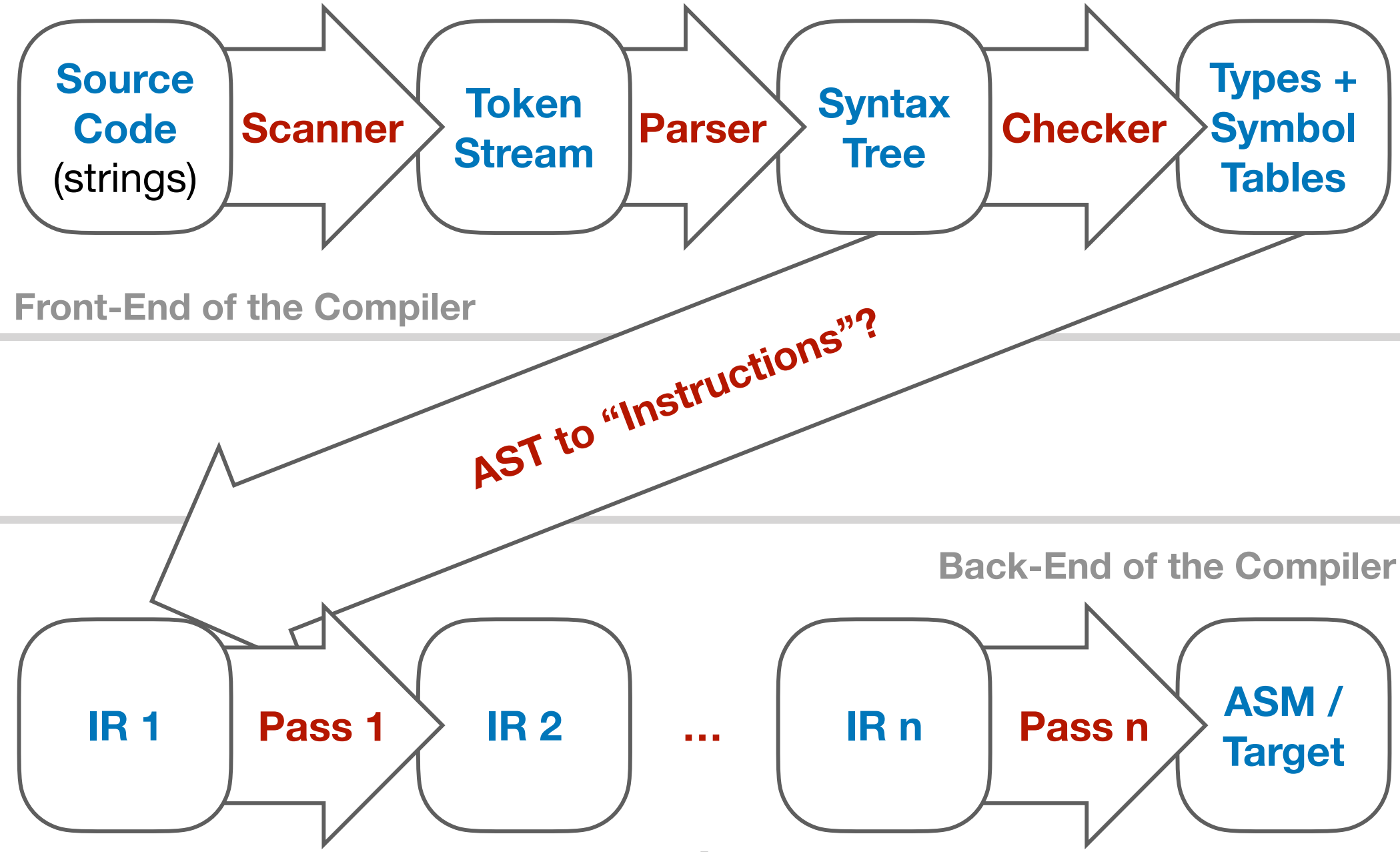
Linear IRs — 3AC & Stack Machines

CFGs & Basic Blocks

Other IRs

Overview & Taxonomies

Our Compiler's Structure



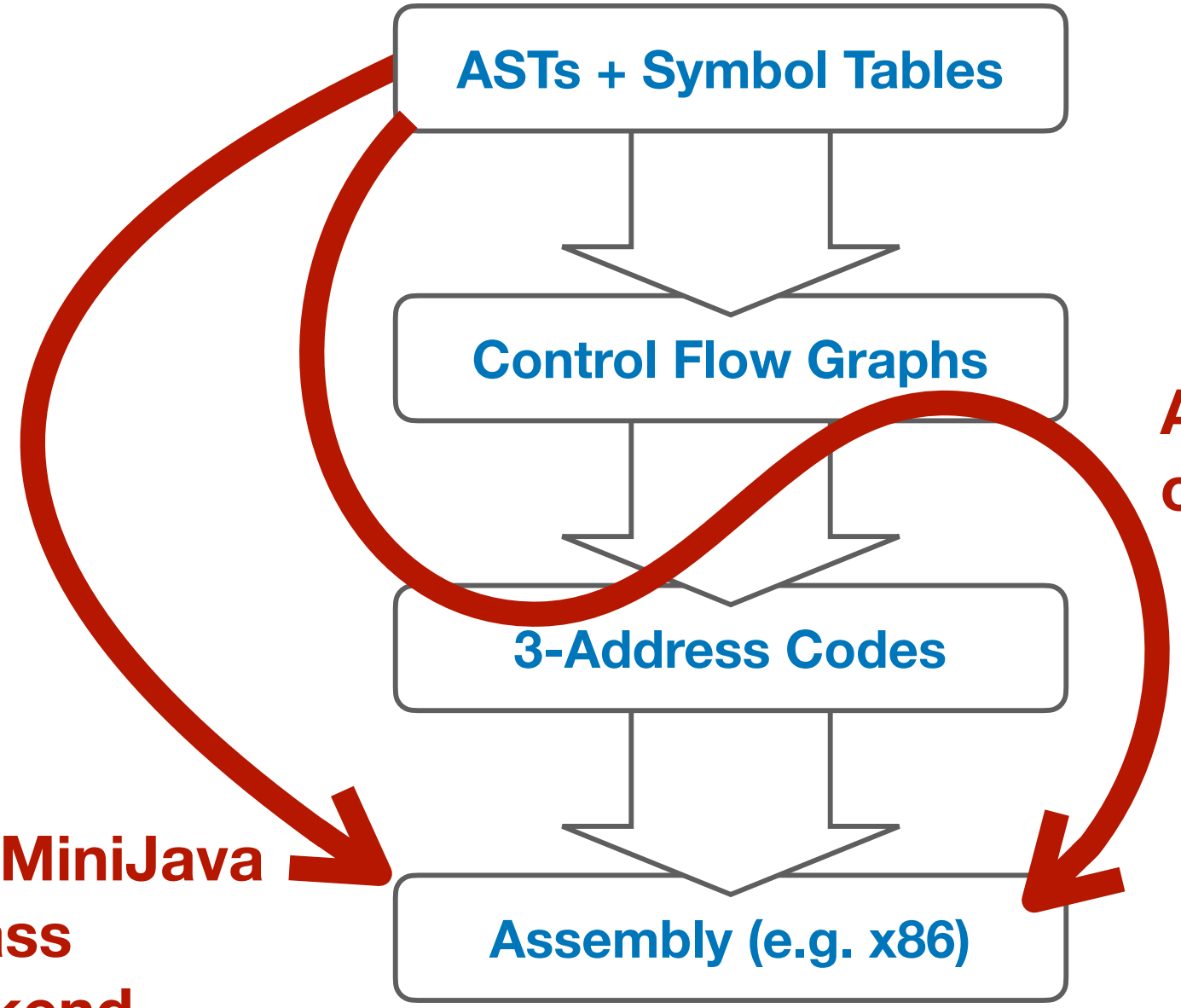
Structure Inside the Backend

- **Lowering Pass to Backend** — transforms ASTs to the middle/main IR for the compiler
- **Middle IRs** — Control Flow Graph + Basic Blocks
  - ✦ Basic Blocks use three-address code or stack machine code; usually in single static assignment form
- **Optimization Passes** — work on middle IRs
- **Analysis IRs** — Dependency Graphs, annotations, etc.
  - ✦ derivative/temporary IRs for specific optimizations
- **Target-specific Passes** — passes that lower from the target-independent middle IRs to a specific assembly code

IRs, IRs, and more IRs

- No one IR is best for all purposes
  - Every compiler defines IRs differently, and defines many different IRs
- Considerations
  - What structure has been **discarded** vs. **introduced**? i.e. is the IR closer to the front or back of the compiler?
  - Does the IR have an efficient/compact encoding? (IR efficiency strongly affects compilation times)
  - Is the IR easy to generate from the preceding pass?
  - Is the IR easy to transform/manipulate?
  - Is the IR easy to analyze?

IR Scheme for this Lecture



Outline

Compiler Architecture & IRs

Linear IRs — 3AC & Stack Machines

CFGs & Basic Blocks

Other IRs

Overview & Taxonomies

Linear IRs

- Code for an ***abstract machine***, in the form of a sequence of instructions (sequence = linear)
- Each instruction does exactly one thing
- Easy to build data structures — array or list of instructions, each instruction of fixed size
- 2 examples — **3-address code** & **stack machine code**

3-address code (3AC)

```
t1 ← 2
t2 ← b
t3 ← t1 * t2
t4 ← a
t5 ← t4 - t3
```

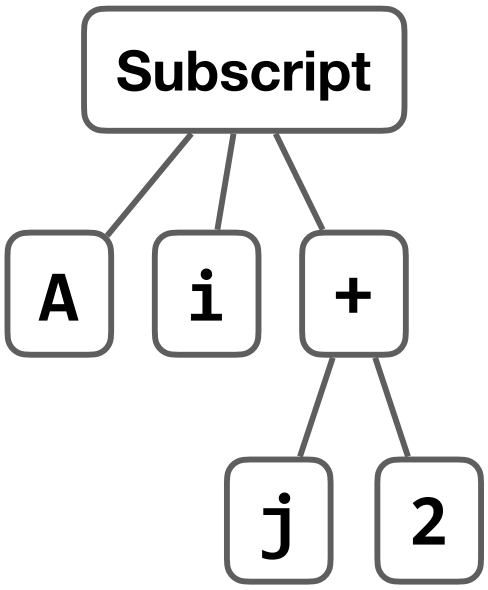
stack machine code

```
push 2
push b
multiply
push a
subtract
```

Example — Array Dereference

Suppose we read a 2d array* — `A[i, j+2]`

AST



High-level 3AC

```
t1 ← j + 2
t2 ← A[i,t1]
```

Mid-level 3AC

```
t1 ← j + 2
t2 ← i * 20
t3 ← t1 + t2
t4 ← 4 * t3
t5 ← addr a
t6 ← t5 + t4
t7 ← load t6
```

Low-level 3AC

```
r1 ← [fp-4]
r2 ← r1 + 2
r3 ← [fp-8]
r4 ← r3 * 20
r5 ← r4 + r2
r6 ← 4 * r5
r7 ← fp-216
f1 ← [r7+r6]
```

**A is a 10x20 array of 32-bit ints*

Level of Abstraction

- What does high-level vs. mid-level vs. low-level mean?
- Oh no, ambiguity!
  - ✦ There's no real definition of "high" vs. "low" level
- **High-level** — should be closer to the AST / source code
  - ✦ Concise, but can't optimize hidden details (e.g. indexing arithmetic)
- **Low-level** — should be closer to the assembly code
  - ✦ Verbose, lots to optimize, but many optimization decisions have already been made
- **Mid-level** — strikes a balance that's good for many target-independent optimizations

Three Address Code (3AC or TAC)

- general form of instruction — **`x ← y op z`**
  - ✦ one operator (op) and 3 names (x, y, z)
  - ✦ degenerate forms **`x ← y`**, **`x ← op y`**
- example — **`x = 2 * (m+n)`** gets converted into

  `t1 ← m + n;  t2 ← 2 * t1;  x ← t2;`

  - ✦ you may prefer assembly-ish syntax — **`add t1, m, n;`**
- 3AC (before register allocation) allows for an arbitrary number of temporary names
- Storage — could have n-address codes, but 3AC has a very *regular* layout in memory (1 op + 3 addresses)

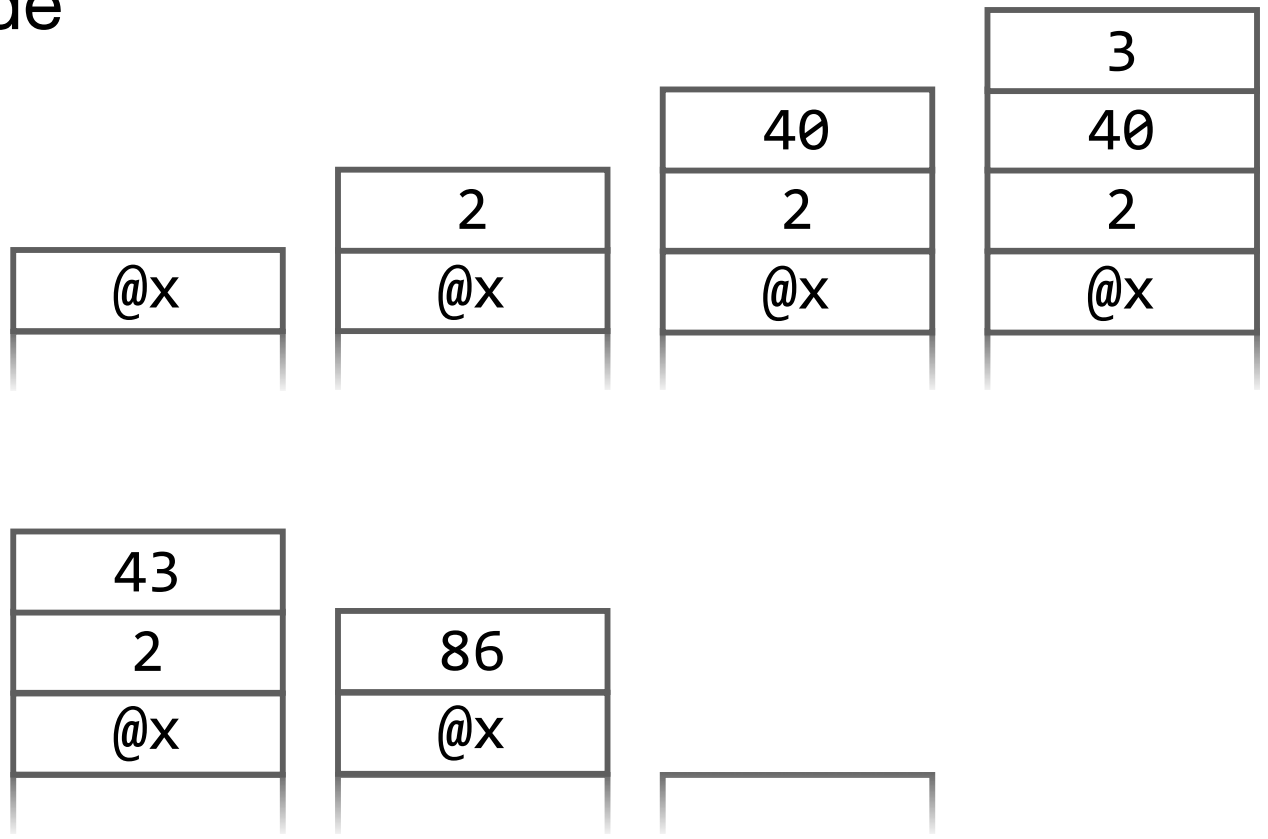
Three Address Code (3AC or TAC)

- Abstract Machine Model
  - "an infinite register file" — arbitrarily many **names**
  - A main memory (loads & stores) or other sim. memory
- Advantages
  - similar to most assembly languages / processor ISAs
  - allows for naming intermediate values explicitly
- Examples
  - ILOC (Cooper & Torczon), LLVM IR (mostly)

Stack Machine Code

- example — **`x = 2 * (40 + 3)`**
- stack machine code

```
pushaddr x
pushconst 2
pushconst 40
pushconst 3
add
mult
store
```



Abstract Machine Model

Stack Machines — History & Practice

- Originally used for stack-based computers (e.g. B5000, circa 1961)
- Famously used for many virtual machines
  - ✦ Pascal — pcode (1973)
  - ✦ Forth (1970)
  - ✦ Post-Script (1984)
  - ✦ JVM (1996) and MSIL/CIL (2002, C#/VB bytecode)
- Stack machine code is viewed as highly portable partly because it does not require decisions about register allocation

Stack Machine Code

- Advantages
  - Gets rid of names for intermediaries
  - Very easy to generate correct machine code for
  - More compact (per-instruction; not necessarily overall)
- Disadvantages
  - Gets rid of names for intermediaries
  - Difficult to optimize directly
  - Does not match actual machine ISAs
- **Our approach to MiniJava codegen will be secretly based on a stack machine!**

Single-Static Assignment (SSA)

- We will discuss more during lectures on dataflow analysis
- SSA is a variant of 3AC with certain additional constraints
  - ✦ Every name should be assigned a value on **a single** line, then not changed or mutated on any line after that. i.e. the assignment to the variable should be **static**
- It is easier to reason about code when variables are not reassigned / mutated (e.g. how variables work in math)

3AC w/mutation

```
x ← 2
y ← b
x ← x * y
y ← a
x ← y - x
```

```
t1 ← 2
t2 ← b
t3 ← t1 * t2
t4 ← a
t5 ← t4 - t3
```

Single Static Assignment

Jumps & Calls?

- Linear IR examples (3AC, Stack Machine Code) that we've seen don't include ***control flow*** such as **if**, or **jump**, or **call function**
- We can include these, but then we probably also need to include labels to jump to
- Should functions be reduced to labels and jumps? (much lower-level) or should function arguments / return values be kept? (higher-level)

Outline

Compiler Architecture & IRs

Linear IRs — 3AC & Stack Machines

CFGs & Basic Blocks

Other IRs

Overview & Taxonomies

Basic Blocks

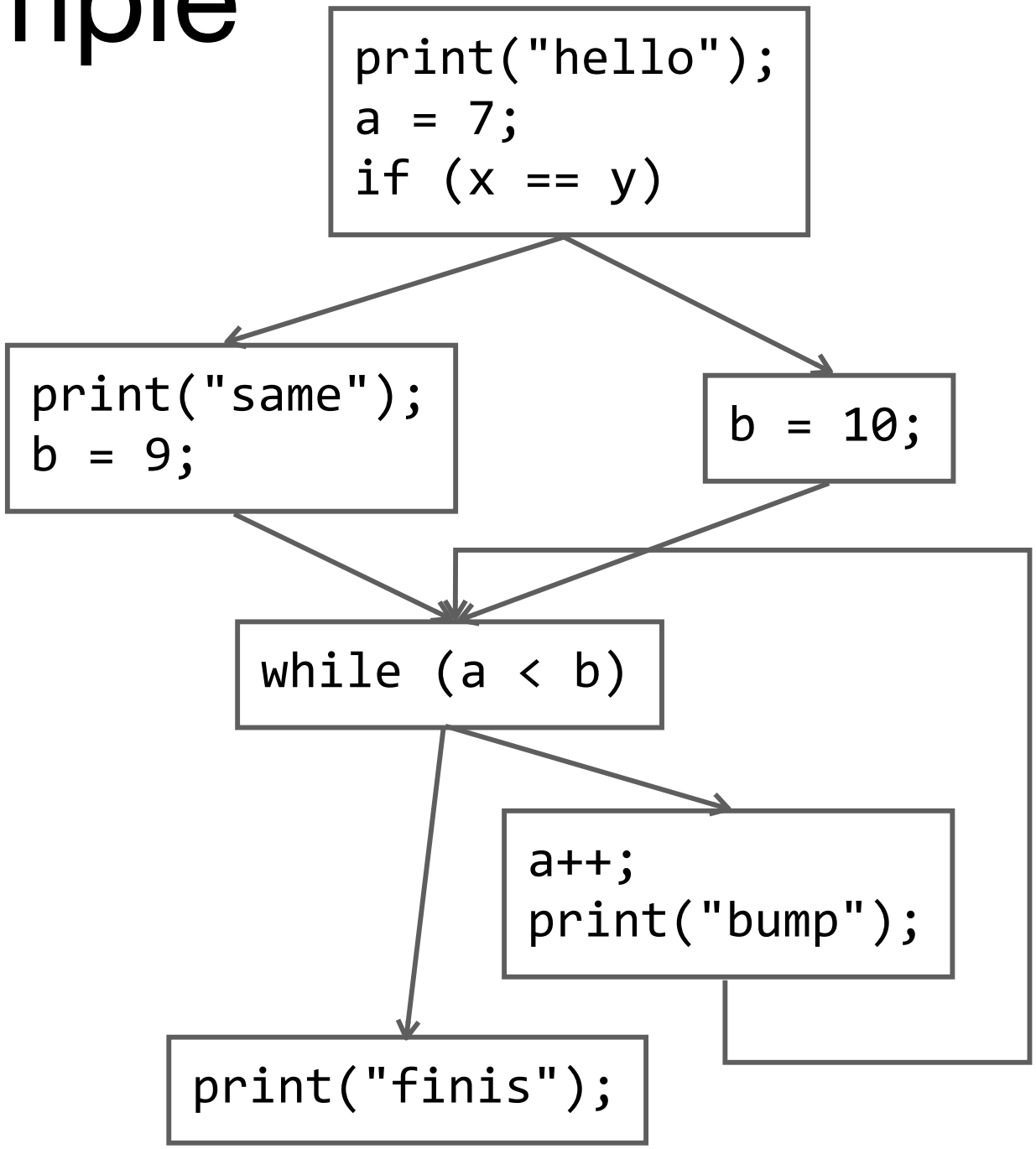
- **Basic Block** — A *maximal sequence* of **straight-line** (i.e. branch free) code *[Cooper & Torczon Ch. 4]*
  - ✦ "thread of control" (program counter, etc.) can only enter at the top and leave at the bottom — therefore can be viewed as *atomic* w.r.t. control flow
- Question — Can a basic block call a function in the middle of the block?
- LLVM IR says "sure" but Cooper & Torczon def. says "no"
- Contents of basic blocks are encoded in a linear IR (e.g. 3AC)

Control Flow Graph (CFG)

- CFG no longer means “Context-Free Grammar”
  - ✦ Don’t you love TLAs?
- A graph, potentially cyclic
  - ✦ nodes — basic blocks
  - ✦ edges — possible path of control flow between basic blocks (every execution of the program must be a path/ trajectory through the CFG)

"CFG" Example

```
print("hello");
a = 7;
if (x == y) {
  print("same");
  b = 9;
} else {
  b = 10;
}
while(a < b) {
  a++;
  print("bump");
}
print("finis");
```



Why Should We Use CFGs?

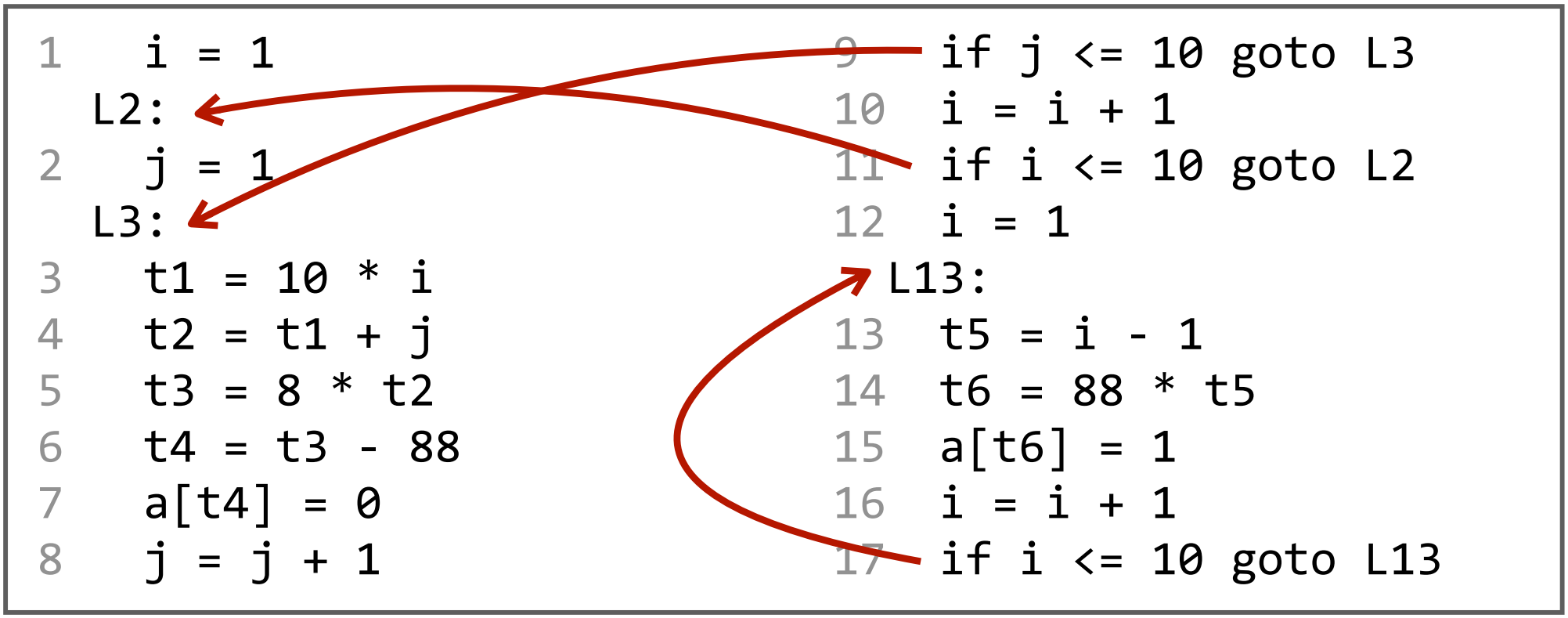
- Why not just have a big long linear IR version of a program with branches? Why should we build a CFG?
- the linear order in which basic blocks get sequenced is (mostly) arbitrary (some optimization differences at lowest-level)
- can think separately about optimizing basic blocks vs. bigger picture
- helps with analyzing the code, e.g.
  - ✦ “Does the program always have to execute block 3 before block 7?”
  - ✦ “Is it possible to reach all basic blocks?”

Constructing CFG from 3AC (1)

```
1    i = 1
   L2:
2    j = 1
   L3:
3    t1 = 10 * i
4    t2 = t1 + j
5    t3 = 8 * t2
6    t4 = t3 - 88
7    a[t4] = 0
8    j = j + 1
9    if j <= 10 goto L3
10   i = i + 1
11   if i <= 10 goto L2
12   i = 1
   L13:
13   t5 = i - 1
14   t6 = 88 * t5
15   a[t6] = 1
16   i = i + 1
17   if i <= 10 goto L13
```

- Suppose we directly generate 3AC from the AST
- What are the **basic blocks**?
  - recall — entrance and exit at only one place!
  - No jumps in or out of the middle
  - "I've started, so I'll finish"

Constructing CFG from 3AC (2)



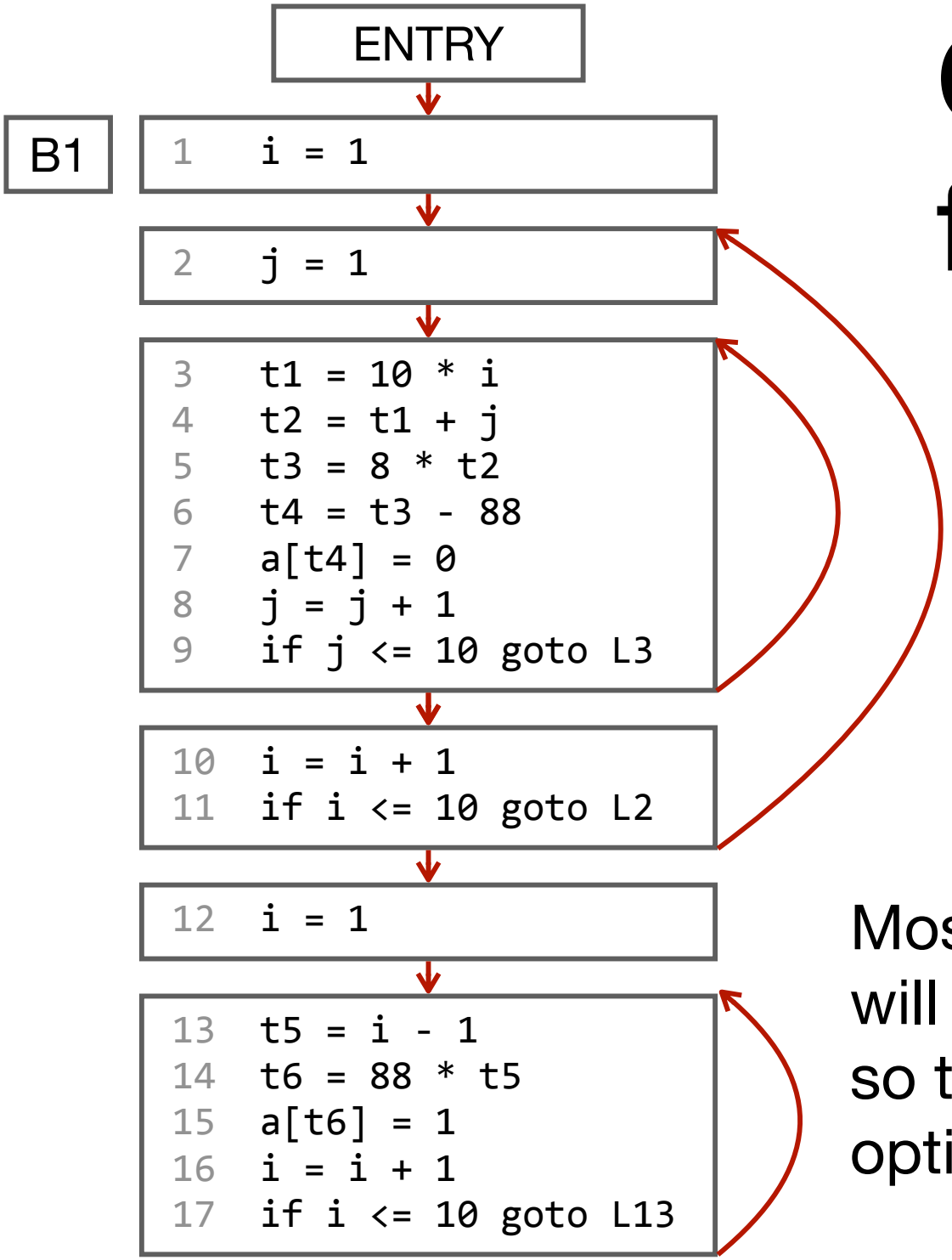
- Can we just break the code at each *label*?
- What are all possible jumps?
- hmmm…

Algorithm: Find Leaders

```
1    i = 1
   L2:
2    j = 1
   L3:
3    t1 = 10 * i
4    t2 = t1 + j
5    t3 = 8 * t2
6    t4 = t3 - 88
7    a[t4] = 0
8    j = j + 1
9    if j <= 10 goto L3
10   i = i + 1
11   if i <= 10 goto L2
12   i = 1
  L13:
13   t5 = i - 1
14   t6 = 88 * t5
15   a[t6] = 1
16   i = i + 1
17   if i <= 10 goto L13
```

- A **leader** is the first instruction in a basic block
  - The very first instruction
  - Any target of a branch/jump/goto
  - Any instruction immediately after a branch/jump/goto

CFG Result from Example



Easier to see now that there are 3 loops, with 2 nested loops

Most of the execution time will be spent in loop bodies, so this suggests where optimization is most valuable

Basic Block Finding Algorithm

1. Perform linear scan of instruction list to identify leaders
   - leader at beginning of method
   - leader at target of branch
   - leader immediately following a branch or return
2. scan a second time, starting a new basic block with each line that is a leader

Outline

Compiler Architecture & IRs

Linear IRs — 3AC & Stack Machines

CFGs & Basic Blocks

Other IRs

Overview & Taxonomies

IRs for Analysis

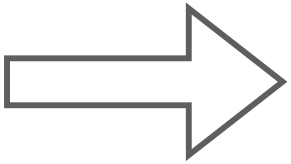
- IRs we've seen up to now represent the code
  - ✦ They are ***definitive*** (Cooper & Torczon) — that is, they define the program being compiled
- During optimization we often want to **analyze** the code, not just represent and manipulate it
  - ✦ IRs that augment definitive IRs are ***derivative*** (Cooper & Torczon) — that is, they are derived from some definitive form of the program being compiled
  - ✦ derivative IRs are constructed for some purpose and then discarded once the definitive IR is changed

Dependency Graphs

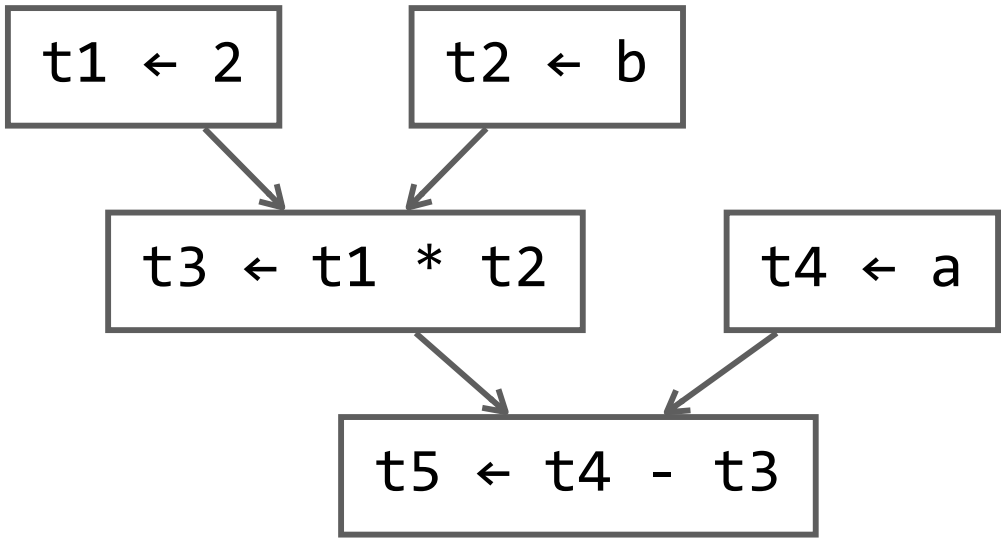
- The most basic dependency question
  - ✦ Given a sequence of instructions `s1; s2; s3; s4`
  - ✦ Is the sequence `s1; s3; s2; s4` equivalent?
    i.e. is it legal to reorder two instructions?

e.g. 3AC

```
t1 ← 2
t2 ← b
t3 ← t1 * t2
t4 ← a
t5 ← t4 - t3
```



Dependency Graph (not CFG)

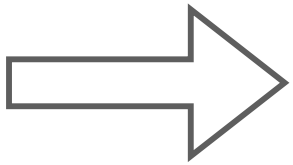


Dependency Graphs

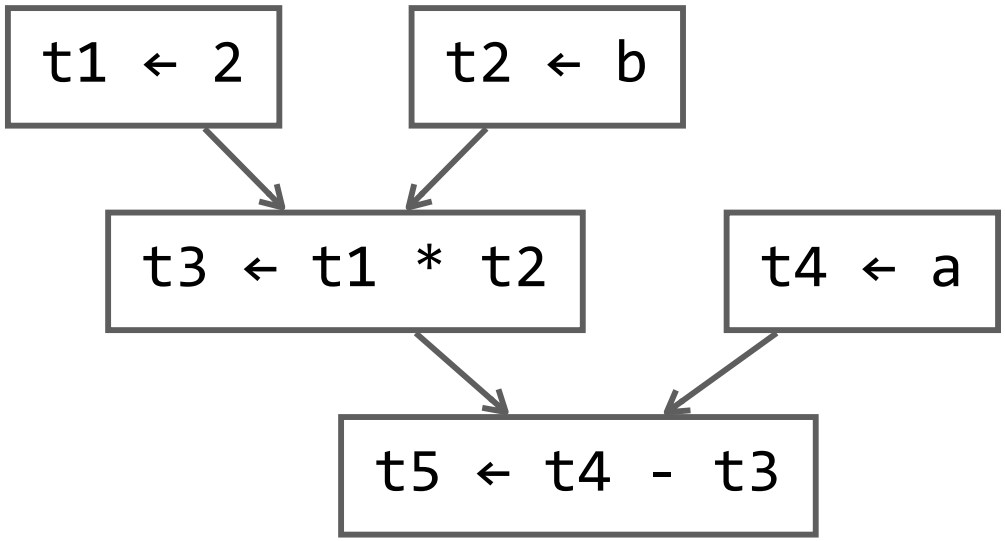
- These dependency graphs are used in an optimization known as **scheduling** (covered near the end of this course)

e.g. 3AC

```
t1 ← 2
t2 ← b
t3 ← t1 * t2
t4 ← a
t5 ← t4 - t3
```



Dependency Graph (not CFG)



Dataflow Graphs

- Similar to dependency graphs, but only concerned with the flow of data
- One version of **dataflow graph** is used for dataflow analysis (will cover later in this course)
- Another version of **dataflow graph** is used for compilers in machine learning, like PyTorch, TensorFlow, XLA, etc.
  - ✦ In machine learning systems, the dataflow graph is ***definitive***, rather than ***derivative***

Outline

Compiler Architecture & IRs

Linear IRs — 3AC & Stack Machines

CFGs & Basic Blocks

Other IRs

Overview & Taxonomies

Taxonomy for IRs

- Tree vs. DAG vs. Graphical (cyclic) vs. Linear
  - ✦ Data-structure (often) dictates code structure of passes
  - ✦ e.g. tree = recursion; linear = loop; cyclic = fixed-point
- Are there names in the IR? How can names be used?
- Memory Models
  - ✦ what kinds of memory does the IR have instructions/ nodes to manipulate?
- Definitive ("source of truth") vs. Derivative ("supplement")
- Level of Abstraction (low, medium, high)

Which is the Right IR to Use?

- There is no single, correct IR to use
- In the front-end, probably AST or variant thereof
- Much of the back-end (some variant of CFG + 3AC)
- Depending on what we want to do, we may convert or augment the IR to make some change and then change back
- Level of Abstraction
  - ✦ Which **details** does a choice of IR **allow** you to access?
  - ✦ Which **details** does a choice of IR **impose** on you?

Next Week & On!

- x86-64 review next week, then Midterm on Friday
- Week after — Code shape (how to lower from AST to a linear IR)
  - ✦ we'll go straight to x86-64 for the project!
- Last 3 weeks
  - ✦ Dataflow Analysis
  - ✦ Optimization in general
  - ✦ 3 Key Optimizations — Instruction Selection, Instruction Scheduling, and Register Allocation