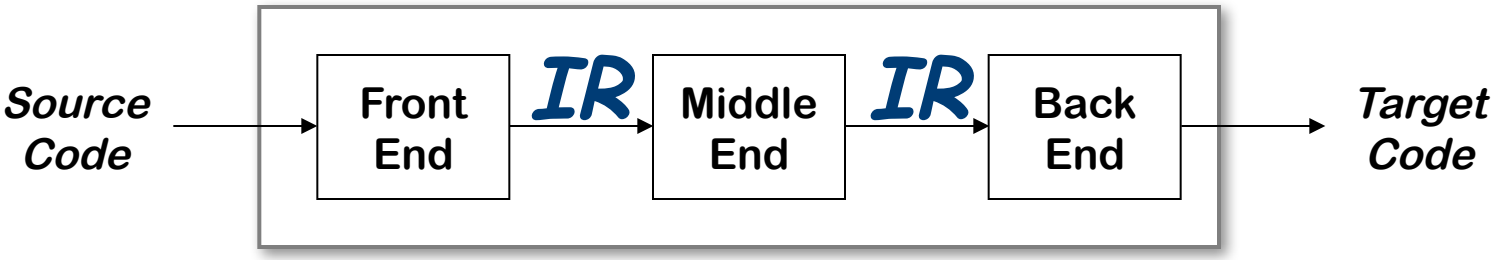
Intermediate Representations

Where In The Course Are We?

- Rest of the course: compiler writer needs to choose among alternatives
  - → Choices affect
    - the quality of compiled code
    - time to compile
  - → There may be no "best answer"

Intermediate Representations

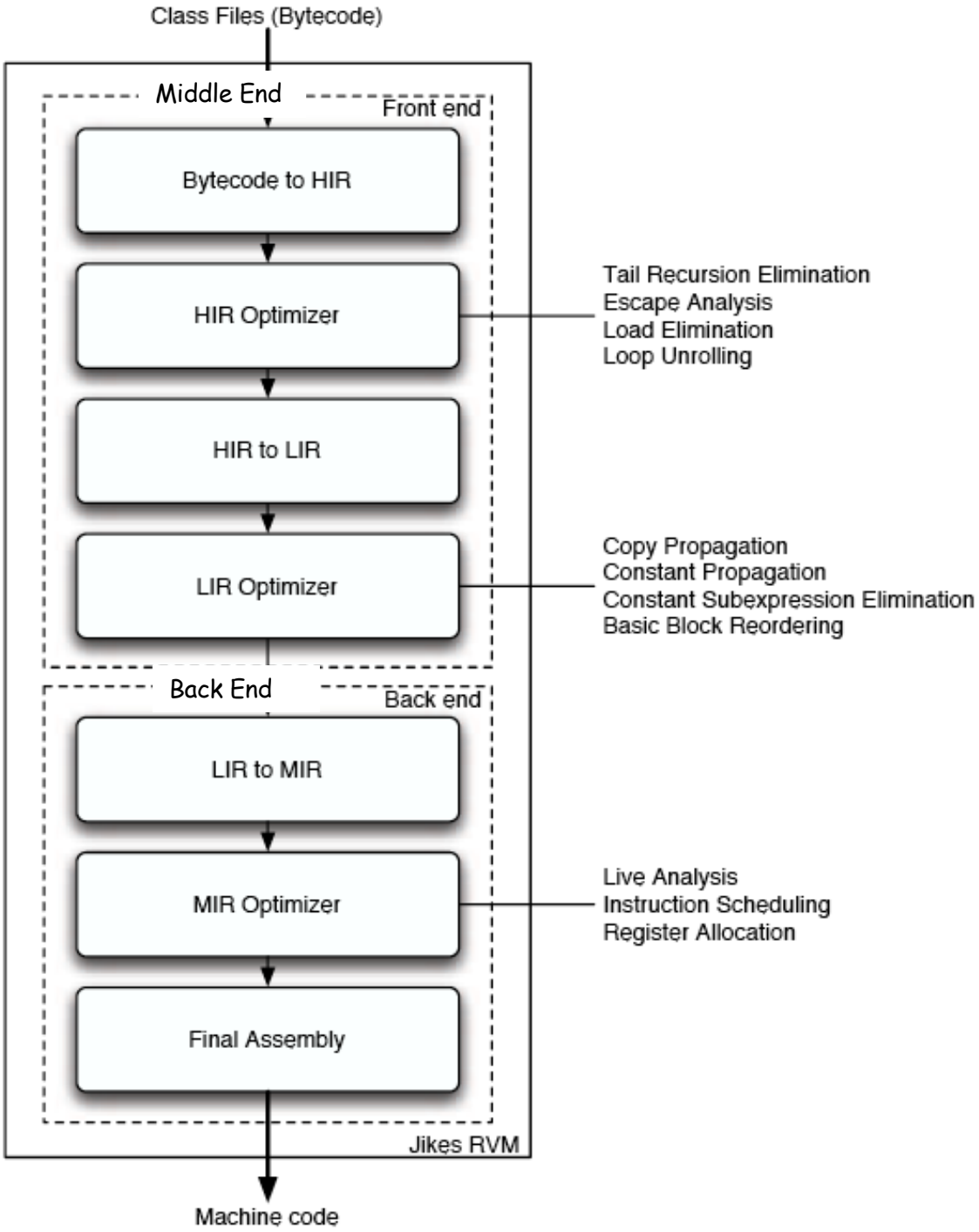


- Front end - produces an intermediate representation (*IR*)
- Middle end - transforms the *IR* into an equivalent *IR* that runs more efficiently
- Back end - transforms the *IR* into native code

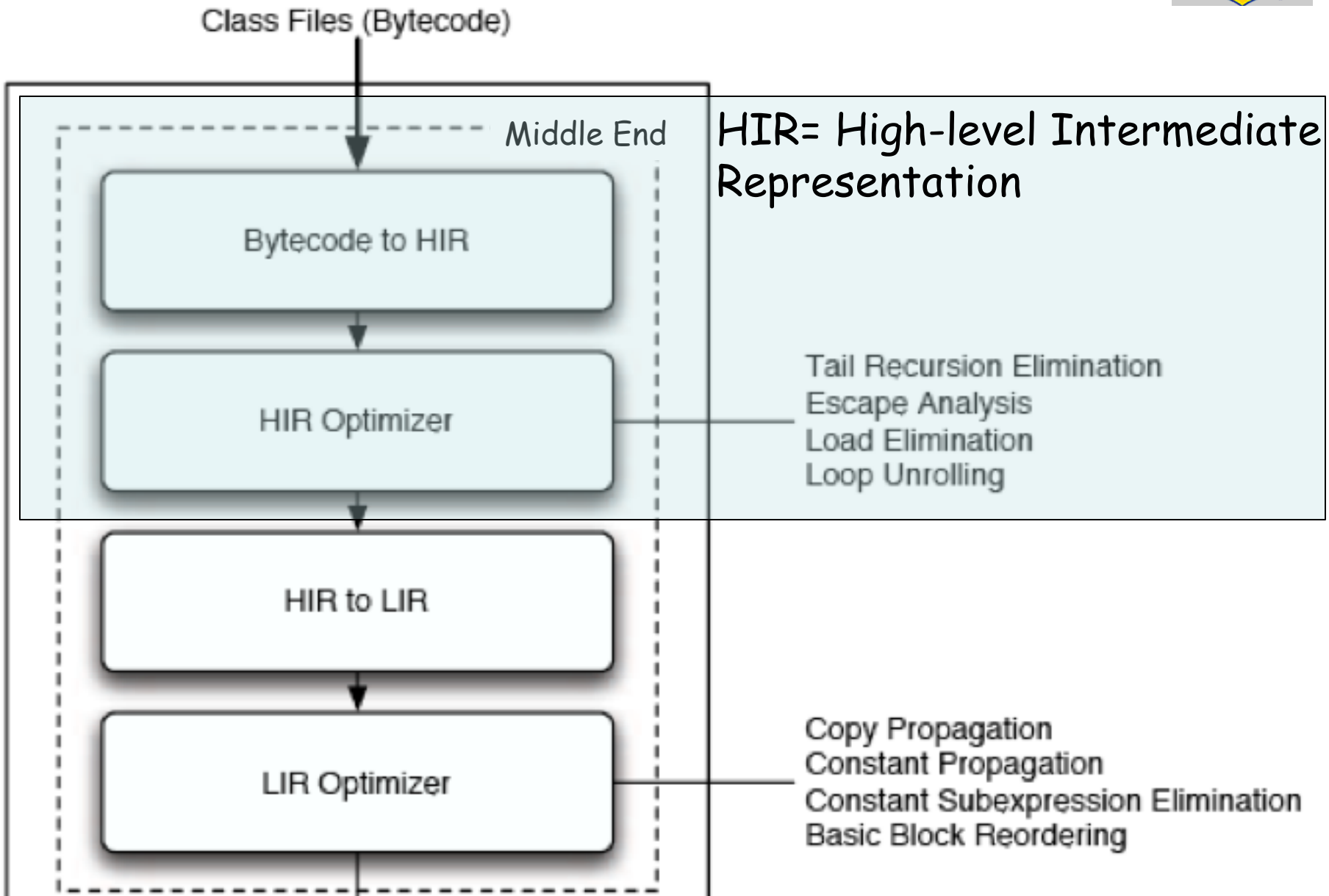
- *IR* encodes the compiler's knowledge of the program
- Middle end usually consists of many passes

JikesRVM

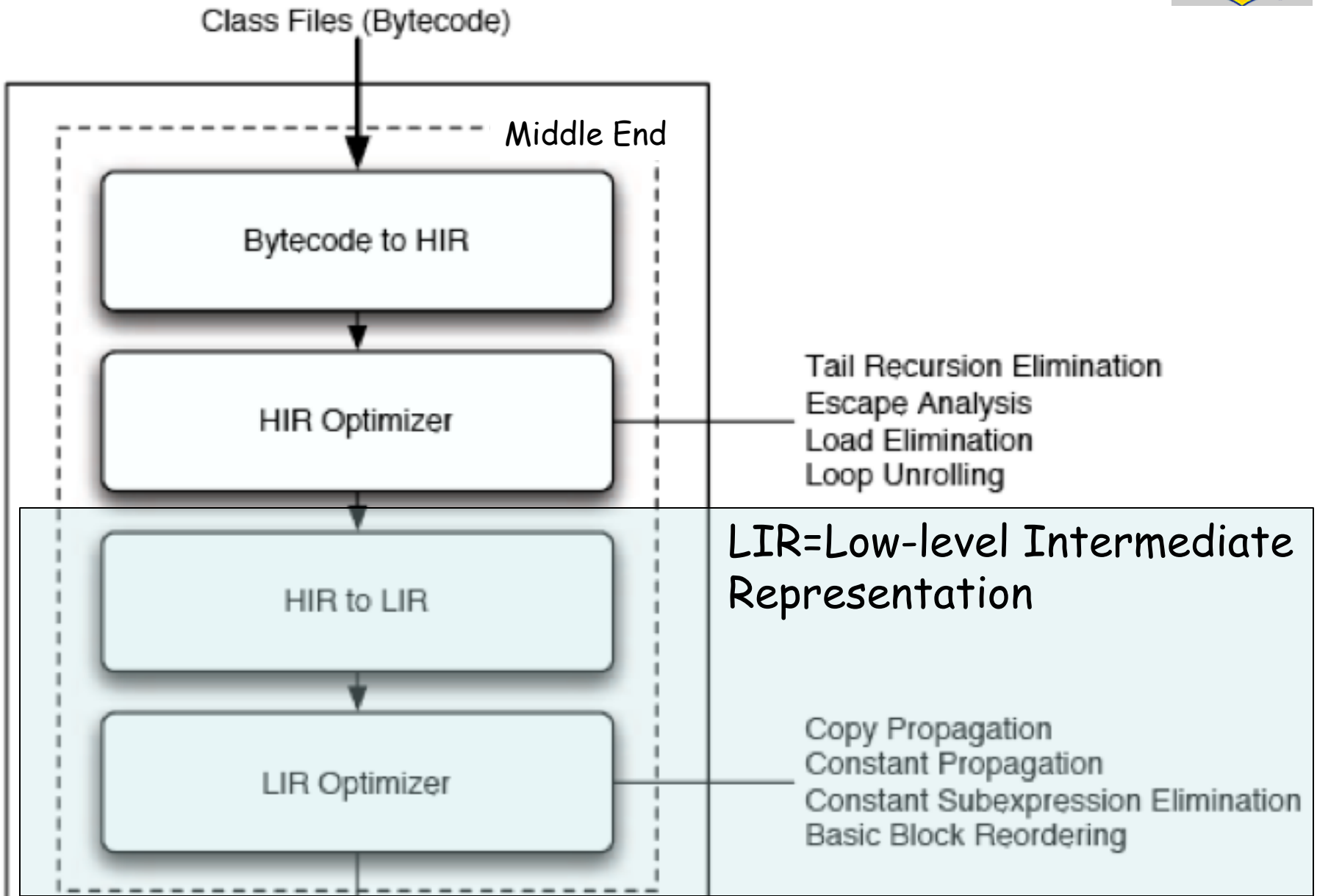
(IBM Open Source Java JIT compiler)



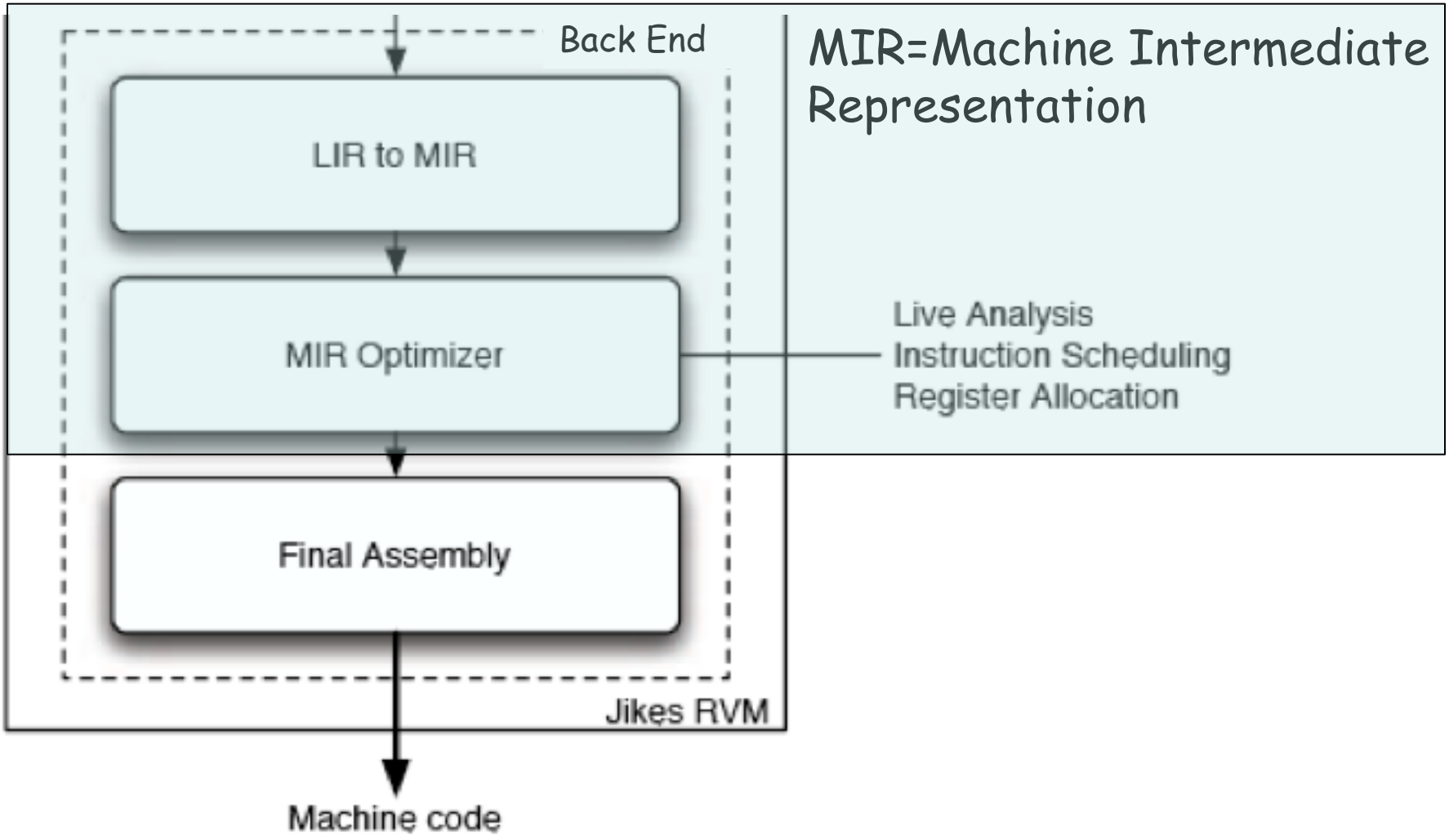
JikesRVM (IBM Open Source Java JIT compiler)



JikesRVM (IBM Open Source Java JIT compiler)



JikesRVM (IBM Open Source Java JIT compiler)



Intermediate Representations

- Decisions in *IR* design affect the speed and efficiency of the compiler

- The importance of different properties varies between compilers
  - → Selecting the "right" *IR* for a compiler is critical

Some Important *IR* Properties

- Ease of generation
  - → speed of compilation
- Ease of manipulation
  - → improved passes
- Procedure size
  - → compilation footprint
- Level of abstraction
  - → improved passes

Types of Intermediate Representations

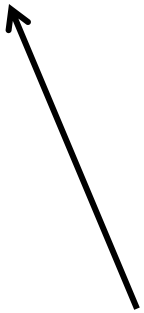
Three major categories

- Structural
- Linear
- Hybrid

Types of Intermediate Representations

Three major categories

- Structural
  - → Graphically oriented
  - → Heavily used in source-to-source translators
  - → Tend to be large
- Linear
- Hybrid



Examples: Trees, DAGs

Types of Intermediate Representations

Three major categories

- Structural
- Linear

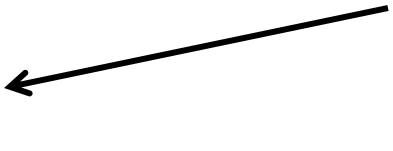
Examples: 3 address code, Stack machine code

  - → Pseudo-code for an abstract machine
  - → Level of abstraction varies
  - → Simple, compact data structures
  - → Easier to rearrange
- Hybrid

Types of Intermediate Representations

Three major categories

- Structural
- Linear
- Hybrid



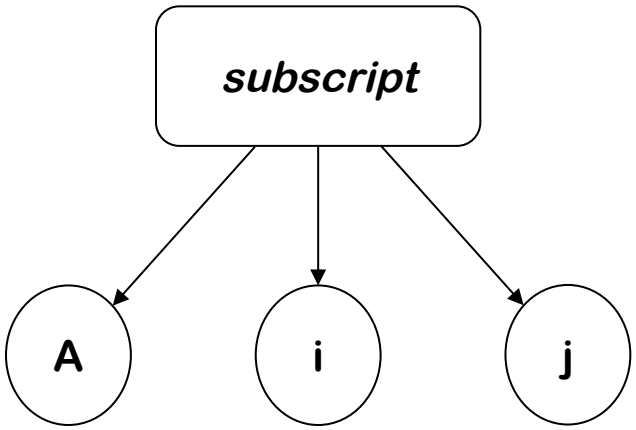
Examples:
Control Flow Graph

→ Combination of graphs and linear code

Level of Abstraction

- Two different representations of an array ref:

High level AST



Good for memory disambiguation

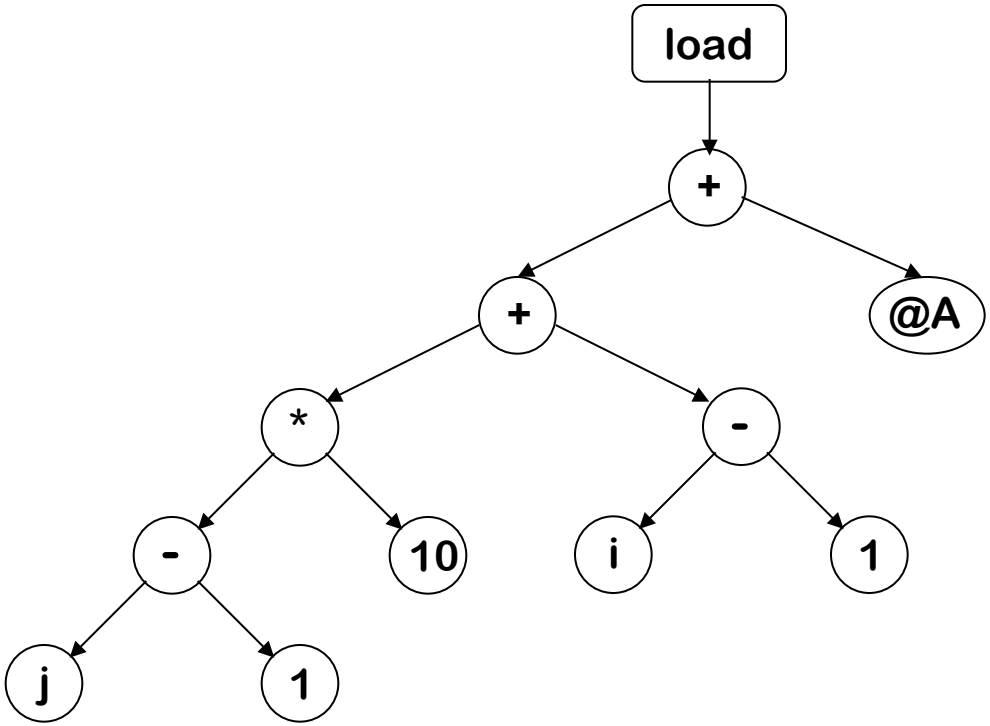
Low level Linear Code

```
loadI  1          => r1
sub    rj, r1     => r2
loadI  10         => r3
mult   r2, r3     => r4
sub    ri, r1     => r5
add    r4, r5     => r6
loadI  @A         => r7
add    r7, r6     => r8
load   r8         => rAij
```

Good for address calculation

Level of Abstraction

- Structural *IRs* are usually considered high-level
- Linear *IRs* are usually considered low-level
- Not necessarily true:




```
loadArray A,i,j
```

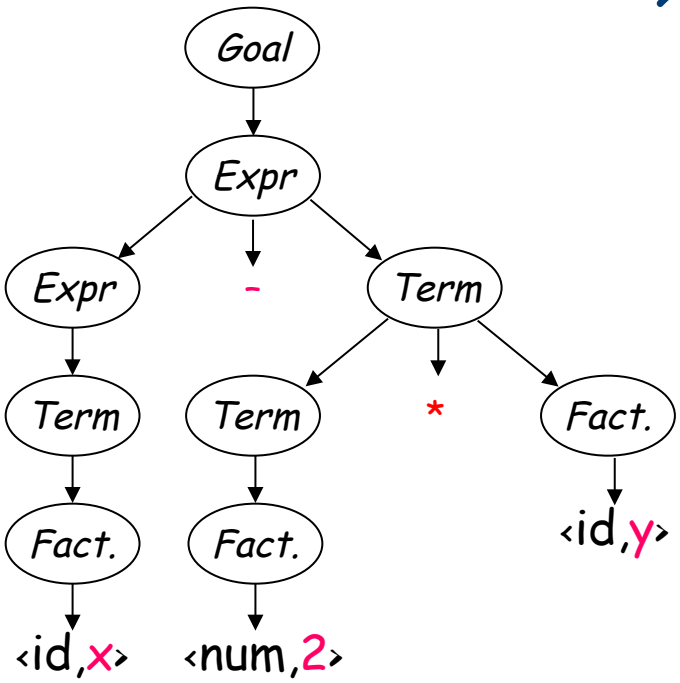
Low level AST

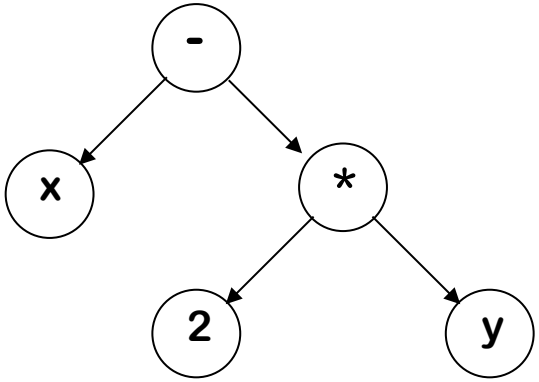
High level linear code

Abstract Syntax Tree

An abstract syntax tree is parse tree with the nodes for most ***non-terminal nodes*** removed

x - 2 * y



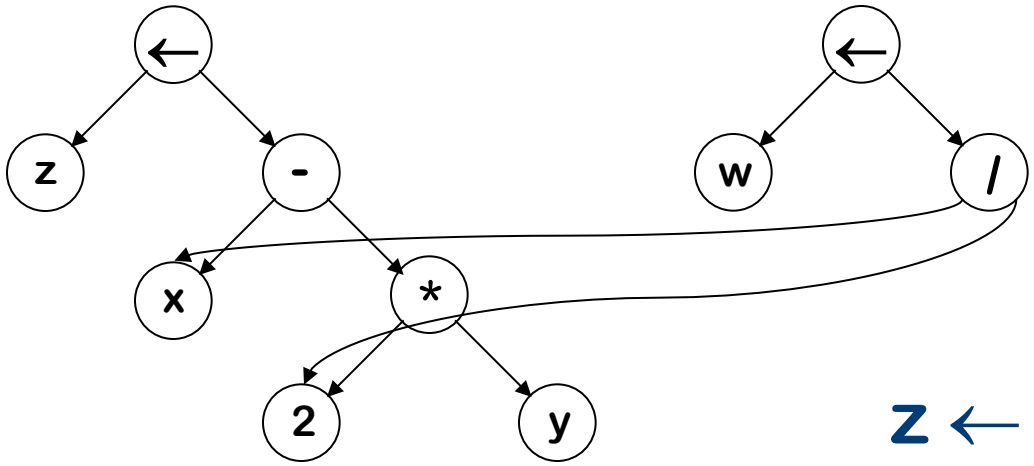


Parse Tree

Abstract Syntax Tree

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



$z \leftarrow x - 2 * y$

$w \leftarrow x / 2$

- Makes sharing explicit
- Encodes redundancy

With two copies of the same expression, the compiler might be able to arrange the code to evaluate it only once.

Stack Machine Code

Originally used for stack-based computers, now Java

- Example:

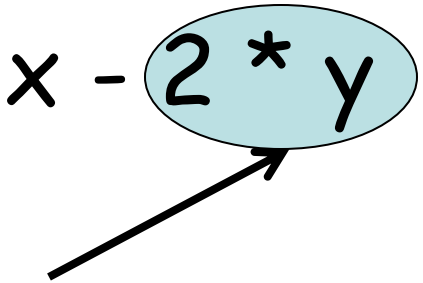
x - 2 * y becomes

```
push x
push 2
push y
multiply
subtract
```

Stack Machine Code

- Operations take operands from a stack
- Compact form
- A form of one-address code
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Stack Machine Code Advantages



Result is stored in a temporary! Explicit name for result.

```
push 2
push y
multiply
push x
subtract
```

Multiply pops two items off of stack and pushes result! Implicit name for result

Three Address Code

Different representations of three address code

- In general, three address code has statements of the form:

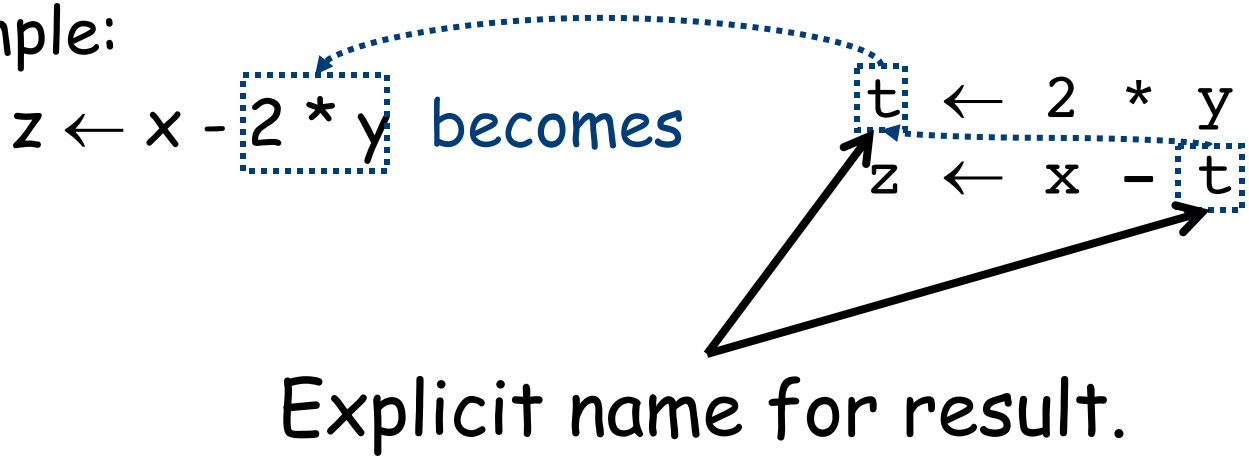
$$x \leftarrow y \ \underline{op} \ z$$

With 1 operator (*op*) and

(at most) 3 names (x, y, & z)

Three Address Code

Example:



Three Address Code Advantages

- Resembles many real (RISC) machines
- Introduces a new set of names
- Compact form

Three Address Code: Simple Array

Naïve representation of three address code

- Table of k * 4 small integers

```
load   r1, y
loadI  r2, 2
mult   r3, r2, r1
load   r4, x
sub    r5, r4, r3
```

RISC assembly code

Destination

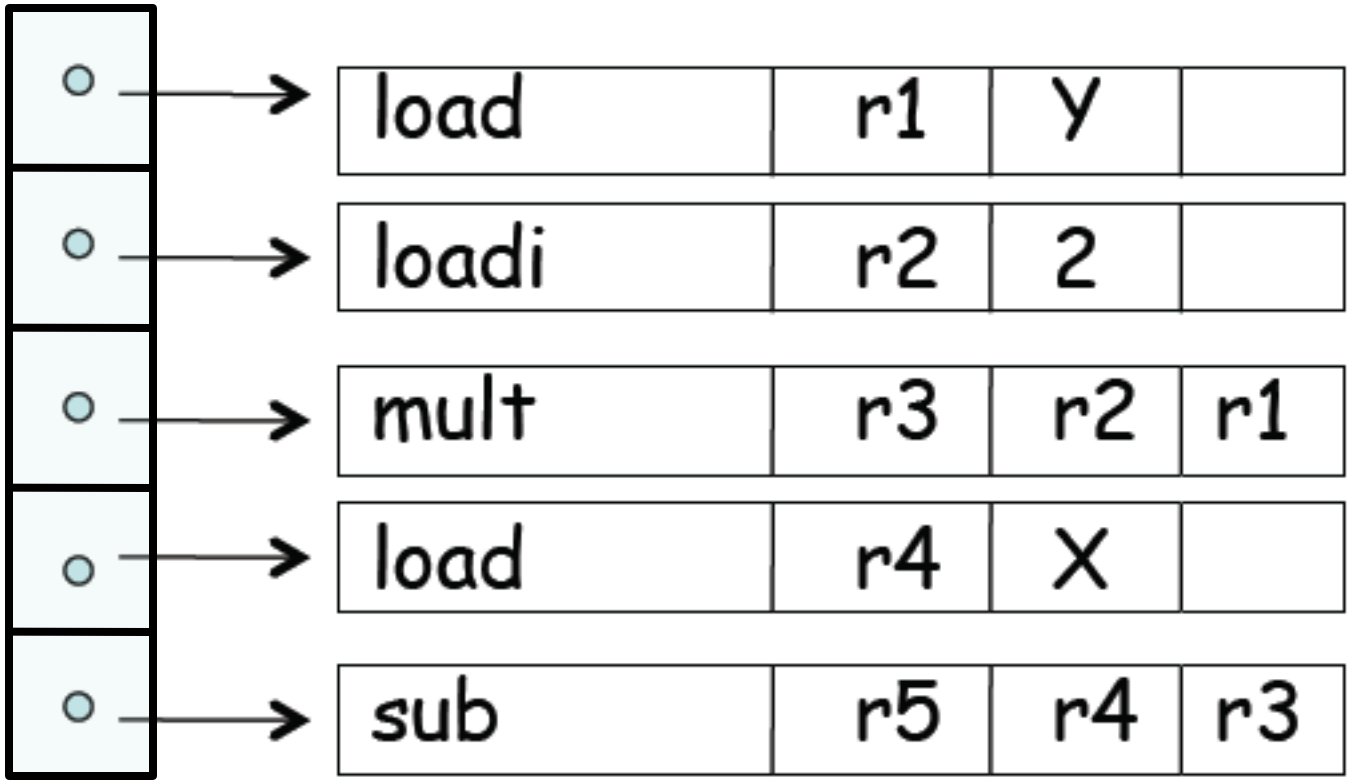
Two operands

load	1	y	
loadi	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Simple Array

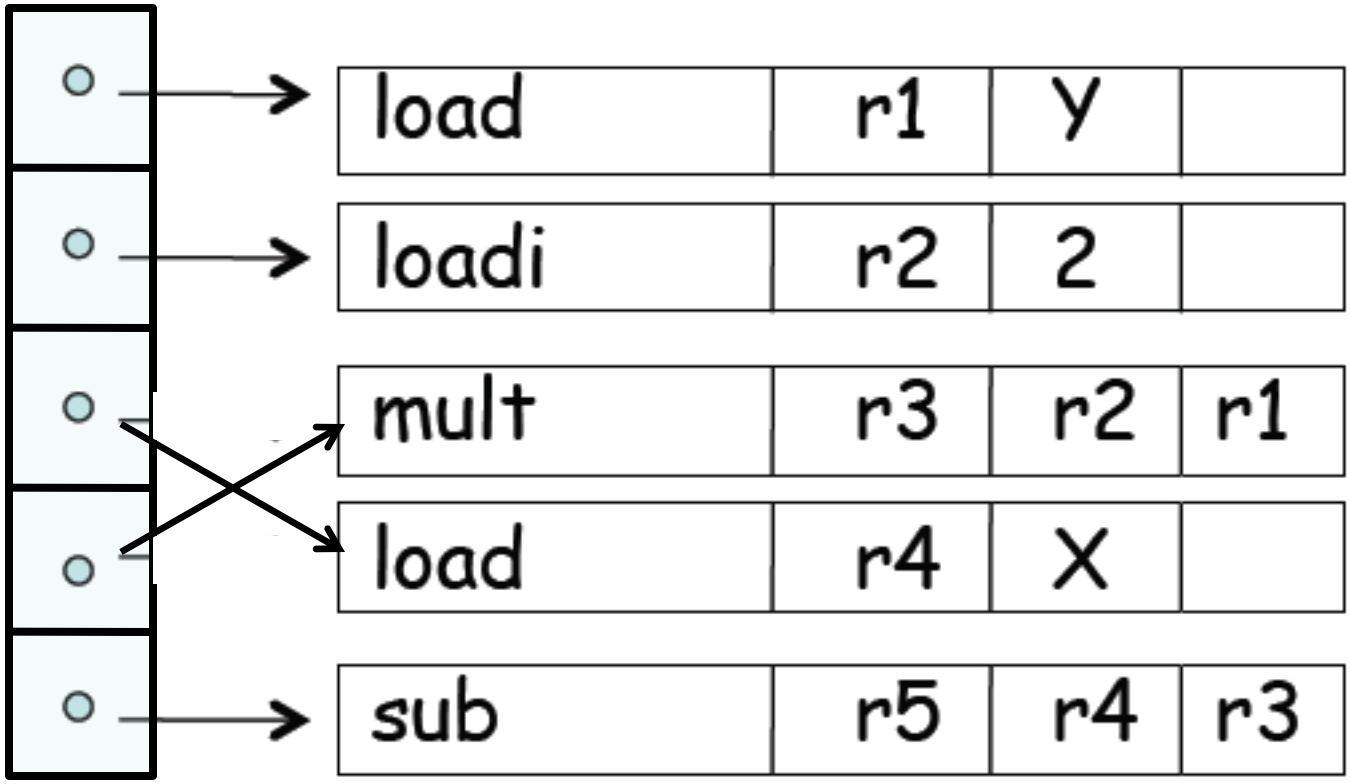
Three Address Code: Array of Pointers

- Index causes level of indirection
- Easy (and cheap) to reorder
- Easy to add (delete) instructions



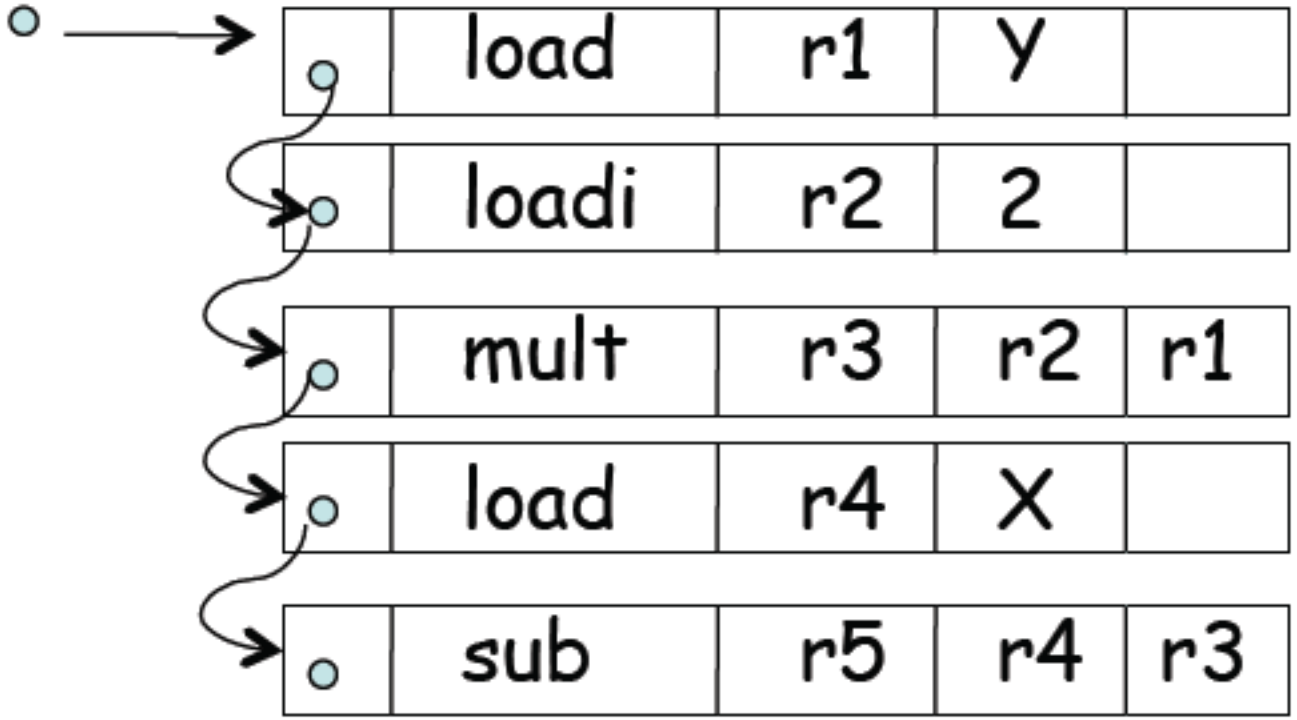
Three Address Code: Array of Pointers

- Index causes level of indirection
- Easy (and cheap) to reorder
- Easy to add (delete) instructions



Three Address Code: Linked List

- No additional array of indirection
- Easy (and cheap) to reorder than simple table
- Easy to add (delete) instructions

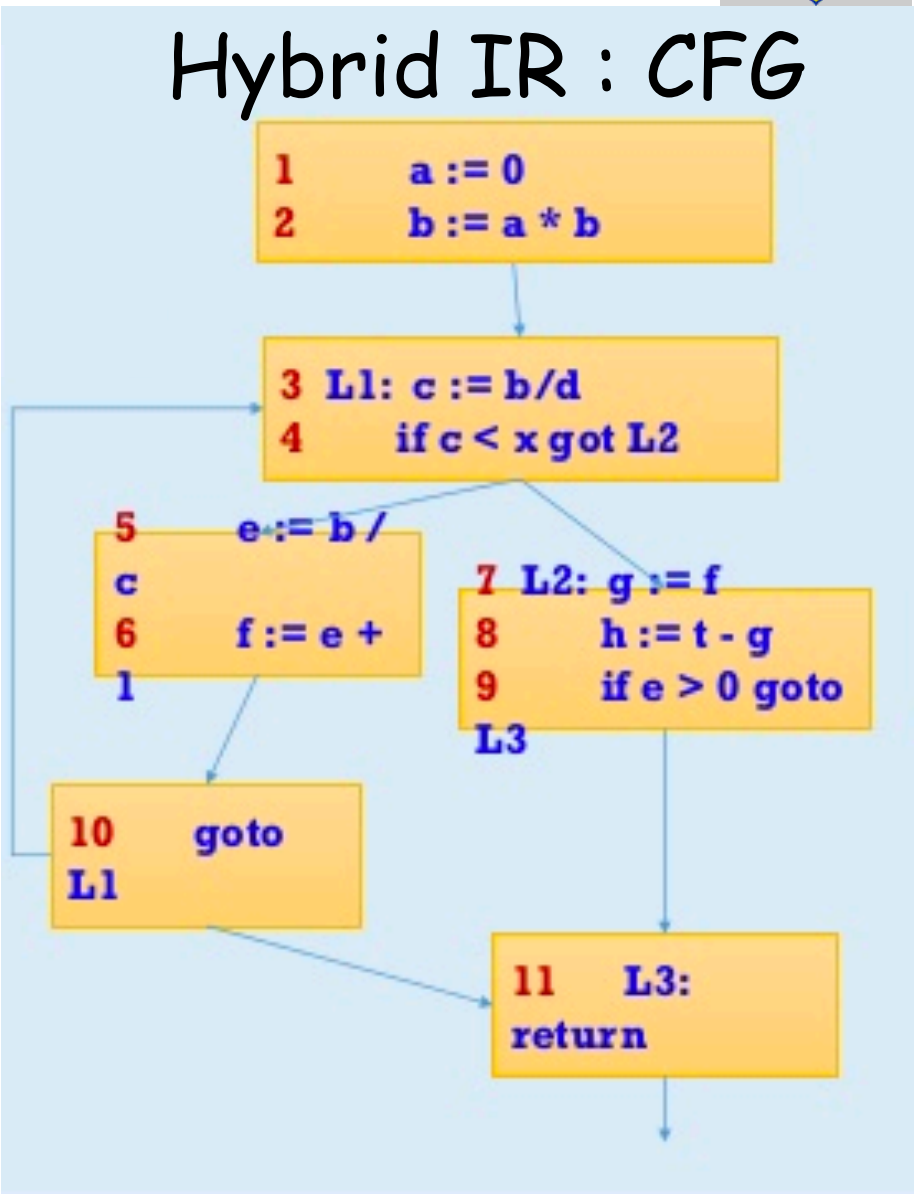


Control-Flow Graphs

Linear IR

```
1       a := 0
2       b := a * b
3  L1:  c := b/d
4       if c < x got L2
5       e := b / c
6       f := e + 1
7  L2:  g := f
8       h := t - g
9       if e > 0 goto L3
10      goto L1
11      L3: return
```

Hybrid IR : CFG



Control-Flow Graphs

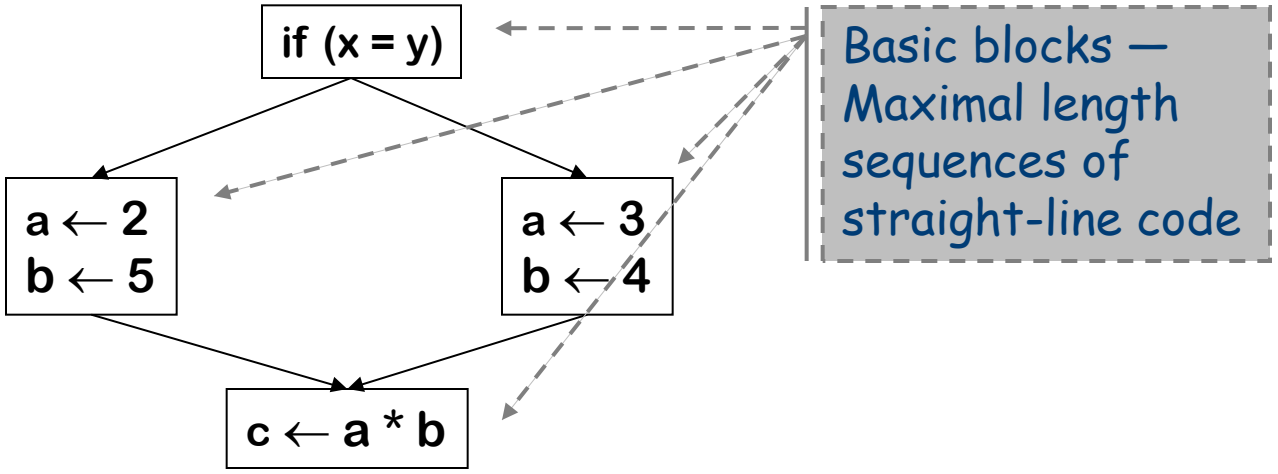
- Node: an instruction or sequence of instructions (a **basic block**)
  - → Two instructions i, j in same basic block *iff* execution of i *guarantees* execution of j
- Directed edge: *potential* flow of control
- Distinguished start node *Entry*
  - → First instruction in program

Control-flow Graph

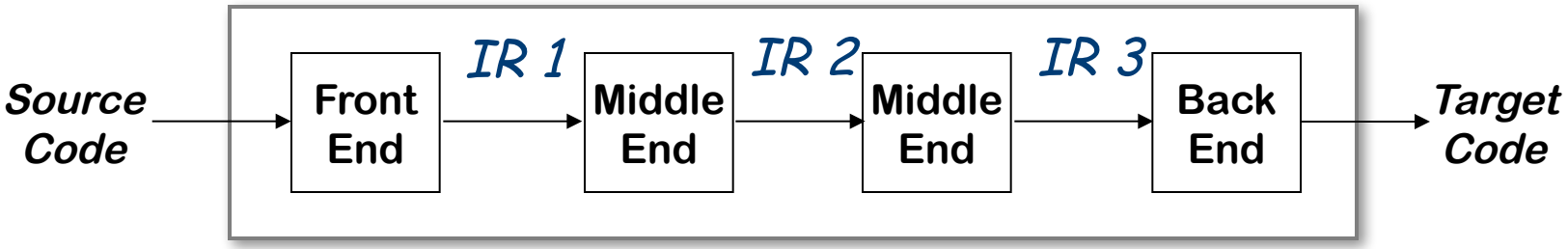
Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - → Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



Using Multiple Representations



- Repeatedly lower the level of the intermediate representation
  - →Each intermediate representation is suited towards certain optimizations

Memory Models

Two major models

- Register-to-register model
  - → Keep all values that can legally be stored in a register in registers
  - → Ignore machine limitations on number of registers
  - → Compiler back-end must insert loads and stores
- Memory-to-memory model
  - → Keep all values in memory
  - → Only promote values to registers directly before they are used
  - → Compiler back-end can remove loads and stores
- Compilers usually use register-to-register
  - → Reflects programming model
  - → Easier to determine when registers are used

The Rest of the Story…

Representing the code is only part of an *IR*

There are other necessary components

- Symbol table
- Constant table
  - → Representation, type
  - → Storage class, offset
- Storage map
  - → Overall storage layout
  - → Overlap information
  - → Virtual register assignments

Symbol Tables

Traditional approach to building a symbol table uses hashing

- One table scheme
  - → Lots of wasted space