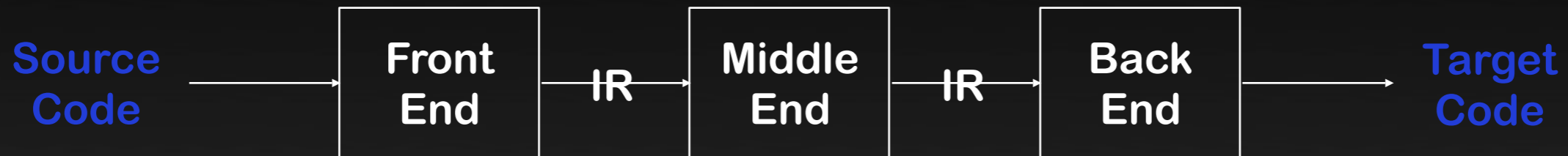


# Intermediate Representations

# Intermediate Representations



- Front end *produces* the intermediate representation (IR)
- Middle end *transforms* the IR:
  - equivalent version that runs more efficiently
- Back end *transforms* the IR
  - target architecture assembly language code
- IR encodes the compiler's knowledge of program
- Middle end usually consists of several passes

# Intermediate Representations

- IR design impacts the speed efficiency of the compiler
- Some important IR properties:
  - Ease of generation
  - Ease of manipulation
  - Resulting code size
  - Freedom of expression
  - Level of abstraction
- Importance of properties varies between compilers
  - Selecting an appropriate IR can be crucial!!

# Types of IR's

Three major categories

- Structural

- Graphically oriented
- Heavily used in source-to-source translators
- Tend to be large

Examples:  
Trees, DAGs

- Linear

- Pseudo-code for an abstract machine
- Simple, compact data structures
- Easier to rearrange

Examples:  
3 address code  
Stack machine code

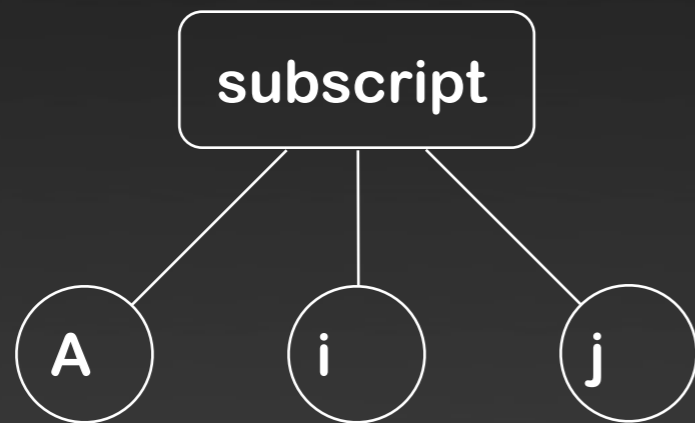
- Hybrid

- Combination of graphs and linear code

Example:  
Control-flow graph

# Level of Abstraction

- Detail level of IR impacts optimizations
- Ex.: representations of an array reference:



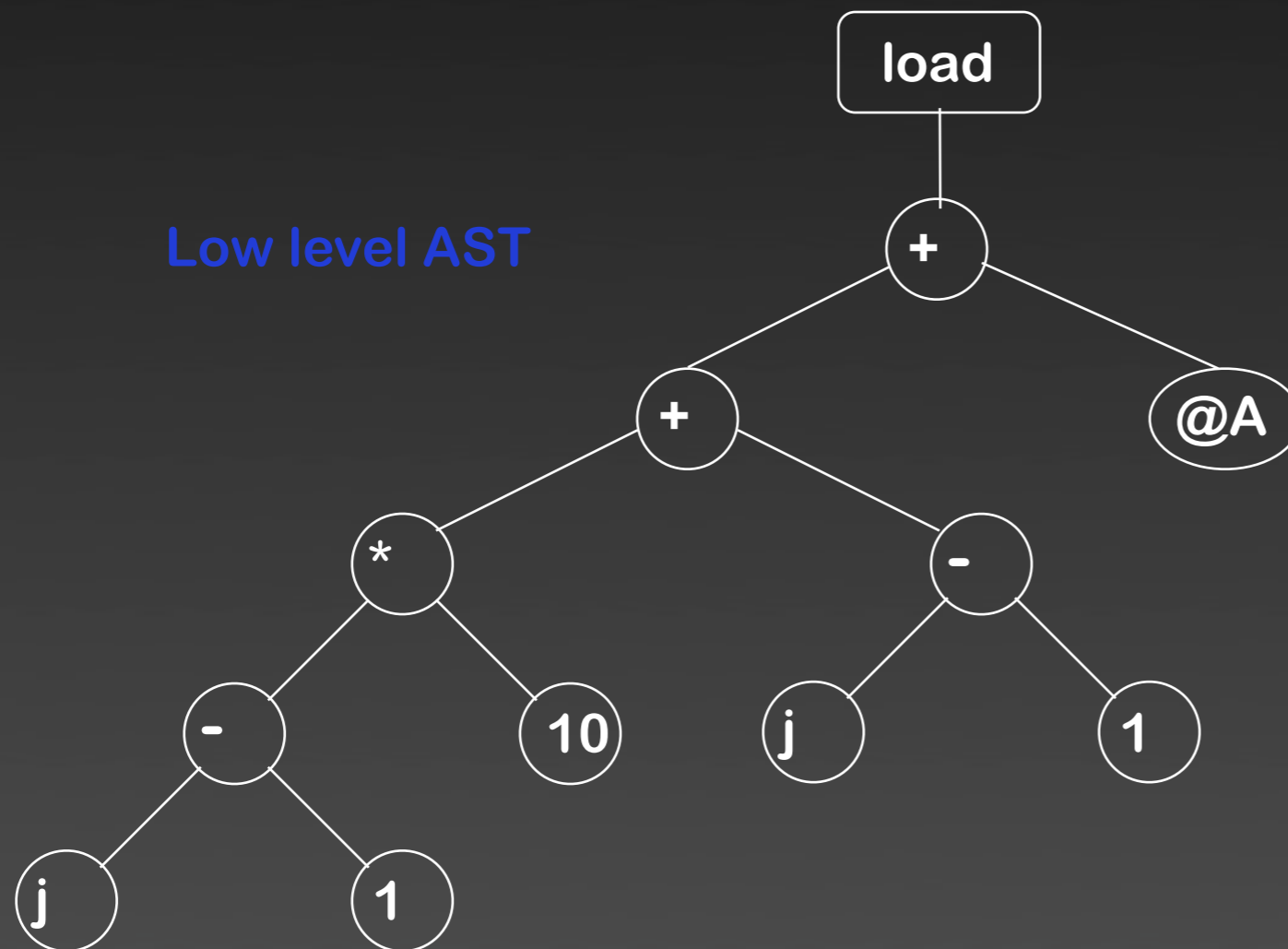
High level AST:  
Good for memory  
disambiguation

```
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
```

Low level linear code:  
Good for address calculation

# Level of Abstraction

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



Low level AST

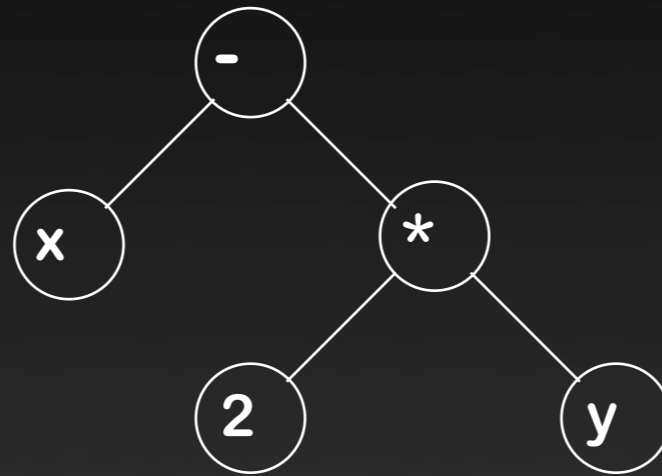
`loadArray A, i, j`

High level linear code

# Abstract Syntax Tree

*abstract syntax tree (AST)* - a parse tree with the nodes for (most) non-terminal nodes removed

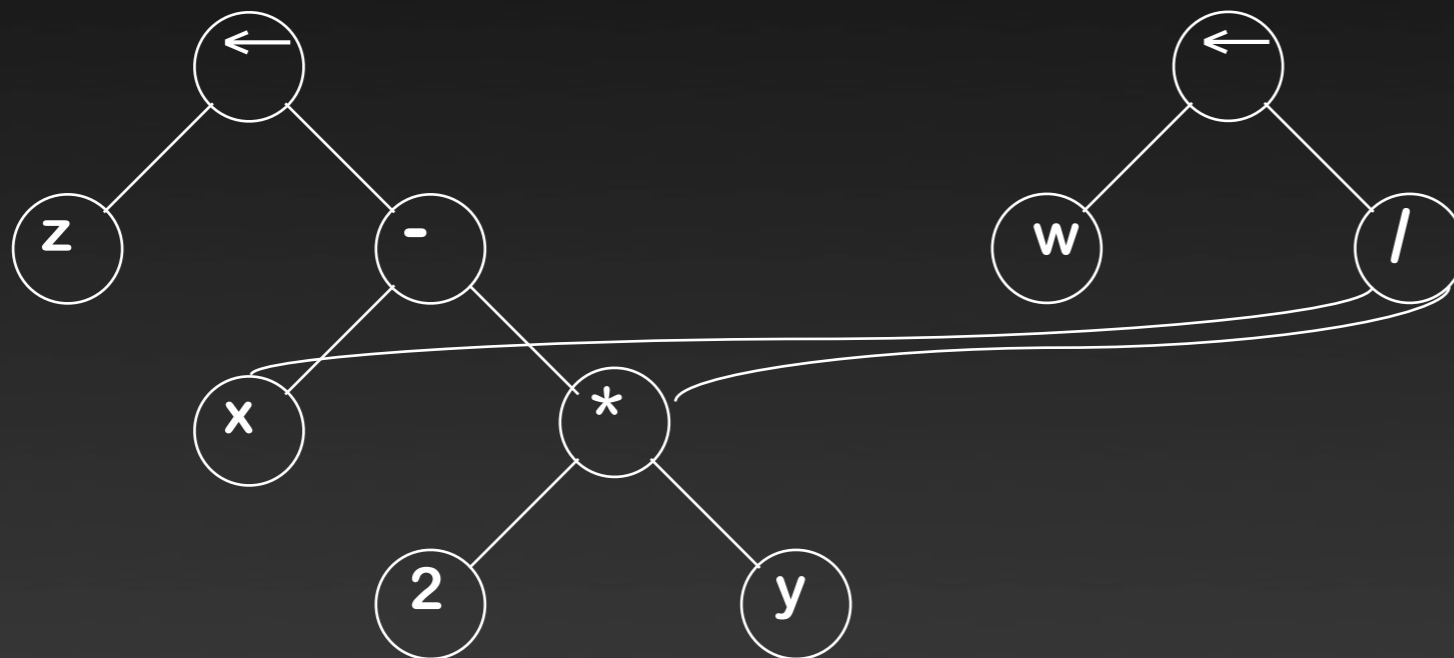
$x - 2 * y$



- Can use linearized form of the tree
  - $x \ 2 \ y \ * \ -$  in postfix form
  - $- \ * \ 2 \ y \ x$  in prefix form
  - Easier to manipulate than pointers

# Directed Acyclic Graph

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



$z \leftarrow x - 2 * y$   
 $w \leftarrow x / (2 * y)$

Same expression(s) twice mean that the compiler *might* arrange to evaluate them just once!

- Makes sharing explicit
- Encodes redundancy

# Stack Machine Code

Originally used for stack-based computers

- Example:

$x - 2 * y$  becomes

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

Implicit names take up  
no space, where  
explicit ones do!

Advantages:

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Useful when code transmitted over slow communication links (ex. Java bytecode over the Internet)

# Three Address Code

Several different representations of three address code

- Most three address code has statements of the form:

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

With 1 operator (op) and, at most, 3 names (x, y, & z)

Example:

$$z \leftarrow x - 2 * y$$

becomes

$$t \leftarrow 2 * y$$

$$z \leftarrow x - t$$

Advantages:

- Resembles many machines
- Introduces a new set of names
- Compact form

# Quadruples

Simple representation of three address code

- Table of  $k * 4$  values (often integers)
- Simple record structure
- Easy to reorder
- Explicit names

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

RISC assembly code

load	Y	t <sub>1</sub>	
loadi	2	t <sub>2</sub>	
mult	t <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>
load	X	t <sub>4</sub>	
sub	t <sub>4</sub>	t <sub>3</sub>	t <sub>2</sub>

Quadruples

# Three Address Code: Triples

- Index used as implicit name
- less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	loadI	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Implicit names take no space!

# Static Single Assignment Form

- The main idea: each name defined exactly once
- Introduce  $\phi$ -functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
  x ← x + 1
  y ← y + x
```

SSA-form

```
x0 ← ...
y0 ← ...
if (x0 > k) goto next
loop: x1 ←  $\phi(x_0, x_2)$ 
      y1 ←  $\phi(y_0, y_2)$ 
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < k) goto loop
next: ...
```

Strengths of SSA-form:

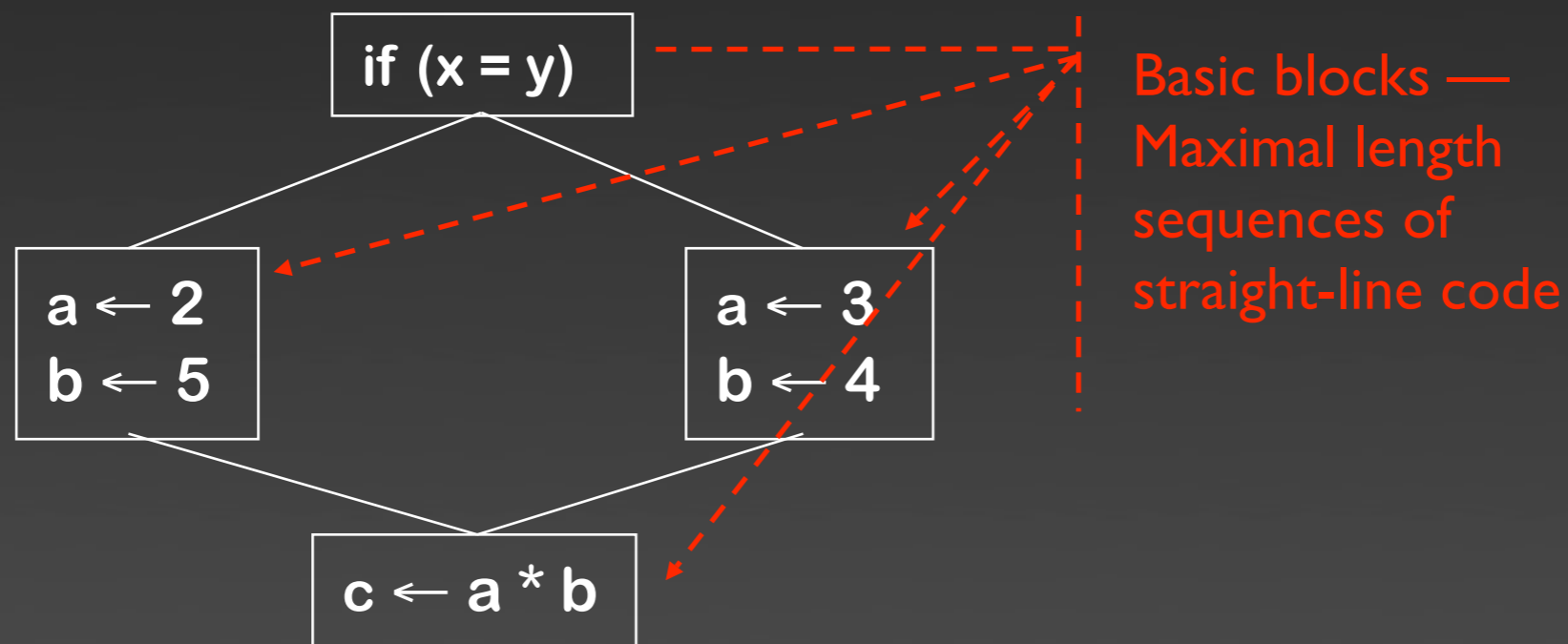
- Sharper analysis
- $\phi$ -functions give hints about placement
- (sometimes) faster algorithms

# Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can use quads or any other linear representation
- Edges in the graph represent control flow

Example:



# Memory Models for IR

- Register-to-register model
  - Keep all possible values 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
  - Place values in registers as they are being used
  - Compiler back-end can remove loads and stores
- Compilers for RISC usually use register-to-register
  - Reflects programming model
  - Easier to determine when registers are used