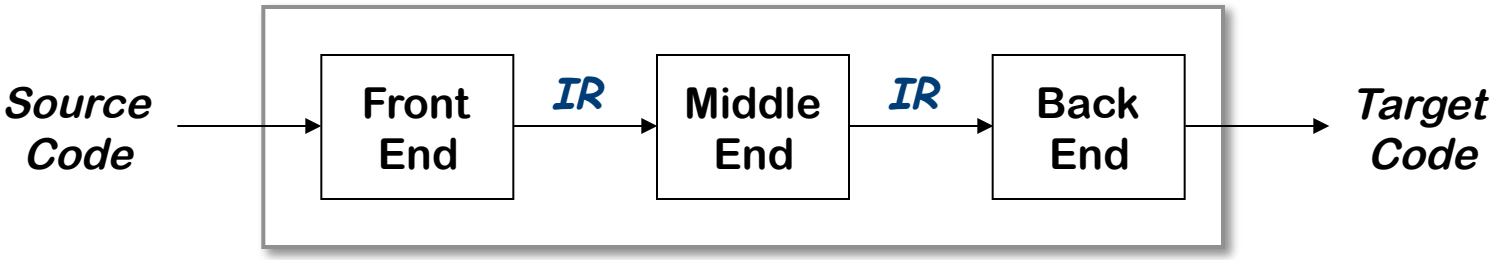
Learning Objectives

- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- Understand SSA form and it's advantages
- List the pros and cons of various memory models

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 several passes

Intermediate Representations

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

- Some important *IR* properties
  - Ease of generation
  - Ease of manipulation
  - Procedure size
  - Freedom of expression
  - Level of abstraction

- The importance of different properties varies between compilers
  - Selecting an appropriate *IR* for a compiler is critical

Types of Intermediate Representations

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
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange

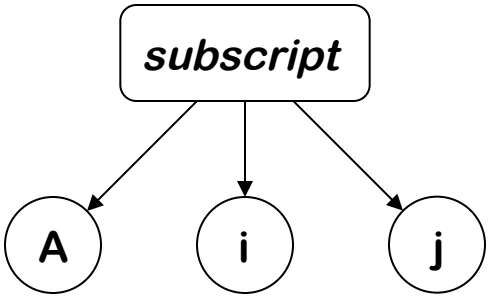
  Examples:
  3 address code
  Stack machine code

- Hybrid
  - Combination of graphs and linear code
  - Example: control-flow graph

  Example:
  Control-flow graph

Level of Abstraction

- The level of detail exposed in an *IR* influences the profitability and feasibility of different optimizations.
- Two different representations of an array reference:



High level AST:
Good for memory disambiguation

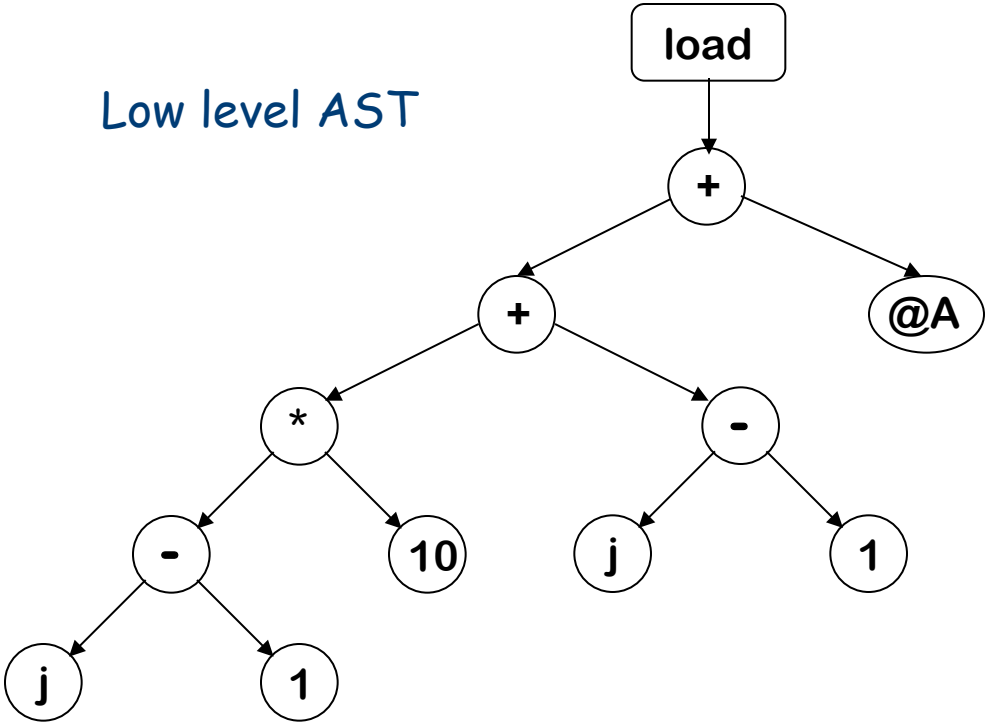
```
loadI  1       => r1
sub    rj, r1  => r2
loadI  10      => r3
mult   r2, r3  => r4
sub    ri, r1  => r5
add    r4, r5  => r6
loadI  @A      => r7
add    r7, r6  => r8
load   r8      => rAij
```

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 – see example below

Low level AST




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

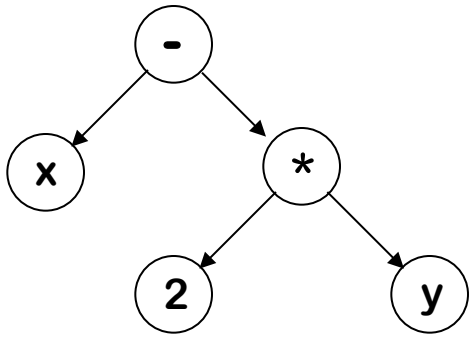
High level linear code

Learning Objectives

- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- Understand SSA form and it's advantages
- List the pros and cons of various memory models

Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed



x - 2 * y

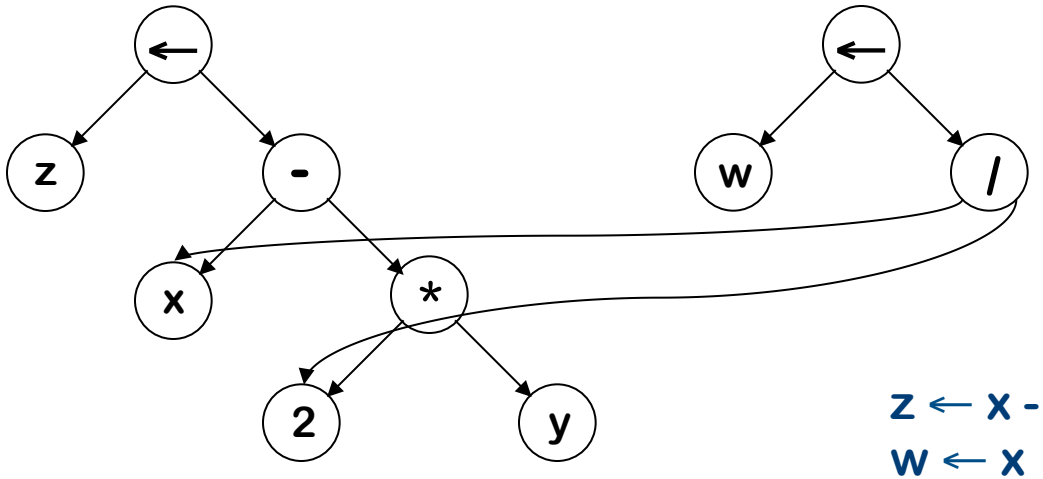
- Can use linearized form of the tree
  - Easier to manipulate than pointers

    `x 2 y * -` in postfix form

    `- * 2 y x` in prefix form
- S-expressions (Scheme,Lisp) are (essentially) ASTs

Directed Acyclic Graph

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



z ← x - 2 * y
w ← 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
```

Advantages

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

Useful where code is transmitted over slow communication links (*the net*)

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

Three Address Code

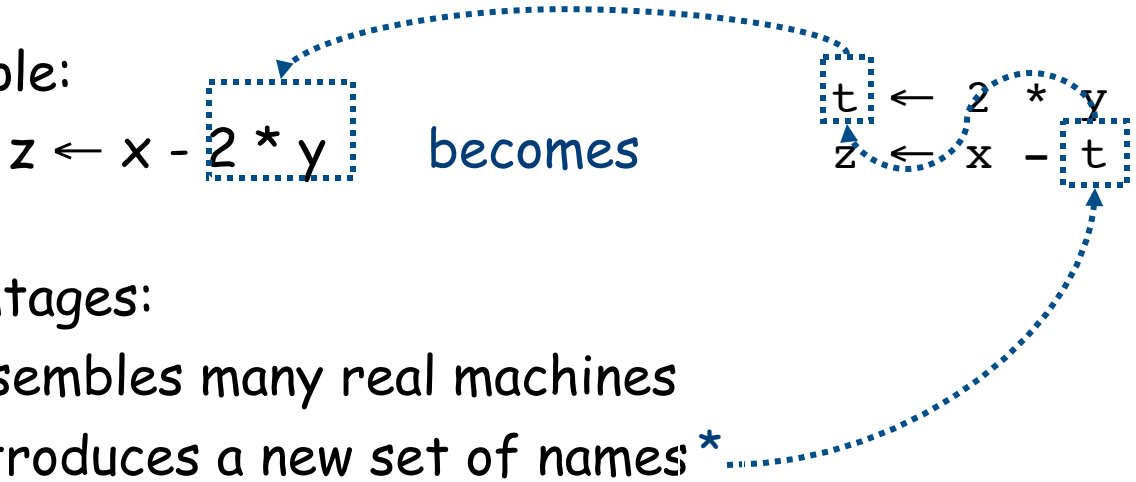
Several different representations of three address code

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

$$x \leftarrow y \; op \; z$$

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

Example:



Advantages:

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

Three Address Code: Quadruples

Naïve representation of three address code

- Table of k * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

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

RISC assembly code

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

Quadruples

Three Address Code: Triples

- Index used as implicit name
- 25% 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 occupy no space

Remember, for a long time, 640Kb was a lot of RAM

Three Address Code: Indirect Triples

- List first triple in each statement
- Implicit name space
- Uses more space than triples, but easier to reorder

Stmt List	Implicit Names	Indirect Triples		
(100)	(100)	load	y	
(105)	(101)	loadI	2	
	(102)	mult	(100)	(101)
	(103)	load	x	
	(104)	sub	(103)	(102)

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  - In the past compile-time space was critical
  - Today, speed may be more important

Two Address Code

- Allows statements of the form

  $x \leftarrow x \; op \; y$

  Has 1 operator (*op*) and, at most, 2 names (x and y)

Example:

$z \leftarrow x - 2 * y$ becomes

$t_1 \leftarrow 2$
$t_2 \leftarrow$ load y
$t_2 \leftarrow t_2 * t_1$
$z \leftarrow$ load x
$z \leftarrow z - t_2$

- Can be very compact

Problems

- Machines no longer rely on destructive operations
- Difficult name space
  - Destructive operations make reuse hard
  - Good model for machines with destructive ops (PDP-11)

Learning Objectives

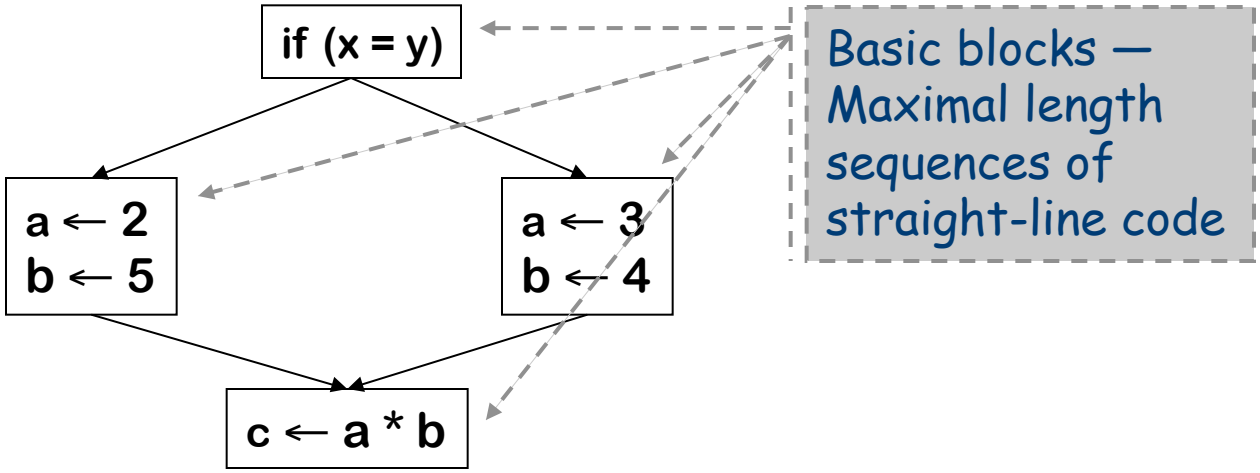
- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- Understand SSA form and it's advantages
- List the pros and cons of various memory models

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



Algorithm for converting linear code to CFG

- Find leaders
  - Identify all nodes that are labels/targets for jumps as leaders

- Break CFG into (leader, last) pairs
  - For each leader, traverse program until we come to another leader. Terminate the block and record (leader, last) pair.
  - If the last instruction is a conditional branch to l1, l2, then record edges from the current block to blocks l1 and l2
  - If the last instruction is a unconditional branch to label 'l', then add an edge from the current block to 'l'
  - If the last instruction is an indirect jump (e.g., jmp r1), then
    - → Strategy 1 (conservative): Add an edge to every basic block
    - → Strategy 2 (precise): Add an edge to only those basic blocks that are targets of the jump – requires pointer/alias analysis to resolve

Add an entry node and exit node and the corresponding arcs

Static Single Assignment Form

- The main idea: each name defined exactly once
- Introduce ϕ-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 ← φ(x0,x2)
      y1 ← φ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < k) goto loop
next:   …
```

Strengths of SSA-form

- Sharper analysis
- ϕ-functions give hints about placement
- Some facts are obvious (e.g., live variables)
- Compact representation of data-flow facts

Algorithm to convert code to SSA (Naïve algorithm)

- **Non-obvious construction algorithm**
  - Will examine this algorithm later in this course
  - Naïve algorithm inserts too many redundant phi functions

- **Naïve algorithm**
  - Traverse the code in linear order
  - The first time you come to a variable, no action is needed
  - When you come to a previously defined variable, rename it to a unique name and replace all references with the new name
  - If you come to a Y-point in the code (i.e., control flow convergence), insert a phi node for every variable used in the downstream computations, and add the predecessor block's latest values as operands of the phi instruction

  - Not as simple as it looks

Learning Objectives

- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- Understand SSA form and it's advantages
- List the pros and cons of various memory models

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

Pros of Register-to-Register memory Model

- Compilers for RISC machines usually use register-to-register
  - Reflects programming model
  - Easier to determine when registers are used
  - Does not limit number of registers in the target
  - Easy to represent data-flow facts about the program (i.e., a value that is safe to move to a register is one that is not indirectly modified, through a pointer, for example)

Cons of Register-to-Register Memory Model

- Pressure on downstream passes and register allocator
  - Additional loads/stores required
- Can provide misleading information about program's performance at the intermediate code level
  - Cannot reason about memory pressure, cache misses etc.
  - Not a good match for register-constrained machines
- Handling memory references may be cumbersome at the IR level as the default mode is to operate on registers

Learning Objectives

- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- List the pros and cons of various memory models