Intermediate Representations (Objectives)

- Given an intermediate representation, the students will be able to describe the representation's advantages and disadvantages related to context-sensitive analysis and code-improving transformations.

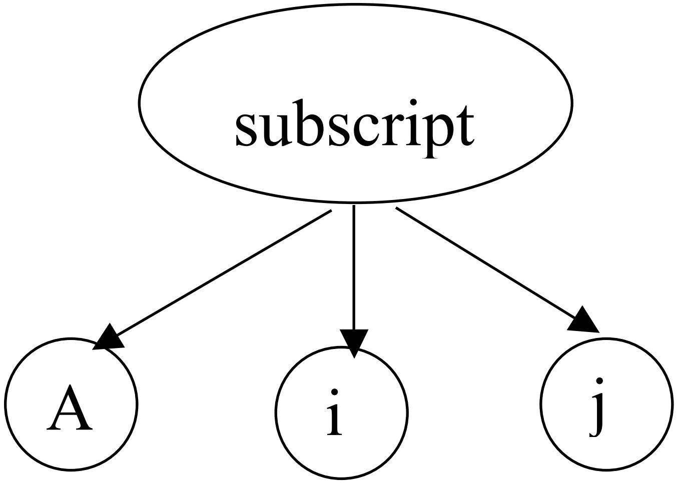
Motivation

- The multi-pass nature of a compiler motivates the need for different intermediate representations of a program.
- The intermediate representation (IR) includes:
  - some form of the actual program code
  - auxiliary tables
    - symbol table
    - constant table
    - label table
- The compiler will need different representations during different phases.

Taxonomy

- Graphical IRs
  - encode information about a program in a graph
  - abstract syntax tree
  - control-flow graph
- Linear IRs
  - resemble simple, assembly-like operations for some abstract machine
  - simple sequence of operations
    - Java bytecode
    - three-address code
- Hybrid IRs
  - combine elements of both
    - linear IR to represent sequence of operations plus a graphical IR to represent the control-flow of a program.

Level of Abstraction



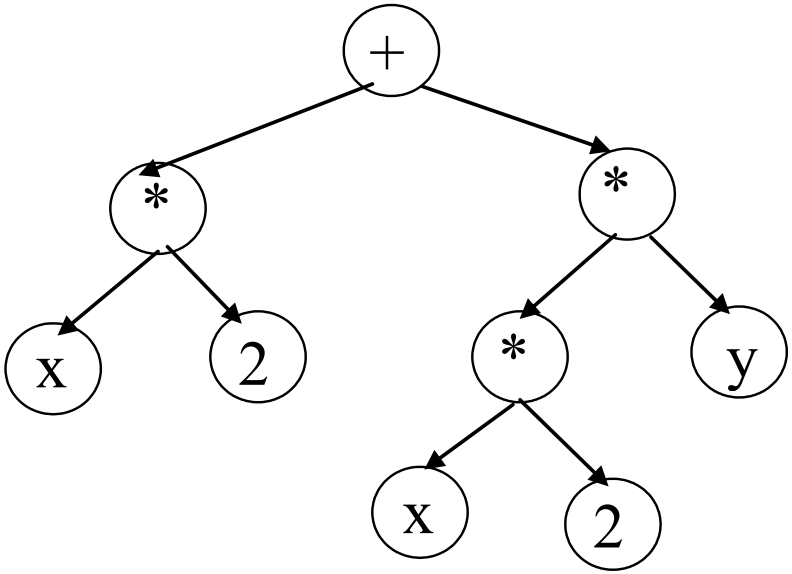
- int A[1..10,1..10]
- reference A[i,j]

loadi	1	$\Rightarrow r_1$
sub	r_j, r_1	$\Rightarrow r_2$
loadi	10	$\Rightarrow r_3$
mul	r_2, r_3	$\Rightarrow r_4$
sub	r_i, r_1	$\Rightarrow r_5$
add	r_4, r_5	$\Rightarrow r_6$
loadi	@A	$\Rightarrow r_7$
loadAO	r_7, r_6	$\Rightarrow r_{Aij}$

AST

- Good for CSA, not optimization
- redundancy in representation

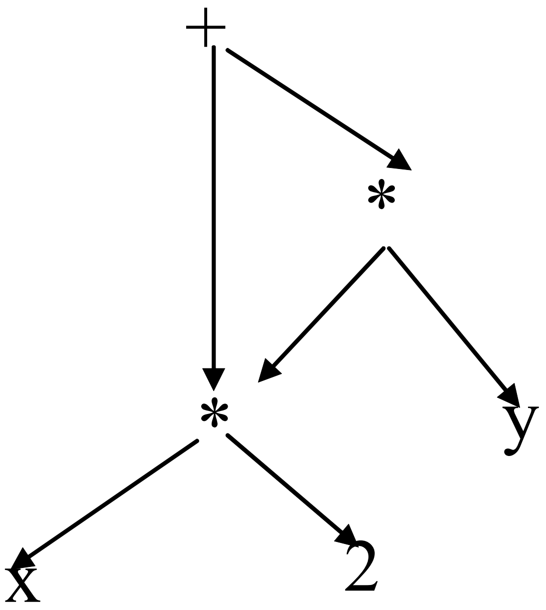
x * 2 + x * 2 * y



Directed Acyclic Graphs

- eliminates redundancy by using a graph instead of a tree
- Good for simple optimization, not for CSA

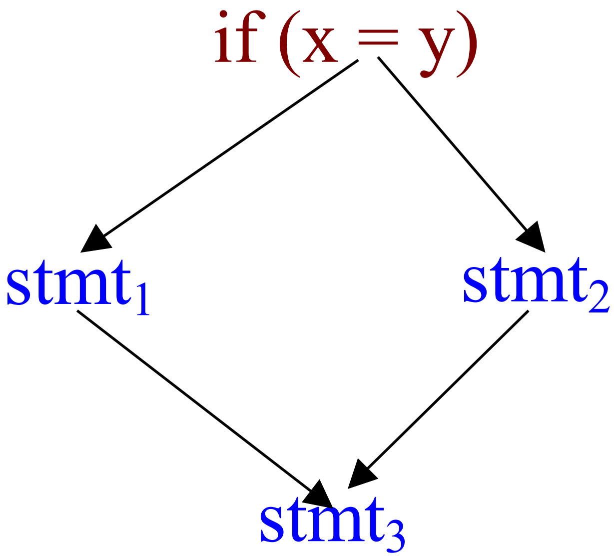
x * 2 + x * 2 * y



Control-Flow Graphs

- models the way that control transfers between single-entry, single-exit sequences of instructions (basic blocks)
- clean representation of all of the run-time possibilities for the paths taken through a program.
- Useful for optimization, not CSA

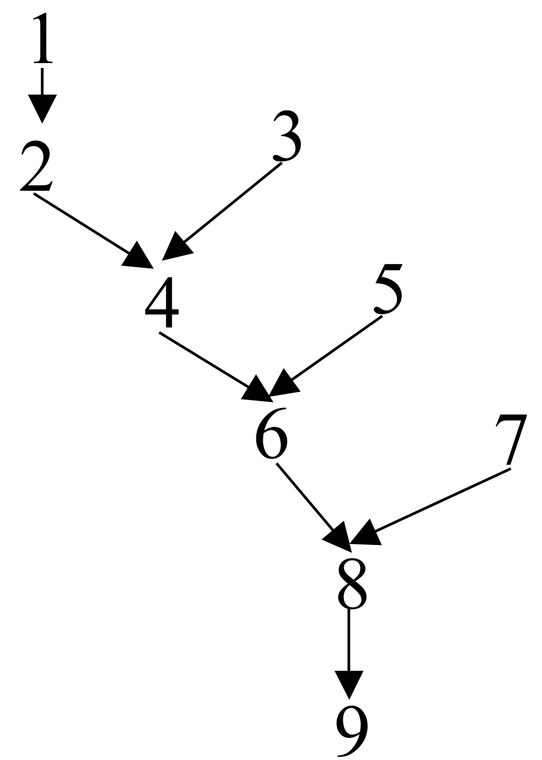
if (x = y)
  then $stmt_1$;
  else $stmt_2$;
$stmt_3$;



Data-Dependence Graph

- Encode the flow of data between operations
- Difficult for CSA, good for optimization

1.	loadAI	r_0,0	$\Rightarrow r_1$
2.	add	r_1, r_1	$\Rightarrow r_1$
3.	loadAI	r_0, 8	$\Rightarrow r_2$
4.	mult	r_1, r_2	$\Rightarrow r_1$
5.	loadAI	r_0, 16	$\Rightarrow r_2$
6.	mult	r_1, r_2	$\Rightarrow r_1$
7.	loadAI	r_0, 24	$\Rightarrow r_2$
8.	mult	r_1, r_2	$\Rightarrow r_1$
9.	storeAI	r_1	$\Rightarrow r_0$, 0



One-address Code

- stack machine code
  - operations manipulate the stack
- compact representation
- Example: Java bytecode
- Why is a compact representation important to Java?
- Difficult for CSA and optimization

- x - 2 * y

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

Two-address Code

- a single operator and at most two names
- operation overwrites one of its operands if three names are required
- Good for PDP-11 which used two address instructions
- Destruction of one operand causes additional operations to preserve it if necessary.
- Not good for either CSA or optimization

- $x - 2 * y$

loadi	$2 \Rightarrow t_1$
load	$y \Rightarrow t_2$
mult	$t_2 \Rightarrow t_1$
load	$x \Rightarrow t_3$
sub	$t_1 \Rightarrow t_3$

Three-address Code

- form
  $x \langle op \rangle y \Rightarrow z$
- good for optimizations
  - resembles machine instructions
  - For load/store architectures it can model values in registers
- not good for CSA
- example
  - quadruples

- $x - 2 * y$

loadi	2	$\Rightarrow t_1$
load	y	$\Rightarrow t_2$
mult	t_1, t_2	$\Rightarrow t_3$
load	x	$\Rightarrow t_4$
sub	t_4, t_3	$\Rightarrow t_5$

Name Spaces for Intermediates

- Compiler generates IR names for variables and compiler temporaries
- Effects quality of code and speed of the compiler
- Some choices
  - new name for each temporary
    - space and speed problems
  - reuse names as soon as previous use not needed
    - hurts optimization
  - same name for lexically identical operations
    - reveals redundancy
    - not too much space

Naming

- Consider two reference to A[i] in a program
- first reference

load	i	$\Rightarrow r_1$
loadI	8	$\Rightarrow r_2$
mult	r_1, r_2	$\Rightarrow r_3$
loadAI	@A	$\Rightarrow r_4$
add	r_3, r_4	$\Rightarrow r_5$
loadAO	$0, r_5$	$\Rightarrow r_6$

- second reference in another part of the code

load	i	$\Rightarrow r_4$
loadI	8	$\Rightarrow r_3$
mult	r_4, r_3	$\Rightarrow r_7$
loadAI	@A	$\Rightarrow r_9$
add	r_7, r_9	$\Rightarrow r_1$
loadAO	$0, r_1$	$\Rightarrow r_2$

- What is the impact if lexically identical operations have the same name?

SSA Form

- each variable is defined once → each use has one reaching definition
- use ϕ-nodes to merge multiple definitions reaching a single point

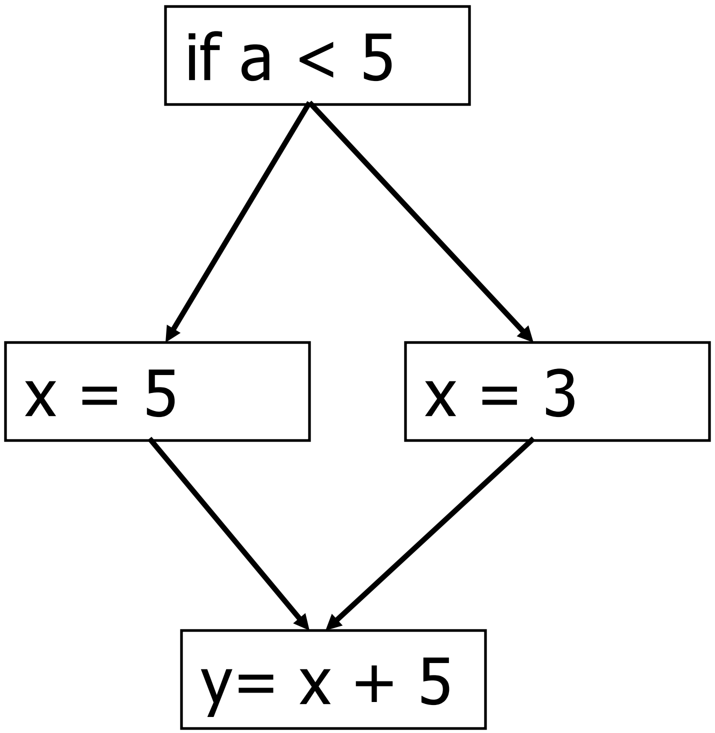
$v = 4$
$x = v + 5$
$v = 6$
$y = v + 7$

becomes

$v_0 = 4$
$x_0 = v_0 + 5$
$v_1 = 6$
$y_0 = v_1 + 7$

Control Flow

- What do we do when there are multiple definitions reaching a single point?
- In the example to the right, which definition of x is used at in the computation of y?

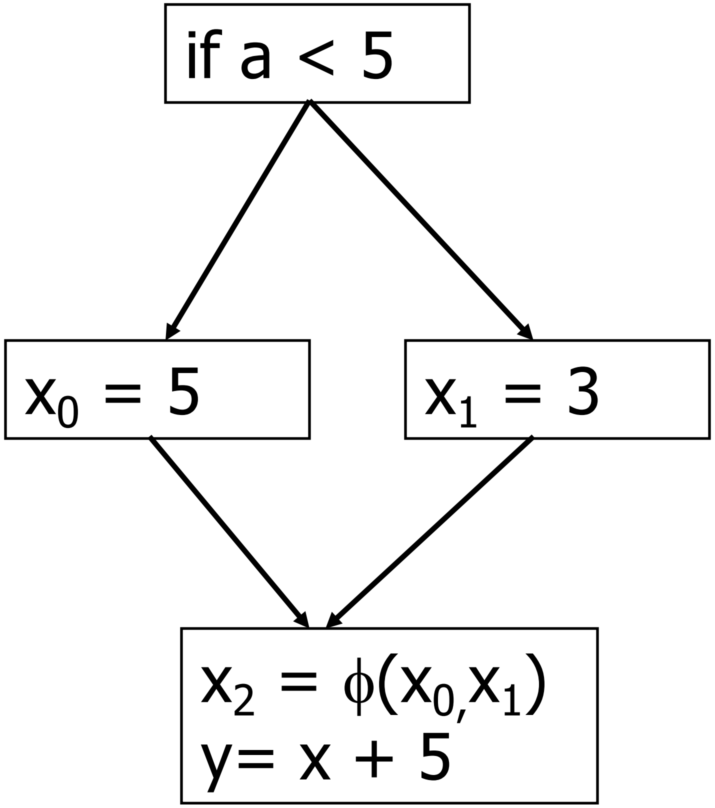


ϕ-nodes

- Def^n: Consider a block b in the CFG with predecessors $\{p_1, p_2, \ldots, p_n\}$ where $n > 1$. A ϕ-node

  $T_0 = \phi(T_1, T_2, \ldots, T_n)$

  in b gives the value of T_i to T_0 on entry to b if the execution path leading to b has p_i as the predecessor to b.



Other Intermediate Forms

- There are other intermediate forms that are used for optimization
  - control-dependence graph
  - program dependence graph
  - program dependence web