Lecture 7

Global Common Subexpression Elimination; Constant Propagation/Folding

I. Available Expressions Analysis

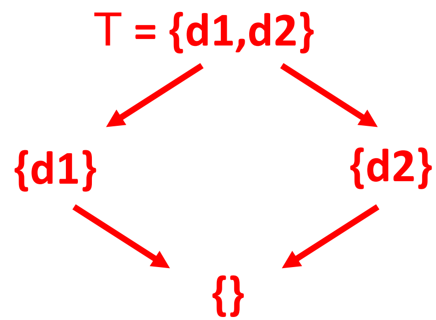
II. Eliminating CSEs

III. Constant Propagation/Folding

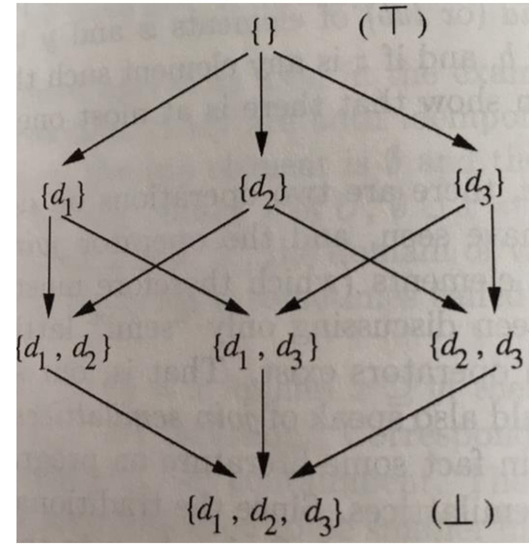
ALSU 9.2.6, 9.4

Review: A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values **V**
  - meet operator $\wedge$
  - Top T
  - finite descending chain?



Meet Operator:
Intersection



Meet Operator:
Union

Review: A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values **V**
  - meet operator $\wedge$
  - Top T
  - finite descending chain?
- **Transfer functions**
  - function of a basic block **f: V → V**
  - closed under composition
  - meet-over-paths **MOP**
  - monotone
  - distributive?

For each node *n*: MOP(*n*) = $\wedge f_{p_i}$(T), for all paths p_i in data-flow graph reaching *n*.

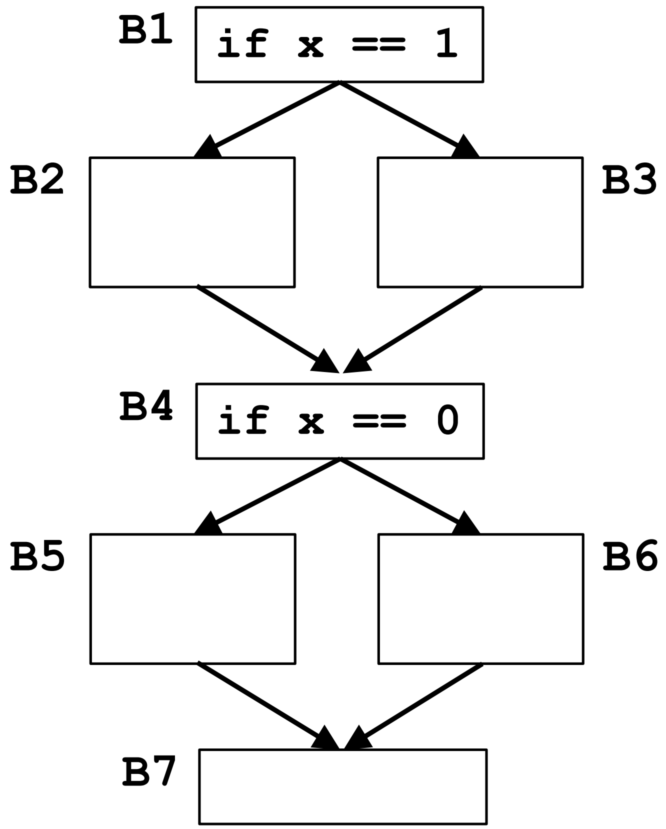
If data flow framework is monotone (i.e., x ≤ y implies f(x) ≤ f(y)) then if the algorithm converges, IN[b] ≤ MOP[b] *, so analysis is ? safe.

Data flow framework (monotone) converges if its lattice has ? a finite descending chain.

If data flow framework is distributive (i.e., f(x $\wedge$ y) = f(x) $\wedge$ f(y)) then if the algorithm converges, IN[b] = MOP[b] *, so ? precision is high.

* for backward analysis OUT[b]

Review: MOP considers more paths than Perfect



Perfect considers only:

B1-B2-B4-B6-B7 (i.e., x=1)
B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths

B1-B2-B4-B5-B7
B1-B3-B4-B6-B7

What changes if $x \in \{0,1,2\}$ **?**

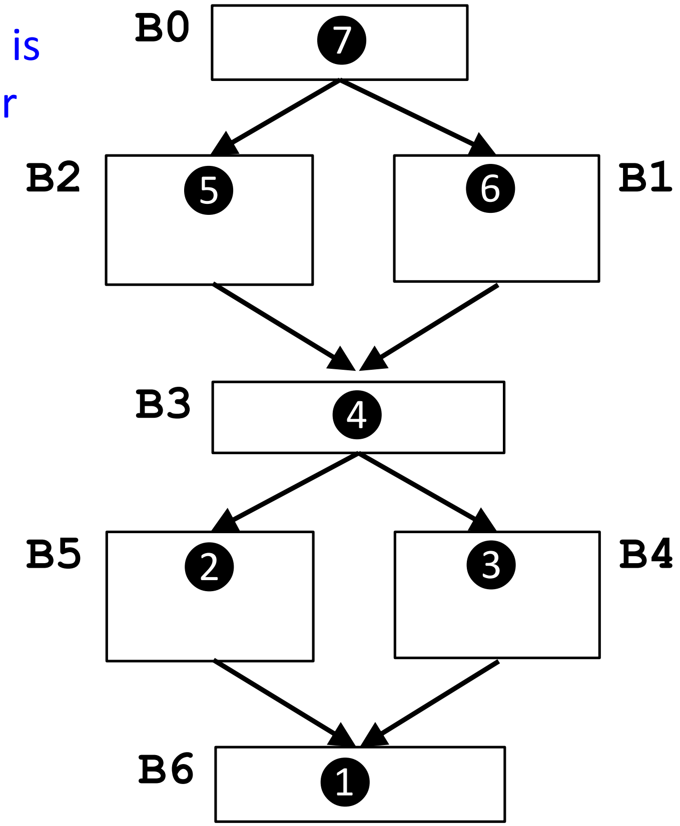
B1-B3-B4-B6-B7 is also a Perfect path

Assume $x \in \{0,1\}$ and B2 & B3 do not update x

Review: A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values **V**
  - meet operator $\wedge$
  - Top T
  - finite descending chain?
- **Transfer functions**
  - function of a basic block **f: V → V**
  - closed under composition
  - meet-over-paths **MOP**
  - monotone
  - distributive?
- **Algorithm**
  - initialization step (entry/exit, other nodes)
  - visit order: **rPostOrder**
  - depth of the graph

B0,B1,…,B6 is rPostOrder



Number of iterations = number of back edges in any acyclic path + 2

[ALSU 9.6.7]

Review: Speed of Convergence

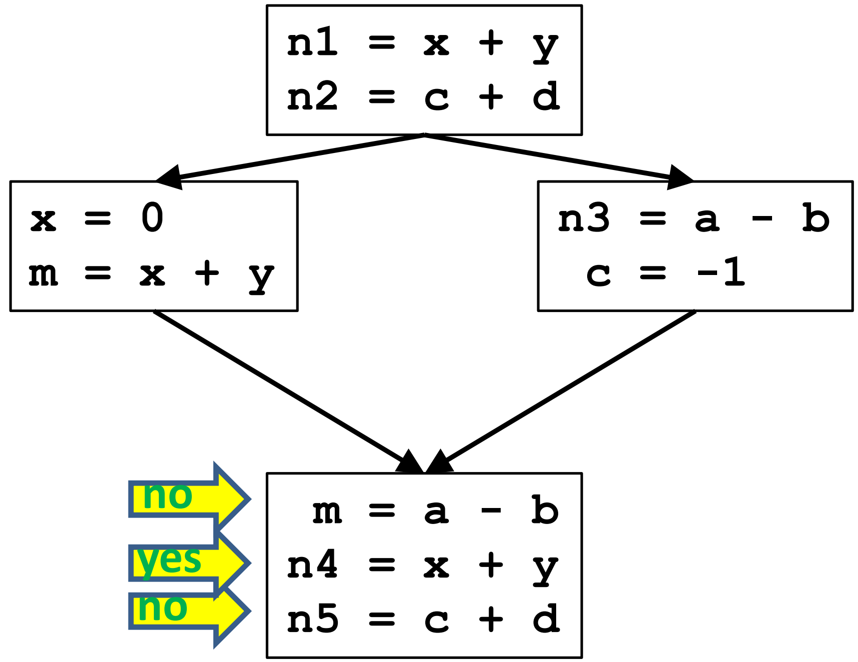
- **If cycles do not add information***
  - information can flow in one pass down nodes of increasing order number:
    - e.g., $\underbrace{1 \text{ -> } 4 \text{ -> } 5 \text{ -> } 7}_{\text{first pass}}$ -> 2 -> 6 ...
  - passes determined by number of back edges in the path
    - essentially the nesting depth of the graph
  - Number of iterations = number of back edges in any acyclic path + 2
    - (2 are necessary even for acyclic CFGs)
    - (2 not 1 since need a last pass where nothing changed)

* E.g., if a defn *d* in node n_1 reaches a node n_k along a path that contains a cycle (i.e., a repeated node), then the cycle can be removed to form a shorter path from n_1 to n_k such that *d* reaches n_k.

```
L: a = b
   b = c
   c = 1
   goto L
```

Example where cycles add information, for constant propagation

I. Available Expressions Analysis

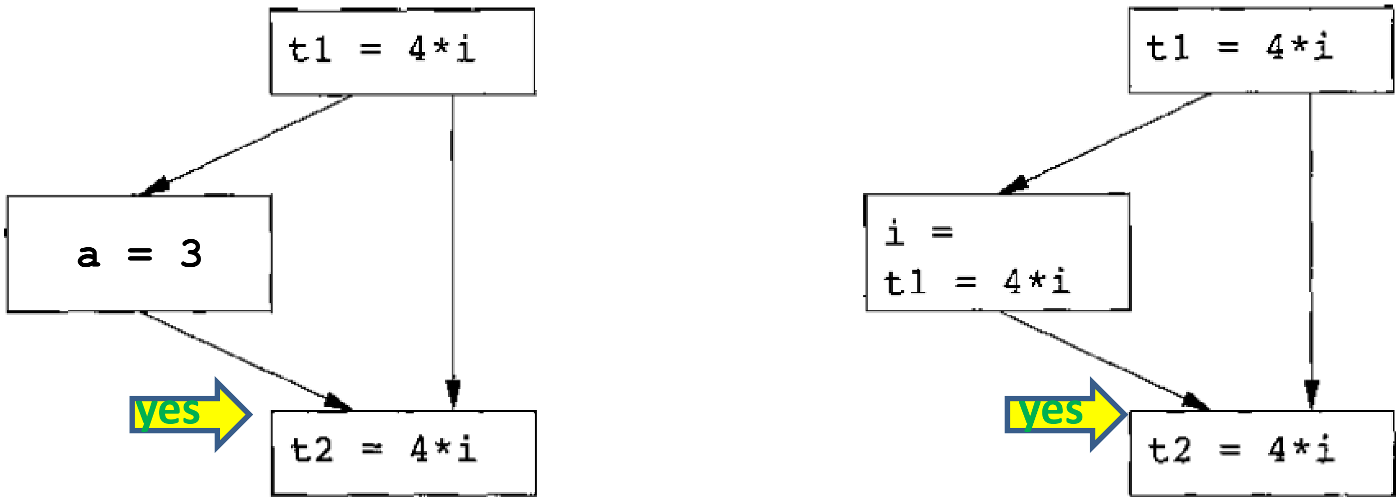


Is right-hand-side expression available?

Part of Assignment #1

- **Availability of an expression E at point P**
  - DEFINITION: Along every path to P in the flow graph:
    - E must be evaluated at least once
    - no variables in E redefined after the last evaluation
  - Observation: E may have different values on different paths (e.g., **`x+y`** above)

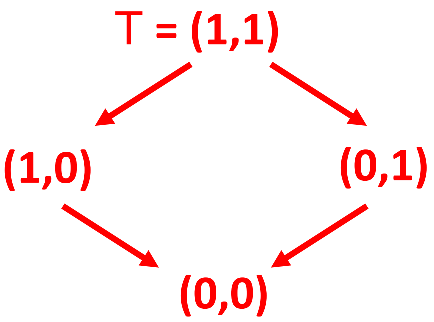
Available Expressions Example



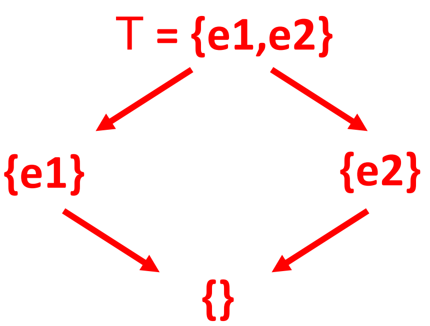
Is `4*i` available at this point?

Formulating the Problem

- **Domain:**
  - a bit vector, with
    a bit for each **“textually unique”** expression in the program
- **Forward or Backward?** Forward
- **Lattice Elements?** All bit vectors of given length
- **Meet Operator?** Elementwise-min
  - check: commutative, idempotent, associative
- **Partial Ordering**
- **Top?** (1,1,…,1)
- **Bottom?** (0,0,…,0)
- **Boundary condition: entry/exit node?** out[entry]=(0,…,0)
- **Initialization for iterative algorithm?** Coming soon…



Meet Operator:
Elementwise-min



Meet Operator:
Intersection

Transfer Functions

- Expression E is available at point P iff along every path to P in the flow graph:
  - E must be evaluated at least once
  - no variables in E redefined after the last evaluation

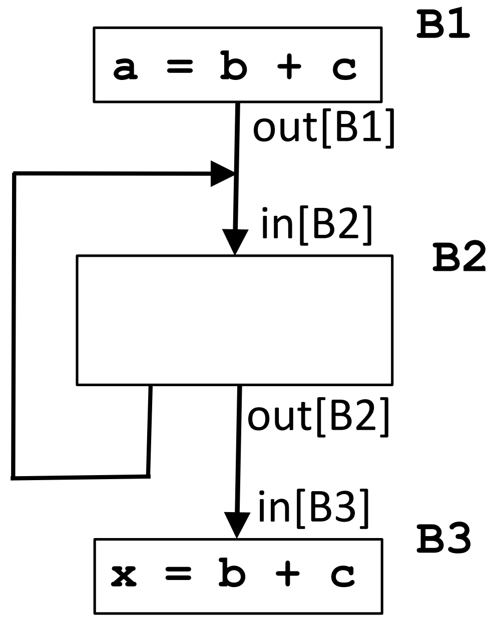
- Can use the same equation as reaching definitions
  - $out[b] = gen[b] \cup (in[b] - kill[b])$

- Start with the transfer function for a single instruction: `x = y + z`
  - When does the instruction kill an expression E? It defines a variable in E.
  - When does it generate an expression E? It evaluates E and doesn't kill it.

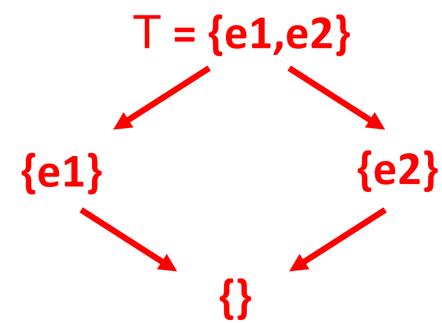
- Calculate transfer functions for complete basic blocks by composing individual instruction transfer functions

Statement	Available Expressions
	{}
`a = b + c`	
	{b+c}
`b = a - d`	
	{a-d}
`c = b + c`	
	{a-d}
`d = a - d`	
	{}

Initialization for Interior Nodes



out[b] = Gen[b] U (in(b)-Kill[b])



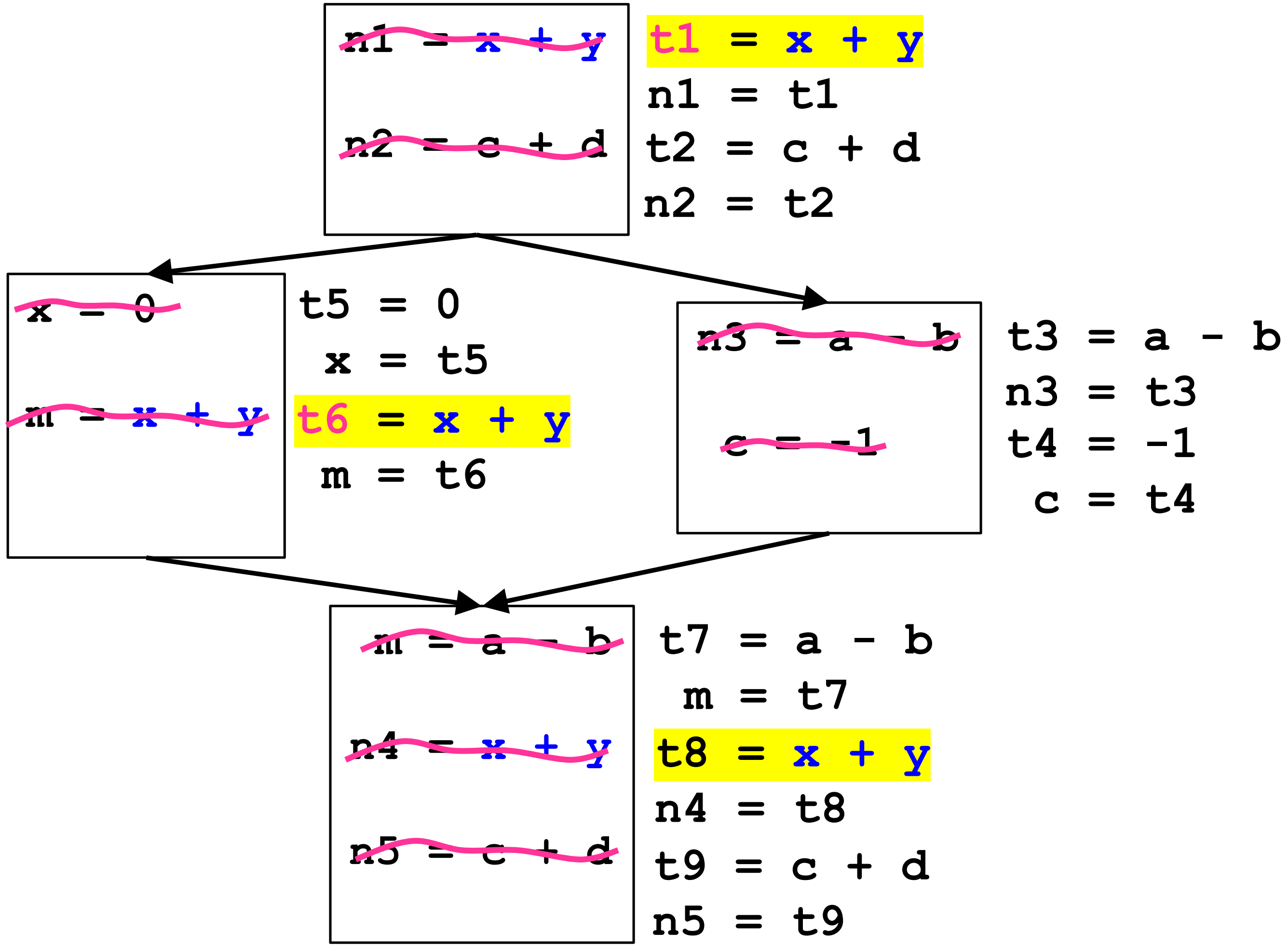
Meet Operator:
Intersection

- What if initialize out[B2] = {}? Imprecise: in[B2]=out[B1] $\wedge$ out[B2] = {}
  Thus, in[B3]={} each iteration, so conclude "b+c" is NOT available in B3.
- What if initialize out[B2] = T? Precise: in[B2]=out[B1]
  Thus, in[B3]={"b+c"}, so conclude "b+c" is available in B3.
- **Initialize out[b]= T for all interior b**

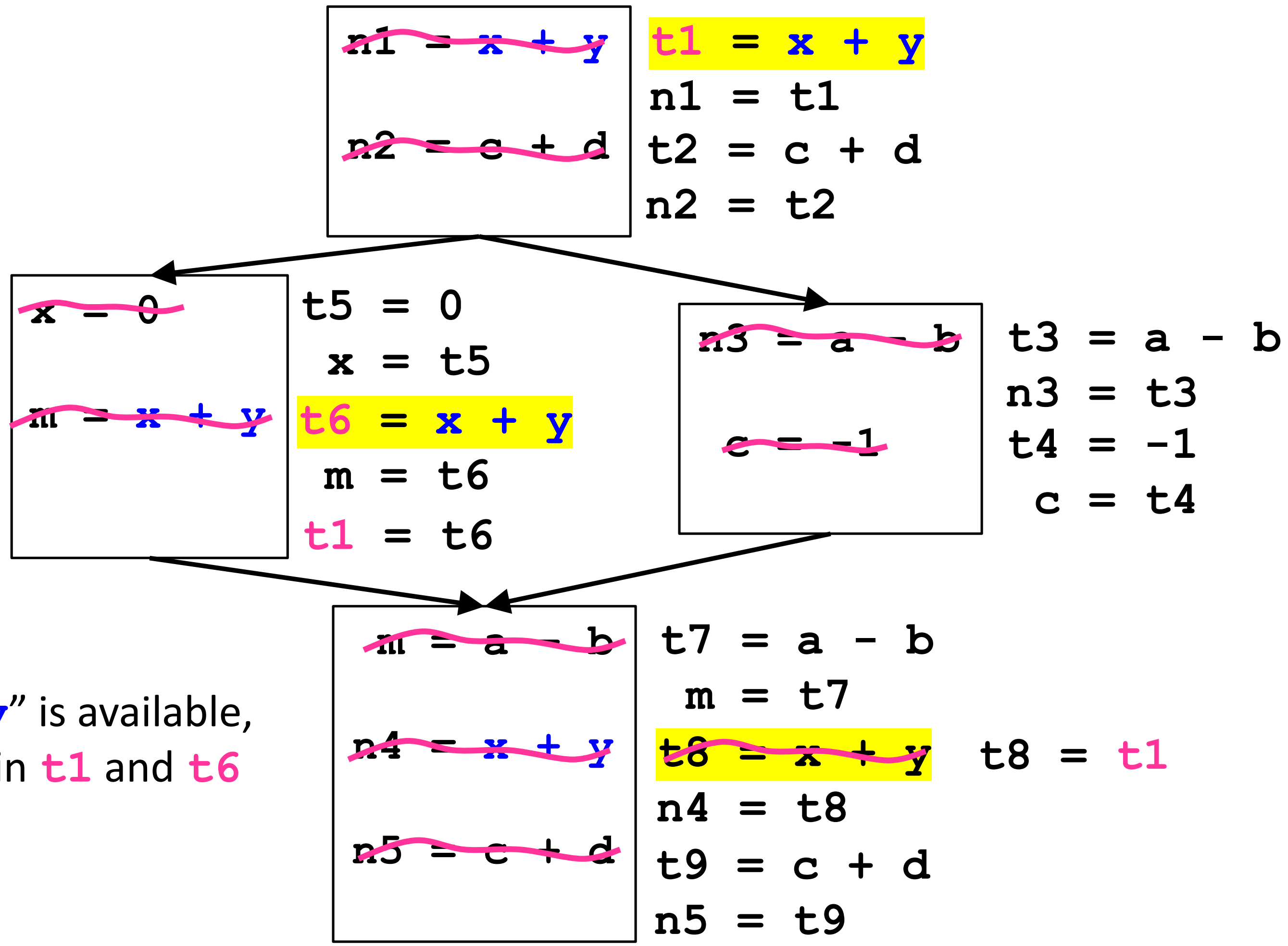
II. Eliminating CSEs

- **Value Numbering (within basic block)**
  - Eliminates local common subexpressions
- **Available expressions (across basic blocks)**
  - Provides the set of expressions available at the start of a block
- **If CSE is an “available expression”, then transform the code**
  - Original destination may be:
    - a temporary register
    - overwritten
    - different from the variables on other paths
  - One solution: Copy the expression to a new variable at each evaluation reaching the redundant use

Example Revisited: Value Numbering Only



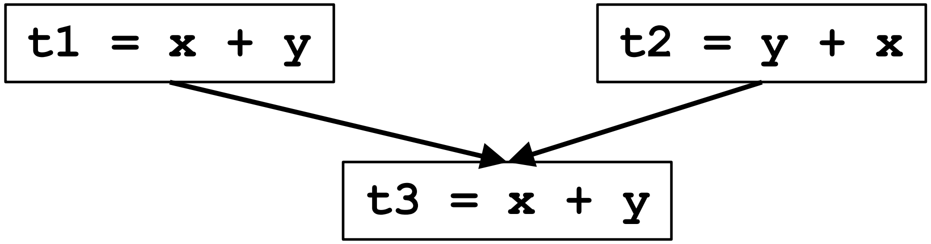
Example Revisited: Eliminating the CSE



"**x+y**" is available, but in t1 and t6

Limitation: Textually Identical Expressions

- **Commutative operations**



  - Won't detect **`x + y`** as an available expression
  - Solution: Sort the operands

Further Improvements

- **Examples**
  - Expressions with more than two operands

```
t1 = x + y
t2 = t1 + z
```

```
t3 = y + x
t4 = t3 + z
```

```
t5 = x + y
t6 = t5 + z
```

  - Textually different expressions may be equivalent

```
t1 = x + y
if t1 > y goto L1
z = x
```

~~t2 = z + y~~

After copy propagation:

```
t2 = x + y
```

Solution: Use multiple passes of GCSE combined with copy propagation

Carnegie Mellon

Summary

	Reaching Definitions	**Available Expressions**
Domain	Sets of definitions	Sets of expressions
Direction	forward: $out[b] = f_b(in[b])$ $in[b] = \wedge\ out[pred(b)]$	forward: $out[b] = f_b(in[b])$ $in[b] = \wedge\ out[pred(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet Operation ($\wedge$)	$\cup$	$\cap$
Boundary Condition	$out[entry] = \emptyset$	$out[entry] = \emptyset$
Initial interior points	$out[b] = T = \emptyset$	$out[b] = T =$ all expressions

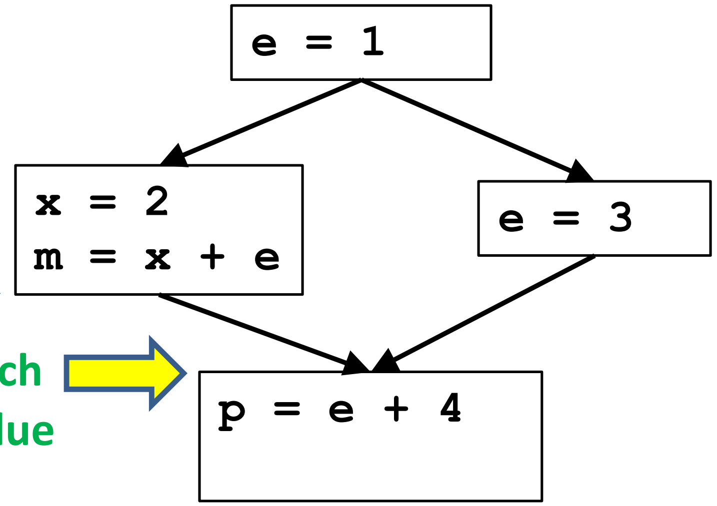
Available Expressions

$Kill_b$ = all E such that block b defines a variable in E

Gen_b = all E such that block b evaluates E and doesn't later kill it

III. Constant Propagation/Folding

- **At every basic block boundary, for each variable v**
  - determine if v is a constant
  - if so, what is the value?

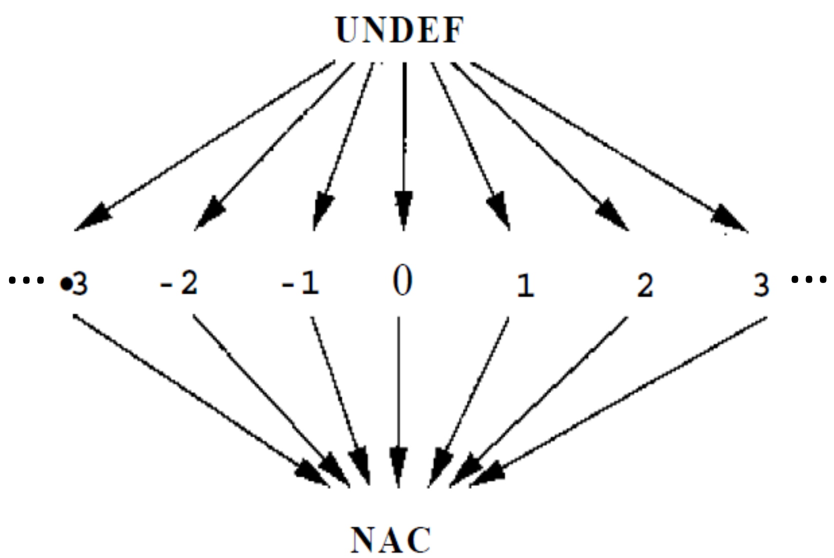


`e`, `x`, `m` are each a constant value

`x`, `m` are each a constant value (but not `e`)

Which variables are constants?

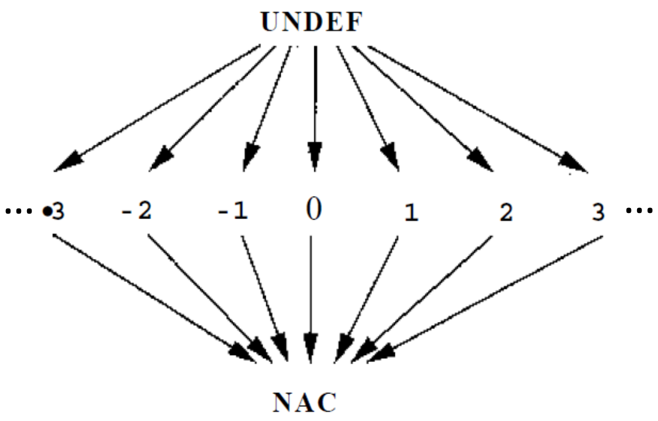
Semi-lattice Diagram



- Finite domain? No (unless bound number of bits)
- Finite height? Yes (2)
- One such lattice for each variable in the program

Meet Operation in Table Form

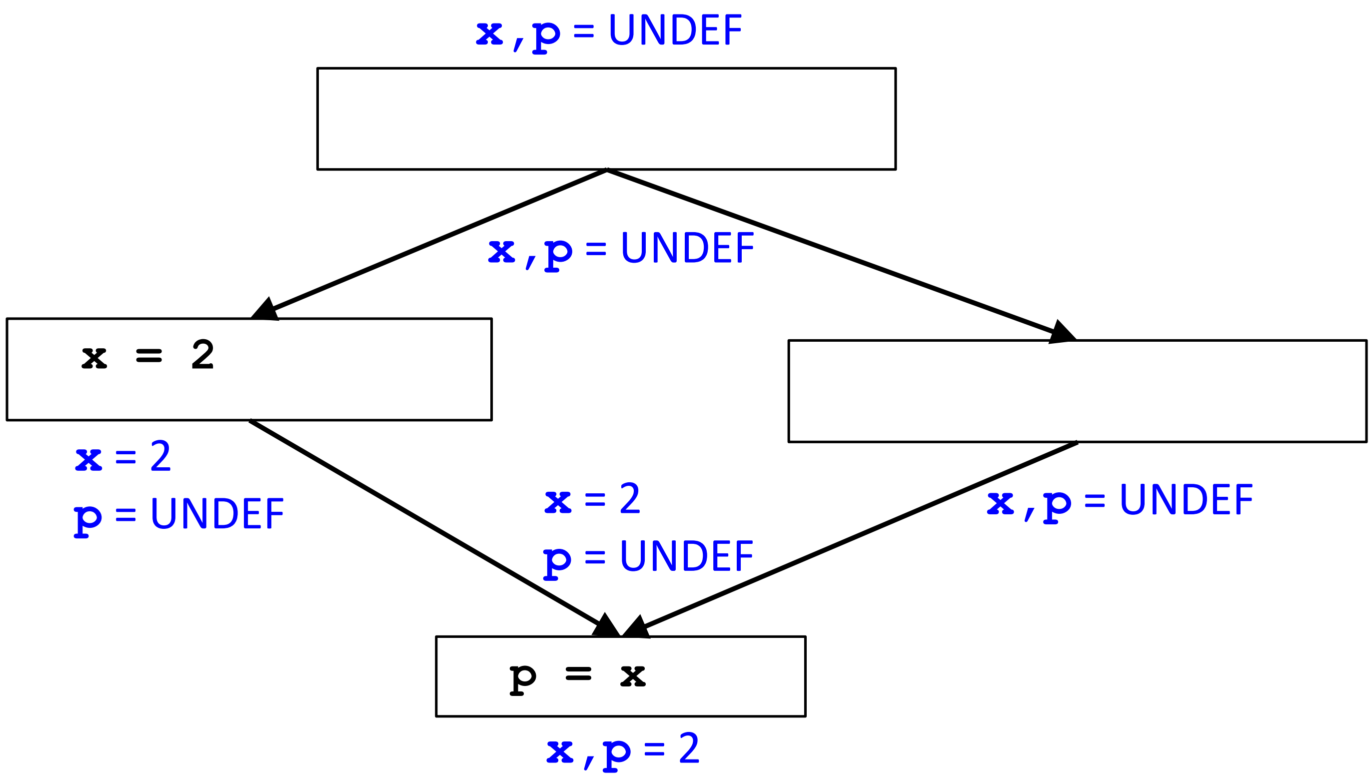
- **Meet Operation:**



v1	v2	v1 $\wedge$ v2
UNDEF	UNDEF	UNDEF
	c_2	c_2
	NAC	NAC
c_1	UNDEF	c_1
	c_2	c_1, if $c_1 = c_2$ NAC otherwise
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

  – Note: UNDEF $\wedge$ c_2 = c_2

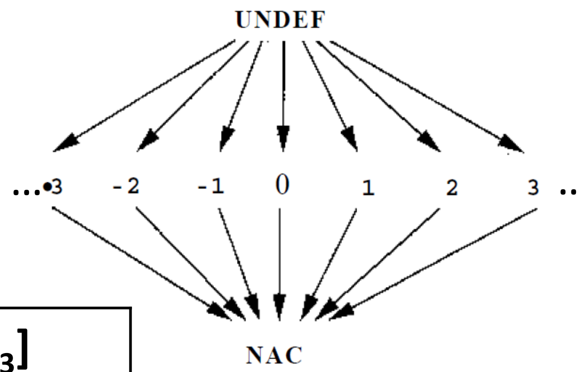
Example



Transfer Function

- **Assume a basic block has only 1 instruction**
- **Let IN[b,x], OUT[b,x]**
  - be the information for variable x at entry and exit of basic block b

- **OUT[entry, x] = UNDEF, for all x.**
- **Non-assignment instructions: OUT[b,x] = IN[b,x]**
- **Assignment instructions: (next page)**



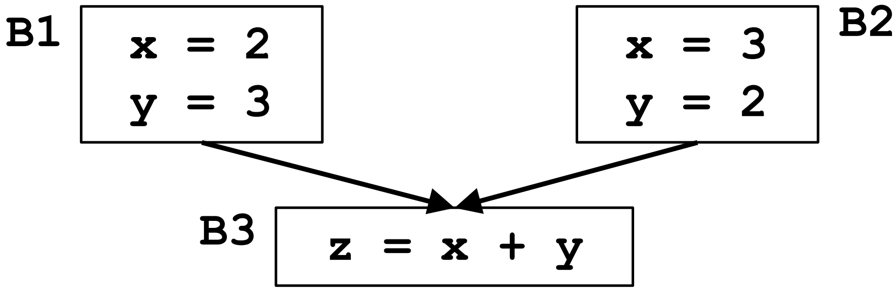
Transfer Function (cont.)

- **Let an assignment be of the form $x_3 = x_1 + x_2$**
  - "+" represents a generic operator
  - OUT[b,x] = IN [b,x], if $x \neq x_3$

$IN[b,x_1]$	$IN[b,x_2]$	$OUT[b,x_3]$
UNDEF	UNDEF	**UNDEF**
	c_2	**UNDEF**
	NAC	**NAC**
c_1	UNDEF	**UNDEF**
	c_2	$\mathbf{c_1 + c_2}$
	NAC	**NAC**
NAC	UNDEF	**NAC**
	c_2	**NAC**
	NAC	**NAC**

- **Use:** $x \leq y$ implies $f(x) \leq f(y)$ to check if framework is monotone
  - $[v_1\ v_2\ ...] \leq [v_1'\ v_2'\ ...]$, $f([v_1\ v_2\ ...]) \leq f([v_1'\ v_2'\ ...])$

Not Distributive



	x	y	z
$f_1(\top)$	2	3	UNDEF
$f_2(\top)$	3	2	UNDEF
$f_1(\top) \wedge f_2(\top)$	NAC	NAC	UNDEF
$f_3(f_1(\top) \wedge f_2(\top))$	NAC	NAC	NAC
$f_3(f_1(\top))$	2	3	5
$f_3(f_2(\top))$	3	2	5
$f_3(f_1(\top)) \wedge f_3(f_2(\top))$	NAC	NAC	5

- Not Distributive: $f_3(f_1(\top) \wedge f_2(\top)) < f_3(f_1(\top)) \wedge f_3(f_2(\top))$
- Iterative solution is not precise. It is not wrong. It is conservative.

Summary of Constant Propagation

- **A useful optimization**
- **Illustrates:**
  - abstract execution
  - an infinite semi-lattice
  - a non-distributive problem

Today's Class

I. Available Expressions Analysis

II. Eliminating CSEs

III. Constant Propagation/Folding

Friday's Class

- Induction Variable Optimizations
  - ALSU 9.1.8, 9.6, 9.8.1