Dataflow Analysis, cont.

Announcements

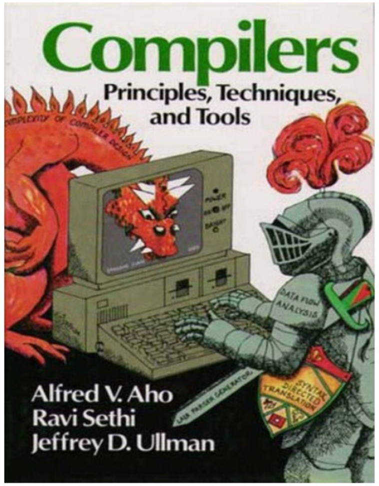
- Monday is Martin Luther King Jr. Day, No classes

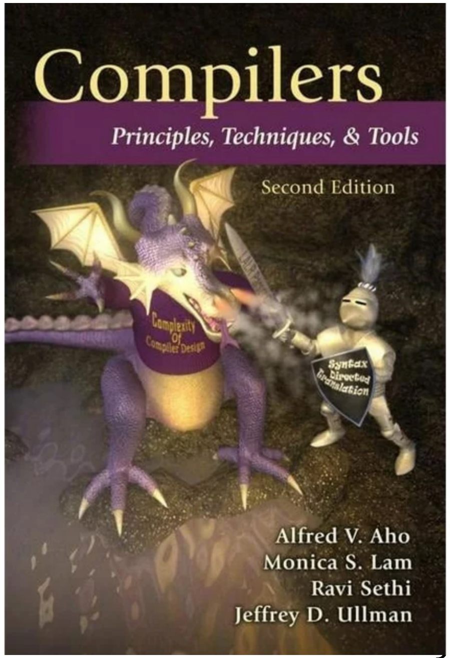
- HW1 problem set is posted, due Jan 25th
  - Work individually or in teams of 2
  - Ask questions on forum
  - Upload in Submitty

Outline of Today's Class

- Classical compiler optimizations
- Building CFG from 3-address code
- Local analysis vs. global analysis
- The four classical dataflow analysis problems
  - Reaching definitions
  - Live variables
  - Available expressions
  - Very busy expressions

- Reading:
  - Dragon Book, Chapter 9.2

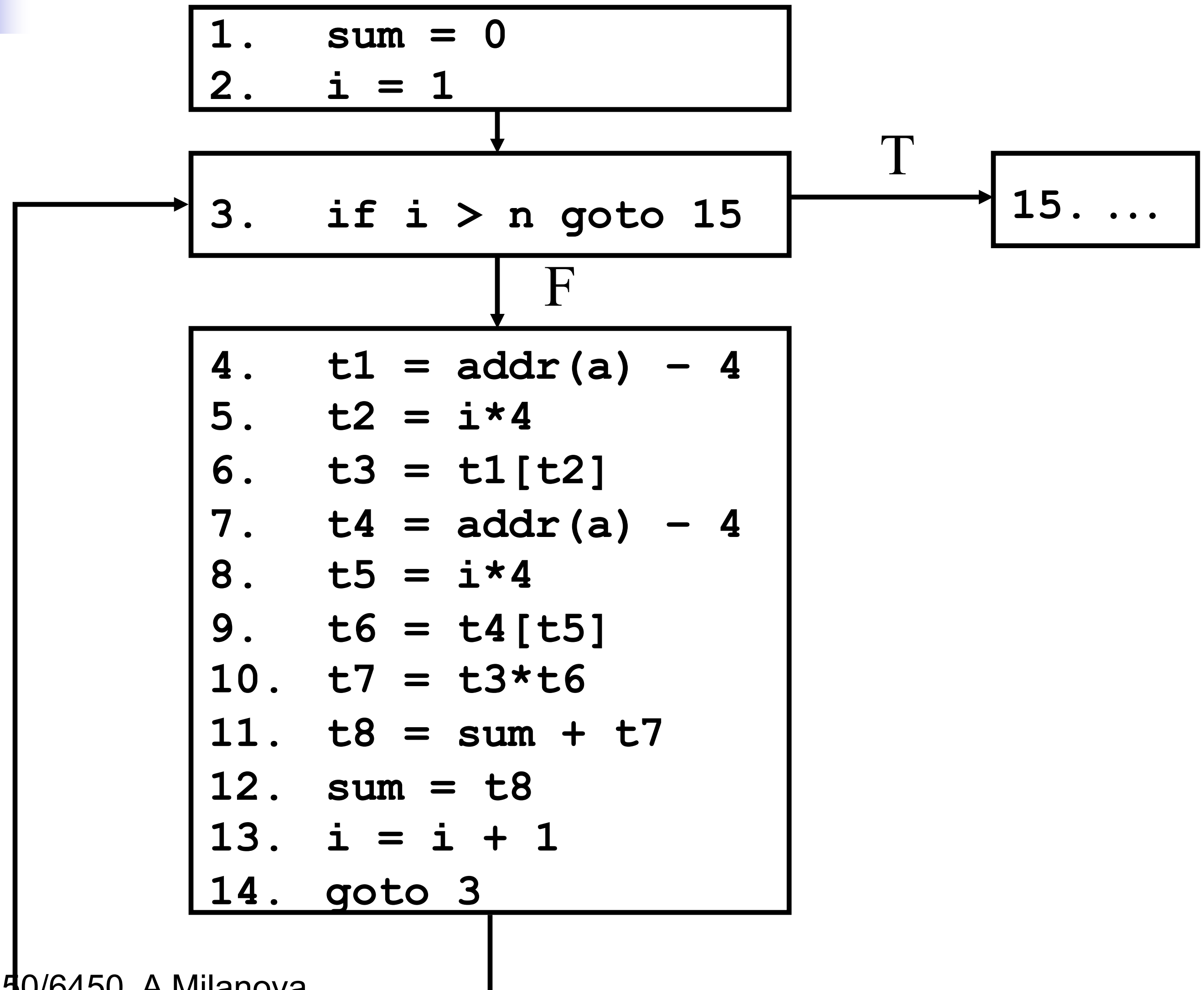




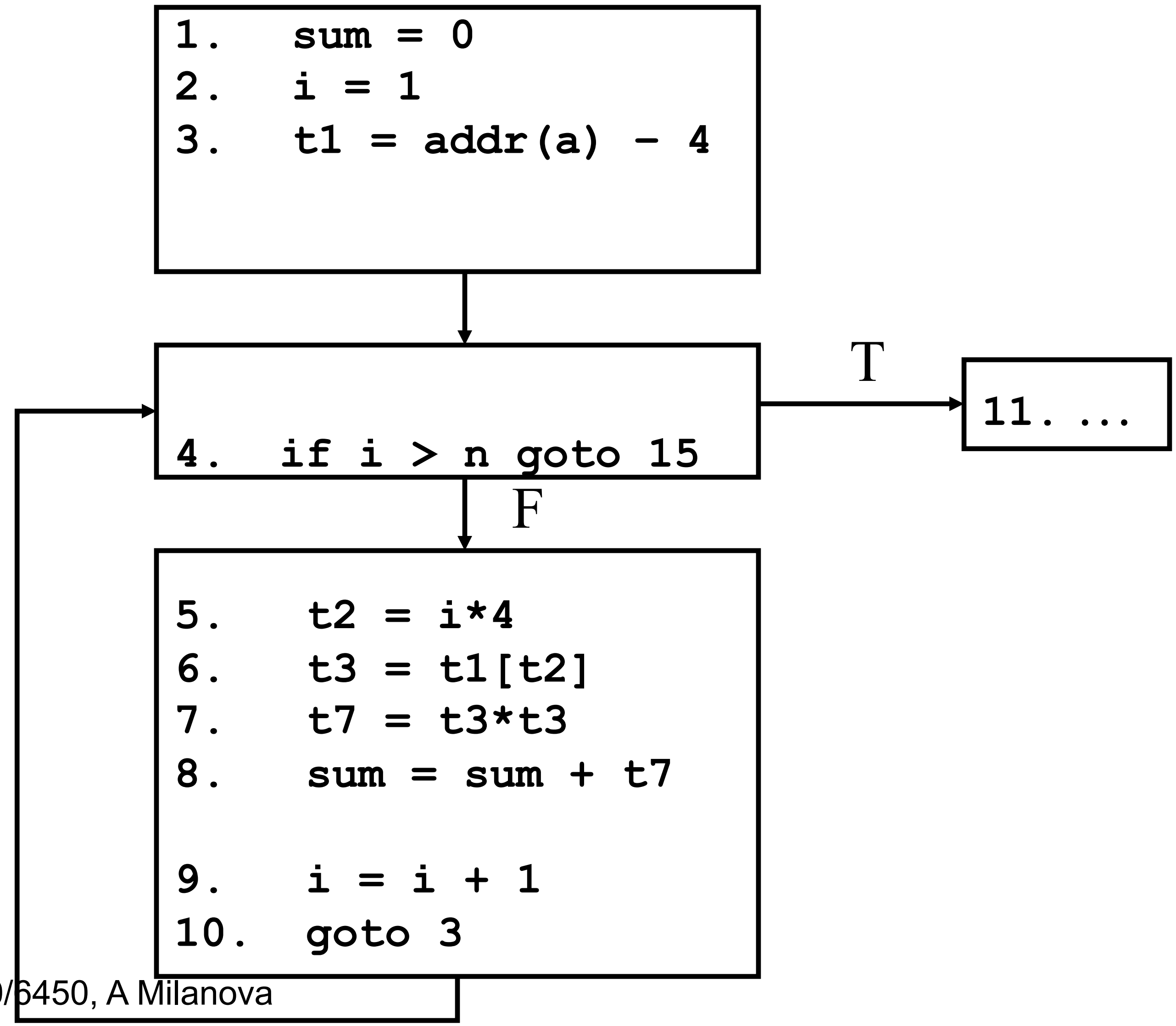
Three Address Code Intermediate Representation (IR)

1.	`sum = 0`	initialize sum
2.	`i = 1`	initialize loop counter
3.	`if i > n goto 15`	loop test, check for limit
4.	`t1 = addr(a) – 4`	
5.	`t2 = i * 4`	a[i]
6.	`t3 = t1[t2]`	
7.	`t4 = addr(a) – 4`	
8.	`t5 = i * 4`	a[i]
9.	`t6 = t4[t5]`	
10.	`t7 = t3 * t6`	a[i]*a[i]
11.	`t8 = sum + t7`	
12.	`sum = t8`	increment sum
13.	`i = i + 1`	increment loop counter
14.	`goto 3`	
15.	`…`	

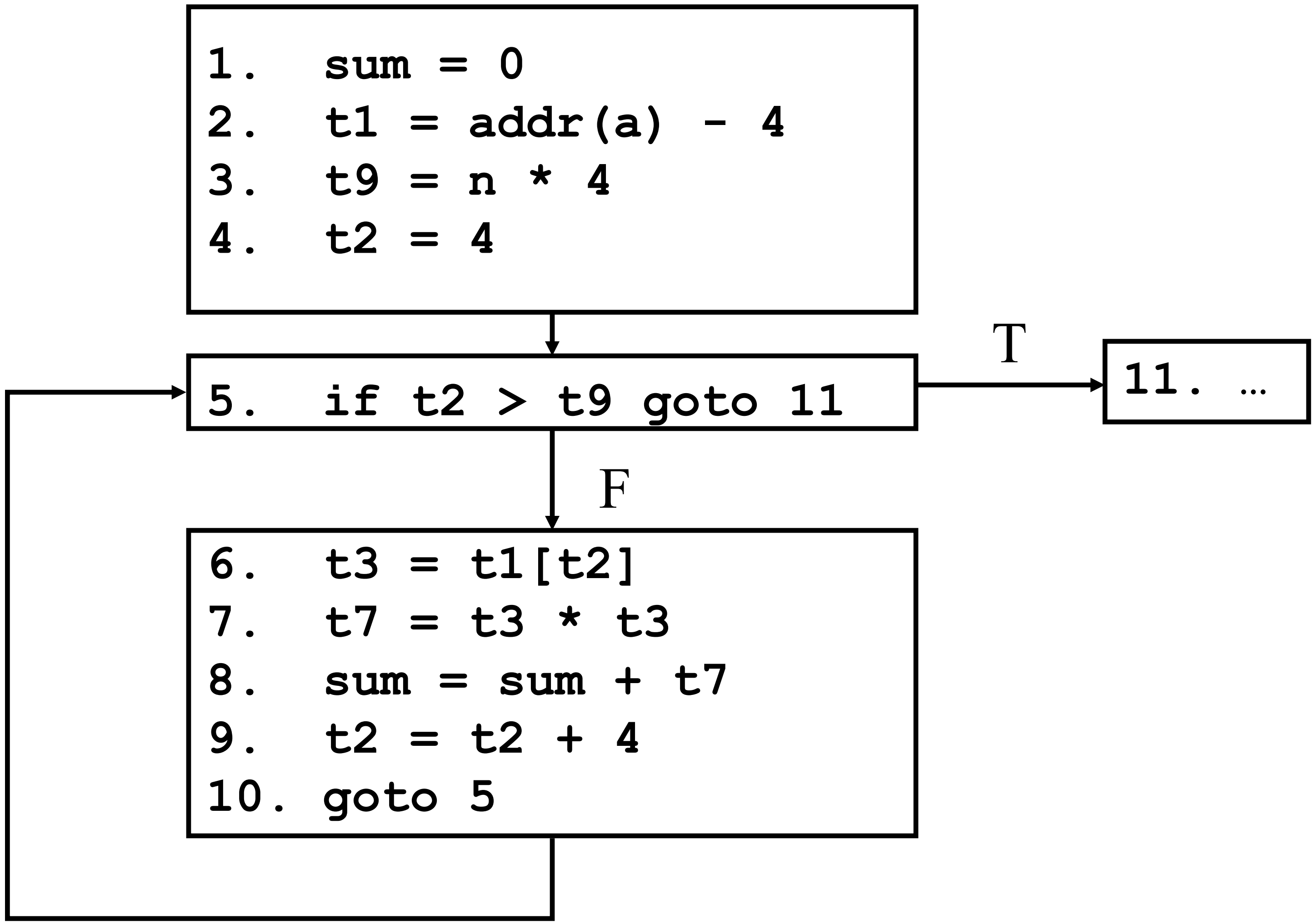
Control Flow Graph (CFG)



Control Flow Graph (CFG)



New Control Flow Graph

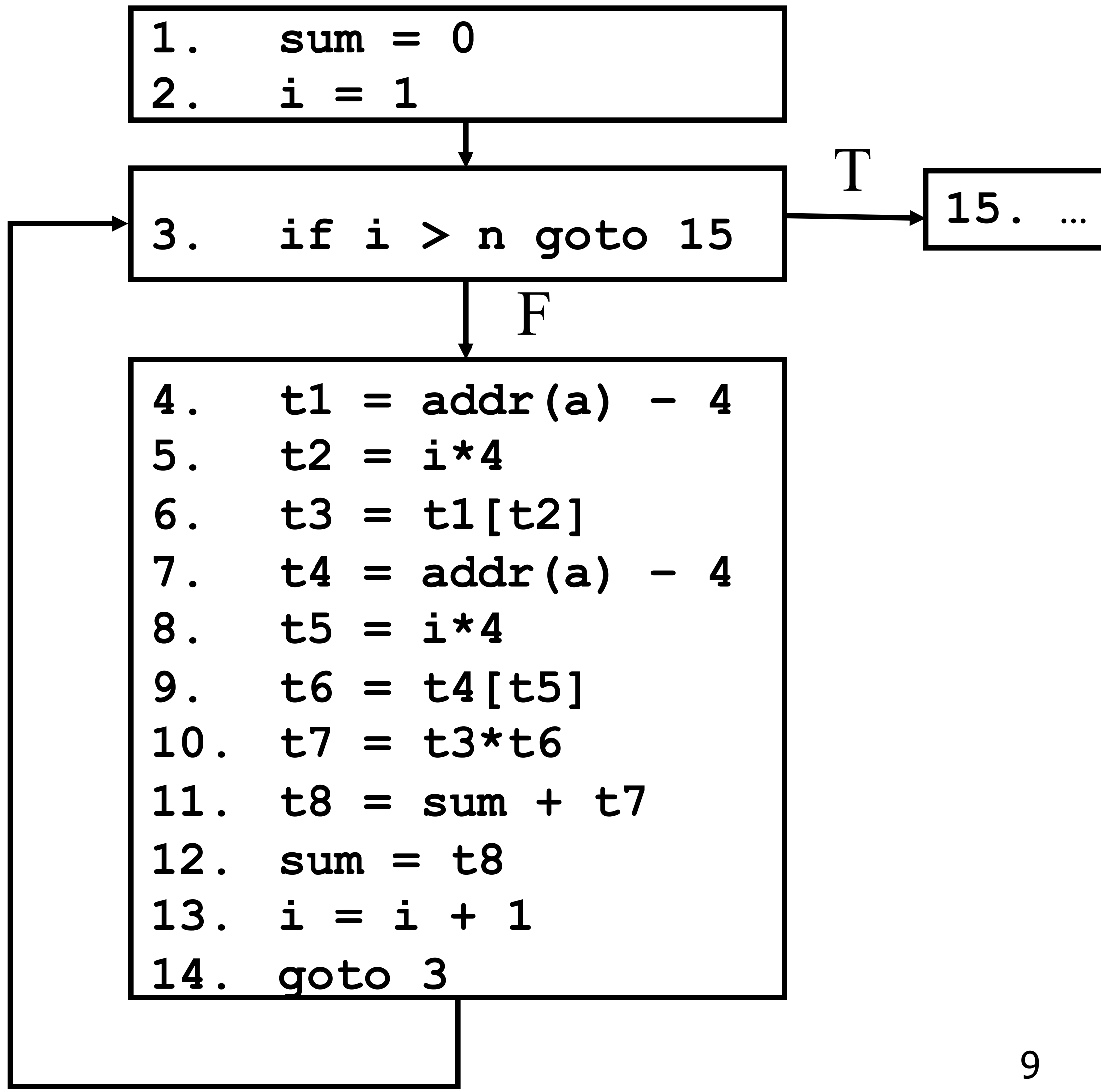


Classical Compiler Optimizations

- To summarize
  - Common subexpression elimination
  - Copy propagation
  - Strength reduction
  - Test elision and induction variable elimination
  - Constant propagation
  - Dead code elimination
- Dataflow analysis <u>enables</u> these optimizations

Building Control Flow Graph

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i*4
6.   t3 = t1[t2]
7.   t4 = addr(a) - 4
8.   t5 = i*4
9.   t6 = t5[t5]
10.  t7 = t3*t6
11.  t8 = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
15.  …
```



Building the Control Flow Graph

Build the CFG from linear 3-address code:

- Step 1: partition code into basic blocks
  - Basic blocks are the nodes of the CFG
- Step 2: add control flow edges

- Aside: in Principles of Software, we built a CFG from "high-level" structural program representation, the AST:
  - *S* ::= **x = y** *Op* **z** | **if (***B***) then** *S* **else** *S* |
    while (*B***)** *S* | *S*;*S*

Step 1. Partition Code Into Basic Blocks

1. Determine the leader statements:

    (i) First program statement

    (ii) Targets of a goto, conditional or unconditional

    (iii) Any statement following a goto

2. For each leader, its basic block consists of the leader and all statements up to, but not including, the next leader or the end of the program

Question. Find the Leader Statements

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i*4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i*4
9.  t6 = t5[t5]
10. t7 = t3*t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. …
```

Step 2. Add Control Flow Edges

- There is a directed edge from basic block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence
- Determine edges as follows:
  (i) There is an edge from B_1 to B_2 if B_2 follows B_1 in three address code, and B_1 does not end in an unconditional goto
  (ii) There is an edge from B_1 to B_2 if there is a goto from the last statement in B_1 to the first statement in B_2

Question. Add Control Flow Edges

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i*4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i*4
9.  t6 = t5[t5]
10. t7 = t3*t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. ...
```

Local Analysis vs. Global Analysis

- Local analysis: analysis within **basic block**
  - Enables optimizations such as local common subexpression elimination, dead code elimination, constant propagation, copy propagation, etc.
- Global analysis: beyond the basic block
  - Enables optimizations such as global common subexpression elimination, dead code elimination, constant propagation, loop optimizations, etc.

Local Common Subexpression Elimination

```
1.  t1 = 4 * i
2.  t2 = a [ t1 ]
3.  t3 = 4 * i
4.  t4 = b [ t3 ]
5.  t5 = t2 * t4
6.  t6 = prod + t5
7.  prod = t6
8.  t7 = i + 1
9.  i = t7
10. if i <= 20 goto 1
```

Local Constant Propagation

Assume a, k, t3, and t4 are used beyond basic block:

```
1.  t1 = 1
2.  a = t1
3.  t2 = 1 + a
4.  k = t2
5.  t3 = cvttoreal(k)
6.  t4 = 6.2 + t3
7.  t3 = t4
```

```
1'. a = 1
2'. k = 2
3'. t4 = 8.2
4'. t3 = 8.2
```

David Gries' algorithm:

- Process 3-address statements in order
- Check if operand is constant; if so, substitute
- If all operands are constant:
  Do operation, and add (LHS,value) to map
- If not all operands constant:
  Delete (LHS,value) entry from map

Arrays and Pointers Make Things Harder

- Consider:

```
1. x = a[k];
2. a[j] = y;
3. z = a[k];
```

- Can we transform this code into:

```
1. x = a[k];
2. a[j] = y;
3. z = x;
```

Local Analysis vs. Global Analysis

- Local analysis is generally easy – a single path from basic block entry to basic block exit

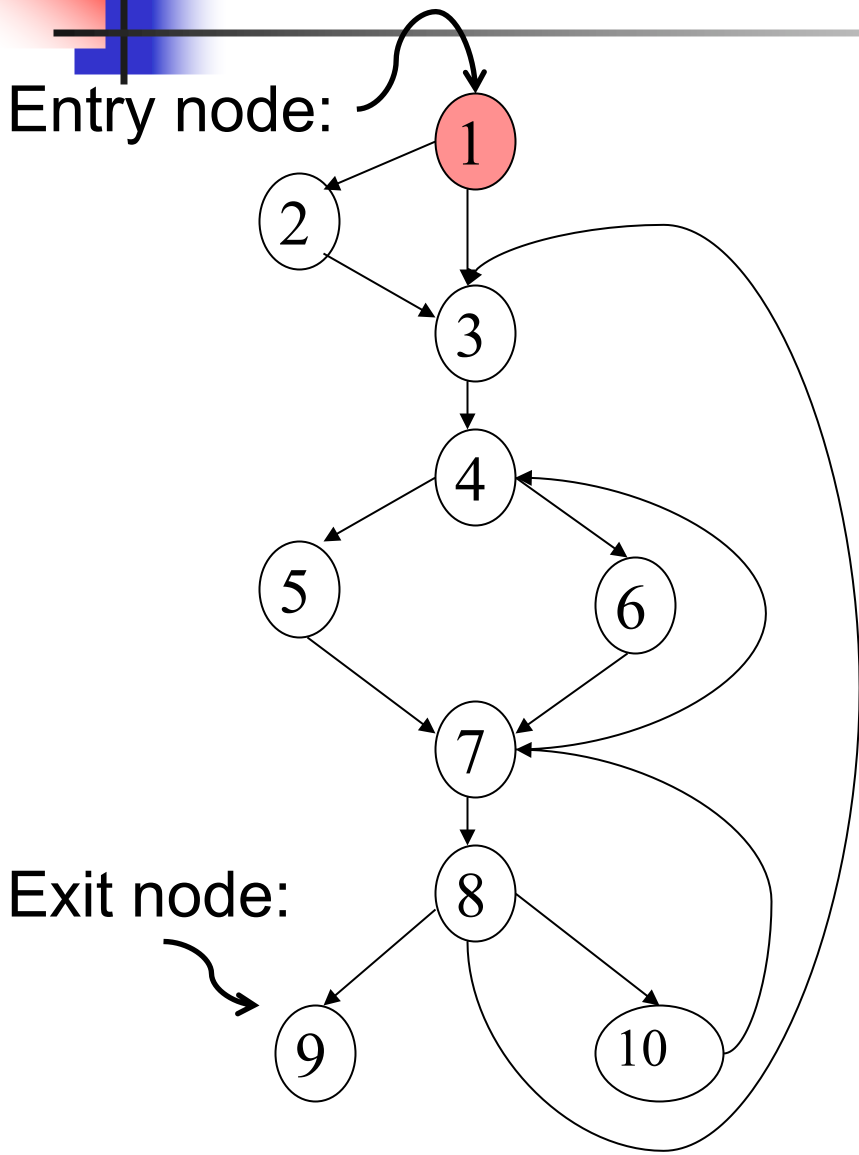
- Global analysis is generally hard – multiple control-flow paths
  - Control flow splits and merges at if-then-else
  - Loops!

Dataflow Analysis

- Collects information for all inputs along all execution paths
  - Control splits and control merges
  - Loops (control goes back)

- Dataflow analysis is a powerful framework
- We can define many different dataflow analysis

Dataflow Analysis



1. Control-flow graph (CFG):
   - G = (N, E, 1)
   - Nodes are basic blocks
2. Data
3. Dataflow equations

out(j) = (in(j) – kill(j)) U gen(j)

(gen and kill are parameters)

4. Merge operator V

in(j) = V out(i)

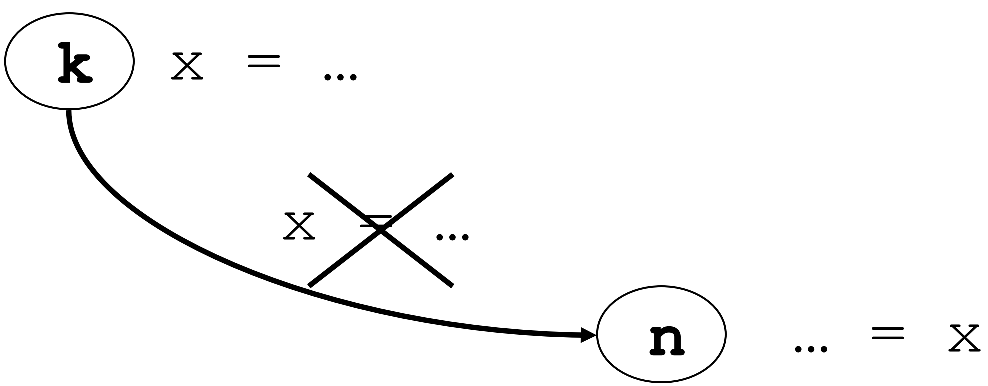
i is predecessor of j

Four Classical Dataflow Problems

- Reaching definitions (*Reach*)
- Live uses of variables (*Live*)
- Available expressions (*Avail*)
- Very busy expressions (*VeryB*)
- *Reach* and the dual *Live* enable several classical optimizations such as dead code elimination, as well as dataflow-based testing
- *Avail* enables global common subexpression elimination
- *VeryB* enables conservative code motion

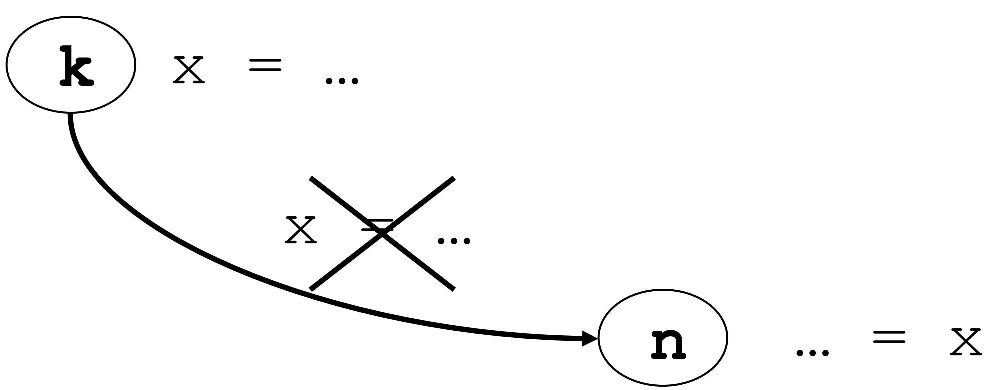
Reaching Definitions

- **Definition** A statement that may change the value of a variable (e.g., **`x=y+z`**)
- **`(x,k)`** denotes definition of **`x`** at node **`k`**
- A definition **`(x,k)`** **reaches** node **`n`** if there is a path from **`k`** to **`n`**, free of a definition of **`x`**



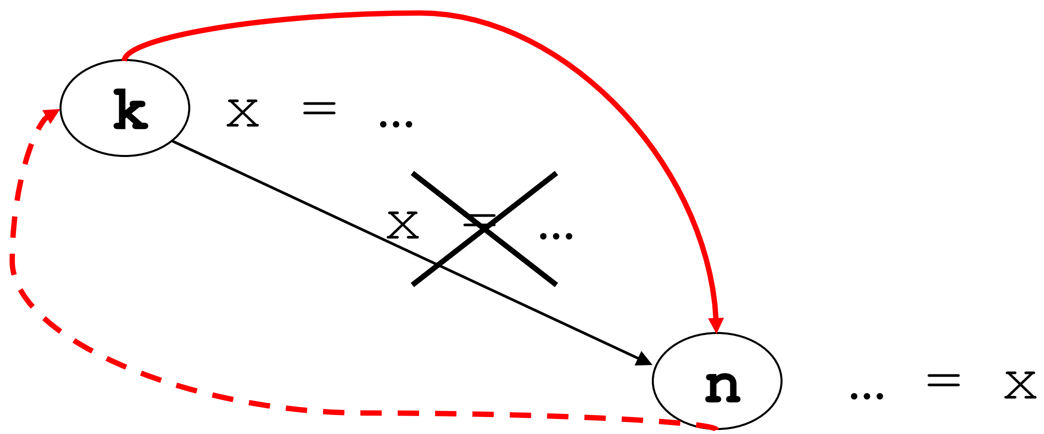
Live Uses of Variables

- **Use** Appearance of a variable as an operand of a 3-address statement (e.g., `x` in `y=x+4`)
- A use of a variable `x` at node `n` is **live on exit** from `k`, if there is a path from `k` to `n` clear of definition of `x`



Def-use Relations

- **Use-def chain** links a use of **x** to a definition of **x** that reaches that use
- **Def-use chain** links a definition to a use that it reaches

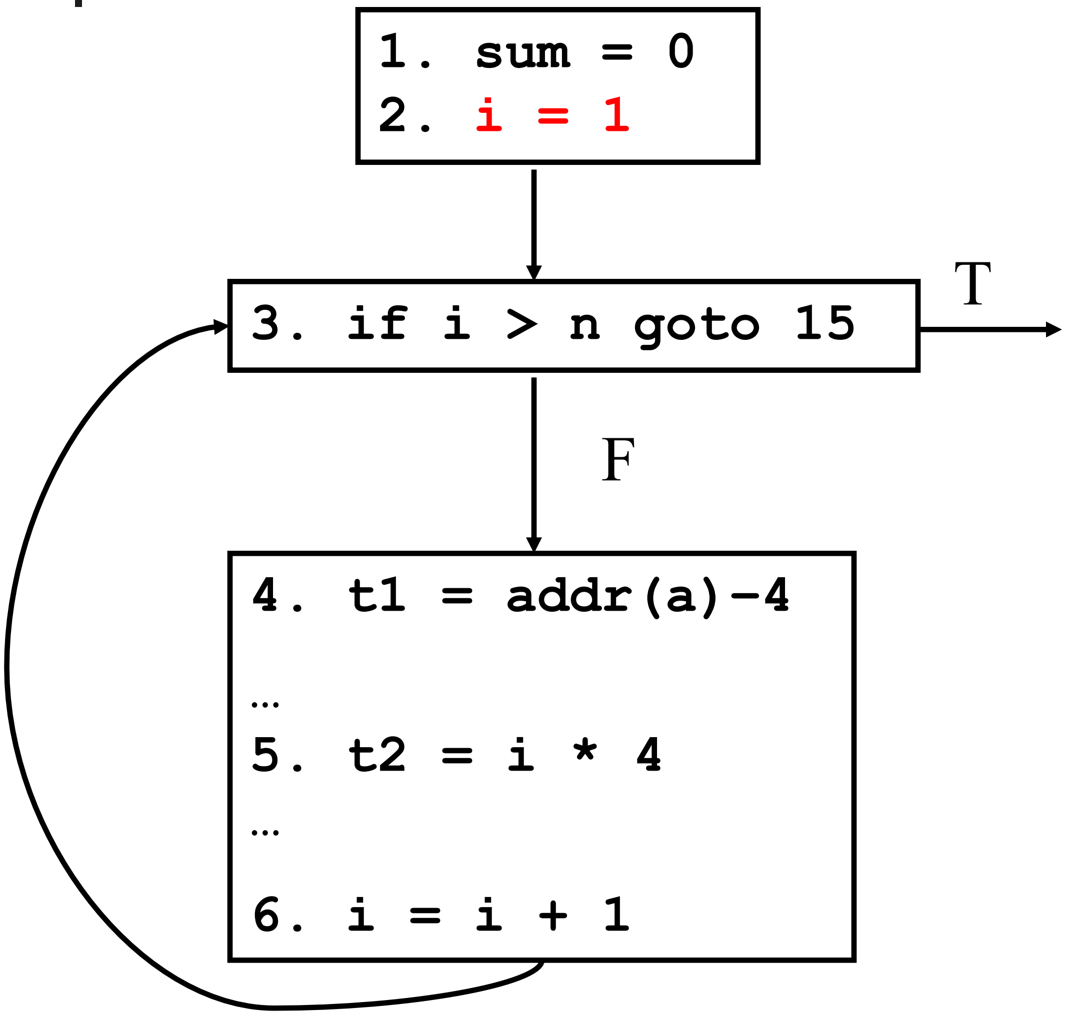


Def-use Enable Optimizations

- Dead code elimination (Def-use)
- Code motion (Use-def)
- Constant propagation (Use-def)
- Strength reduction (Use-def)
- Test elision (Use-def)
- Copy propagation (Def-use)

- Aside: Def-use enables dataflow-based testing. (In Principles of Software)

Question. What are the Def-use Chains that start at 2?



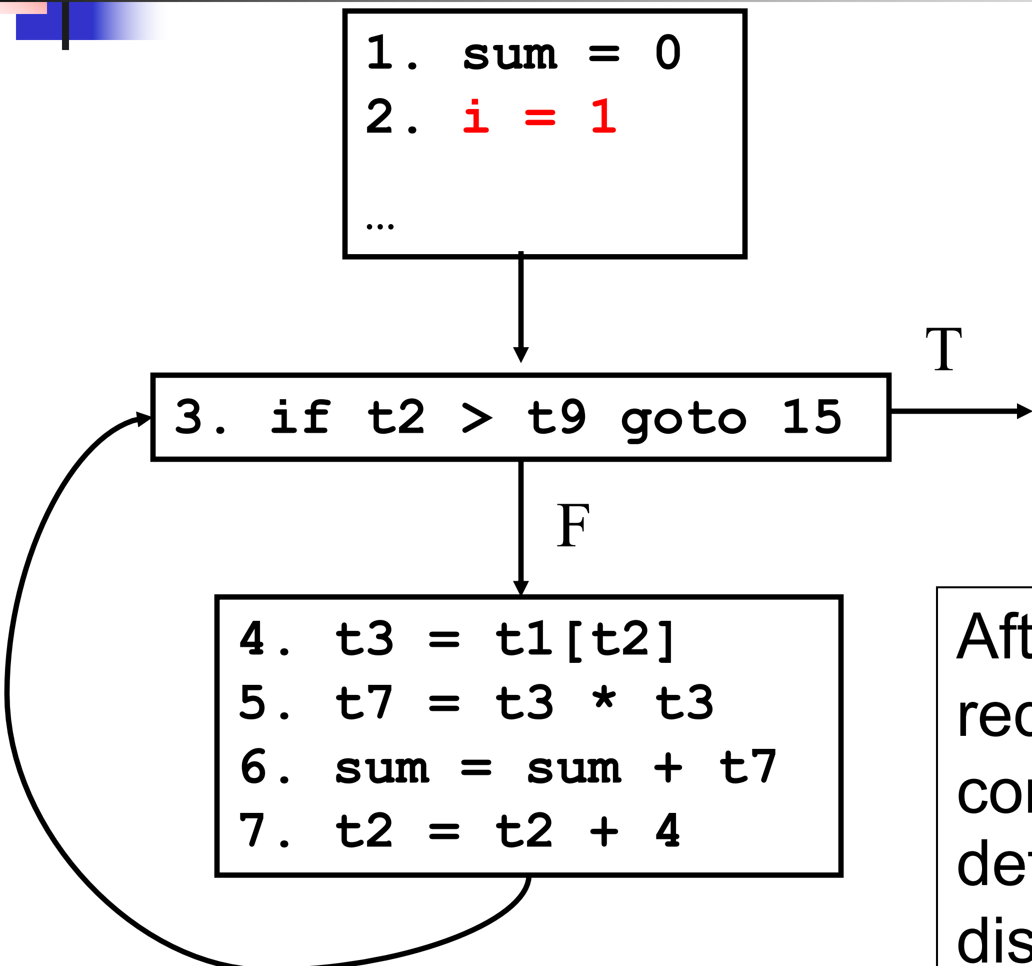
Answer:

(2,3)

(2,5)

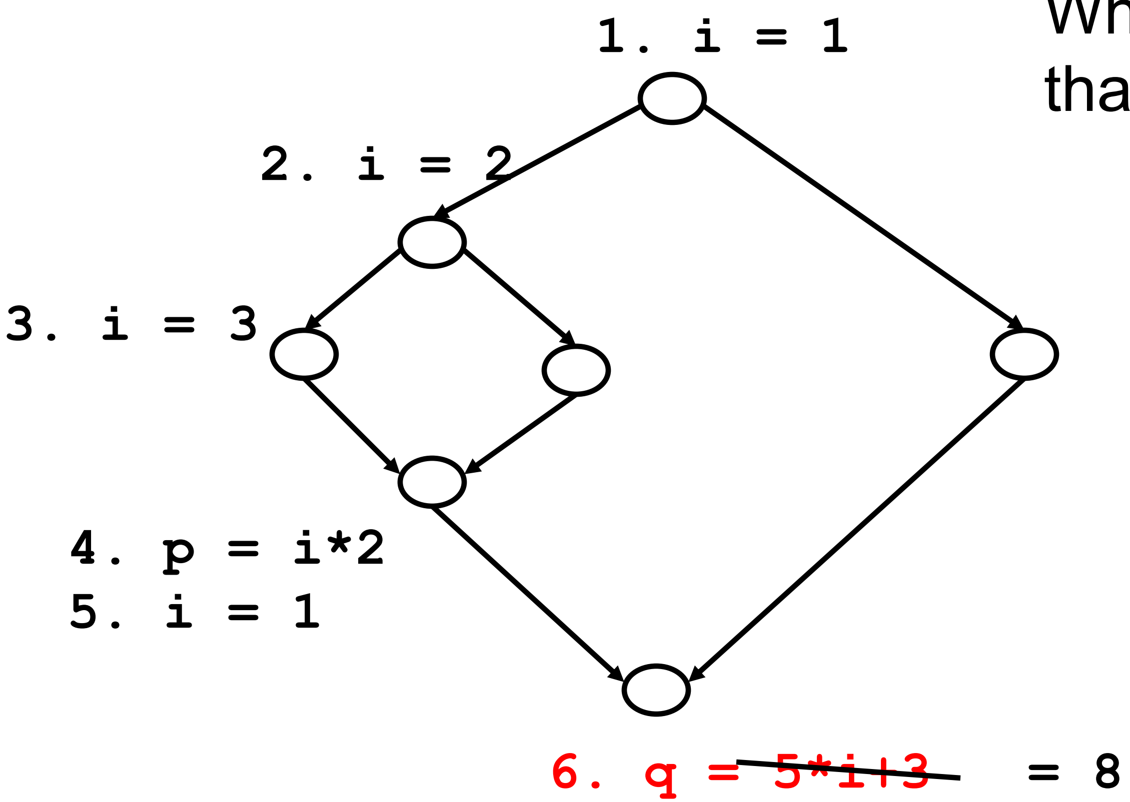
(2,6)

Def-use Enables Dead Code Elimination



After code motion, strength reduction, test elision and constant propagation, the def-use links from `2.i=1` disappear. Thus, `2.i=1` becomes dead code.

Use-def Enables Constant Propagation



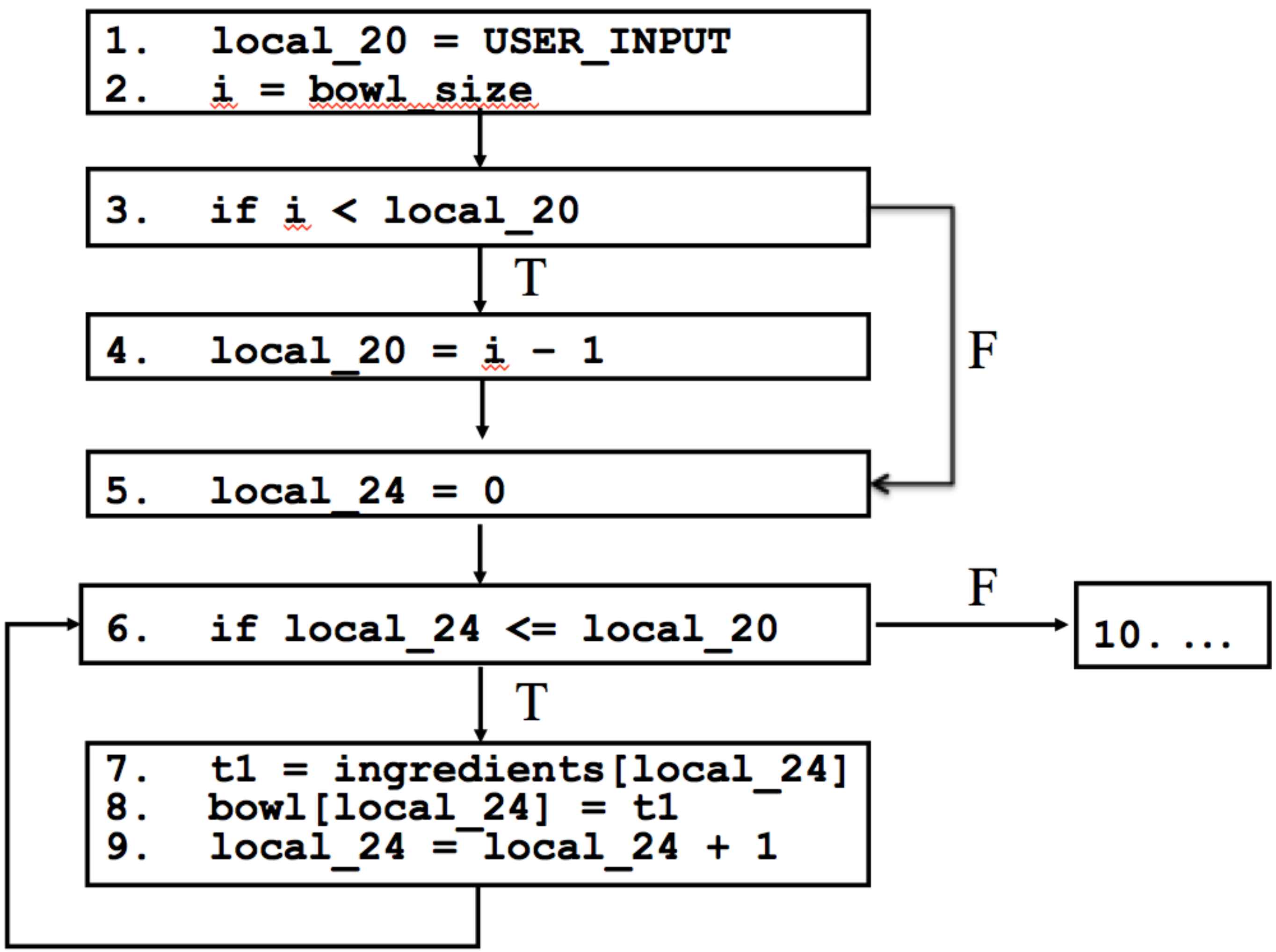
What are the use-def chains that originate at 6?

Answer:

(6,1)

(6,5)

Def-use Enables Reasoning about Buffer Overflows



Problem 1. Reaching Definitions (*Reach*)

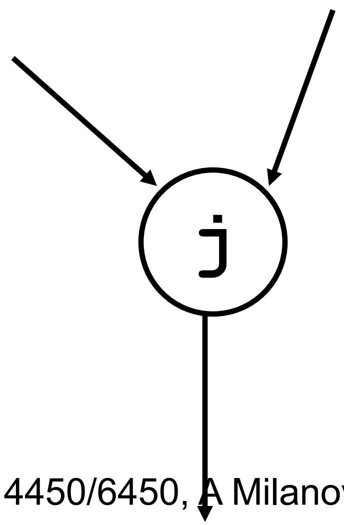
- Problem statement: for each CFG node **`n`**, compute the set of definitions **`(x,k)`** that reach **`n`**

- First, define data (i.e., the dataflow facts) to propagate
  - Primitive dataflow facts are definitions **`(x,k)`**
  - *Reach* propagates sets of definitions, e.g., **`{(i,1),(p,4)}`**

Reaching Definitions (*Reach*)

- Next, define the dataflow equations (i.e., effect of code at node `j` on incoming dataflow facts)

`j: x = y+z` } kill(j): all definitions of `(x,_)`
gen(j): this definition of `x`, `(x,j)`

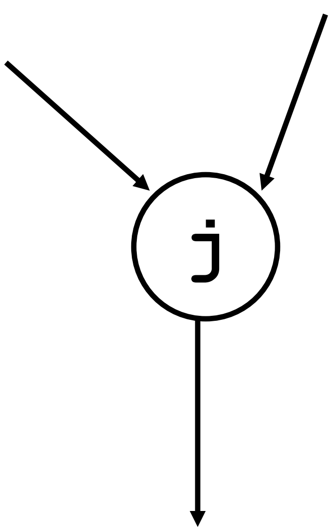


out(j) = (in(j) – kill(j)) U gen(j)

E.g., if in(4) = `{(x,1),(y,2),(x,3)}`
Node 4 is: `x = y+z`
Then out(4) = `{(y,2),(x,4)}`

Reaching Definitions (*Reach*)

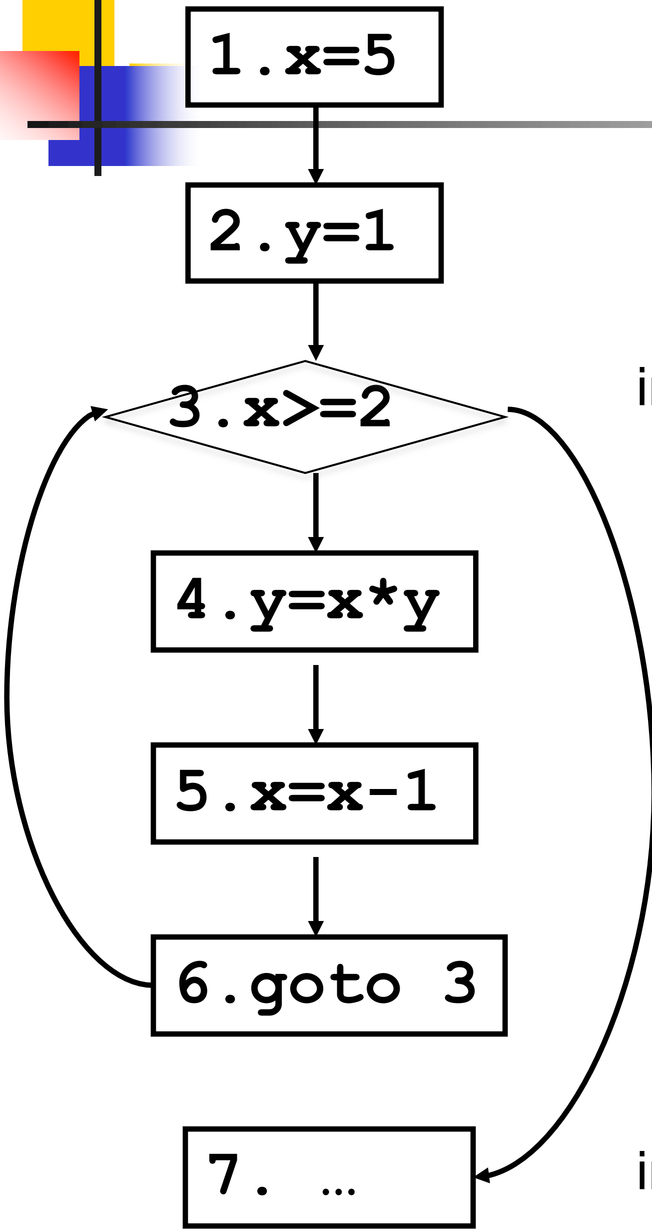
- Next, define the merge operator V (i.e., how to combine data from incoming paths)
- For *Reach*, V is the set union U



in(j) = { U out(i) | i is predecessor of j }

E.g., if out(2) = **{(x,1),(y,2)}** and
out(3) = **{(x,3)}** and
2 and 3 are predecessors of 4
in(4) = **{(x,1),(x,3),(y,2)}**

Reach: Dataflow Equations



$in(1) = \emptyset$

$out(1) = (in(1) - D_x) \cup \{(x,1)\}$

$in(2) = out(1)$

$out(2) = (in(2) - D_y) \cup \{(y,2)\}$

$in(3) = out(2) \cup out(6)$

$out(3) = in(3)$

$in(4) = out(3)$

$out(4) = (in(4) - D_y) \cup \{(y,4)\}$

$in(5) = out(4)$

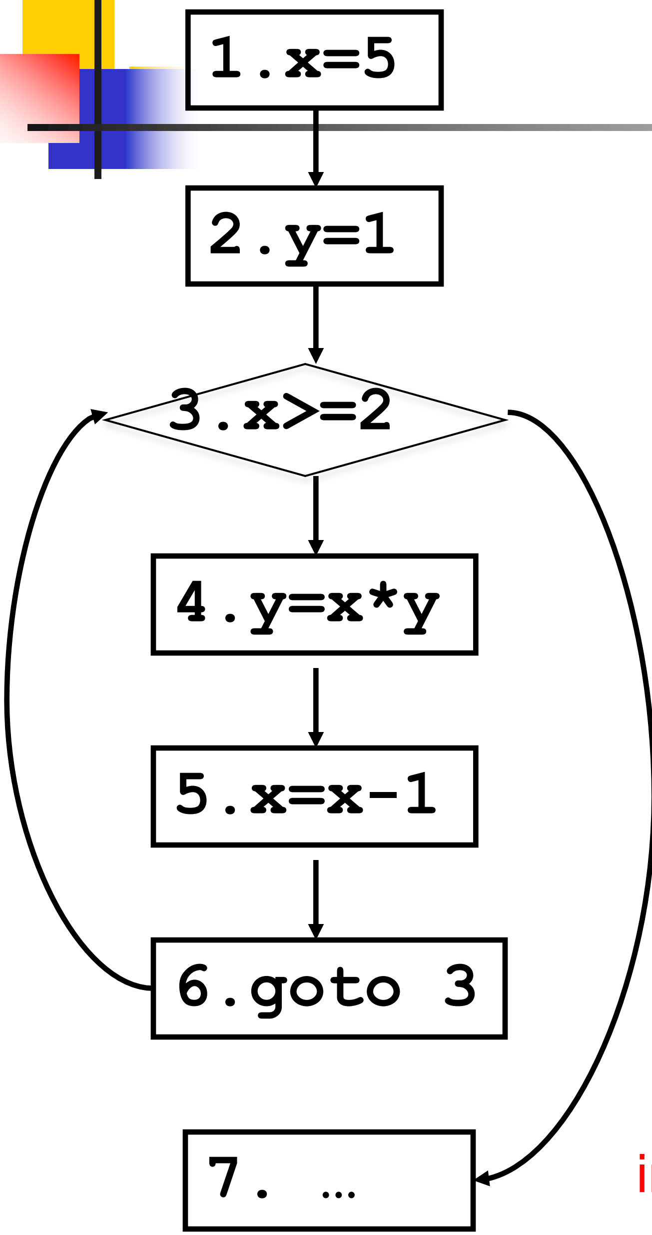
$out(5) = (in(5) - D_x) \cup \{(x,5)\}$

$in(6) = out(5)$

$out(6) = in(6)$

$in(7) = out(3)$

Reach: Solution of Equations



in(1) = Ø

out(1) = {(x,1)}

in(2) = {(x,1)}

out(2) = {(x,1), (y,2)}

in(3) = {(x,1),(x,5),(y,2),(y,4)}

out(3) = {(x,1),(x,5),(y,2),(y,4)}

in(4) = {(x,1),(x,5),(y,2),(y,4)}

out(4) = {(x,1),(x,5),(y,4)}

in(5) = {(x,1),(x,5),(y,4)}

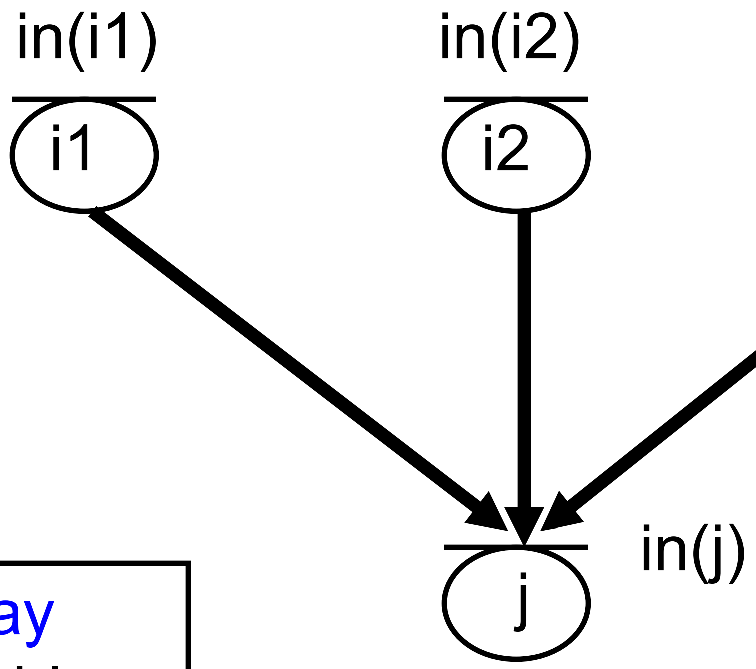
out(5) = {(x,5),(y,4)}

in(6) = {(x,5),(y,4)}

out(6) = {(x,5),(y,4)}

in(7) = {(x,1),(x,5),(y,2),(y,4)}

Reaching Definitions



Forward, may dataflow problem

Problem 2. Live Uses of Variables (*Live*)

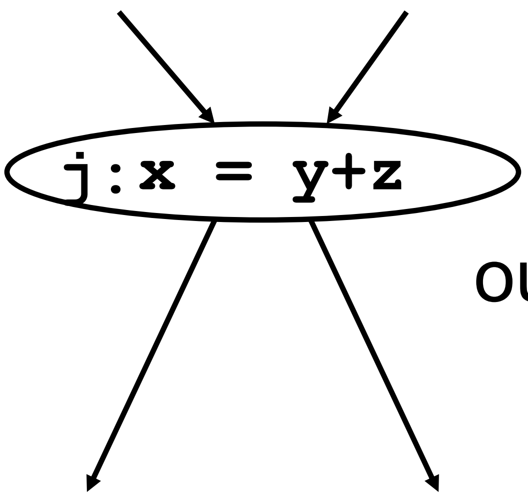
- We say that a variable **`x`** is "live on exit from node **`j`**" if there is a live use of **`x`** on exit from **`j`** (recall the definition of "live use of **`x`** on exit from **`j`**")
- Problem statement: for each node **`n`**, compute the set of variables that are live <u>on exit</u> from **`n`**.

1. x=2; 2. y=4; 3. x=1; if (y>x) then 5. z=y; else 6. z=y*y; 7. x=z;

What variables are live on exit from statement 3? Statement 1?

Live Uses of Variables (*Live*)

- Problem statement: for each node **`n`**, compute the set of variables that are live on exit from **`n`**.

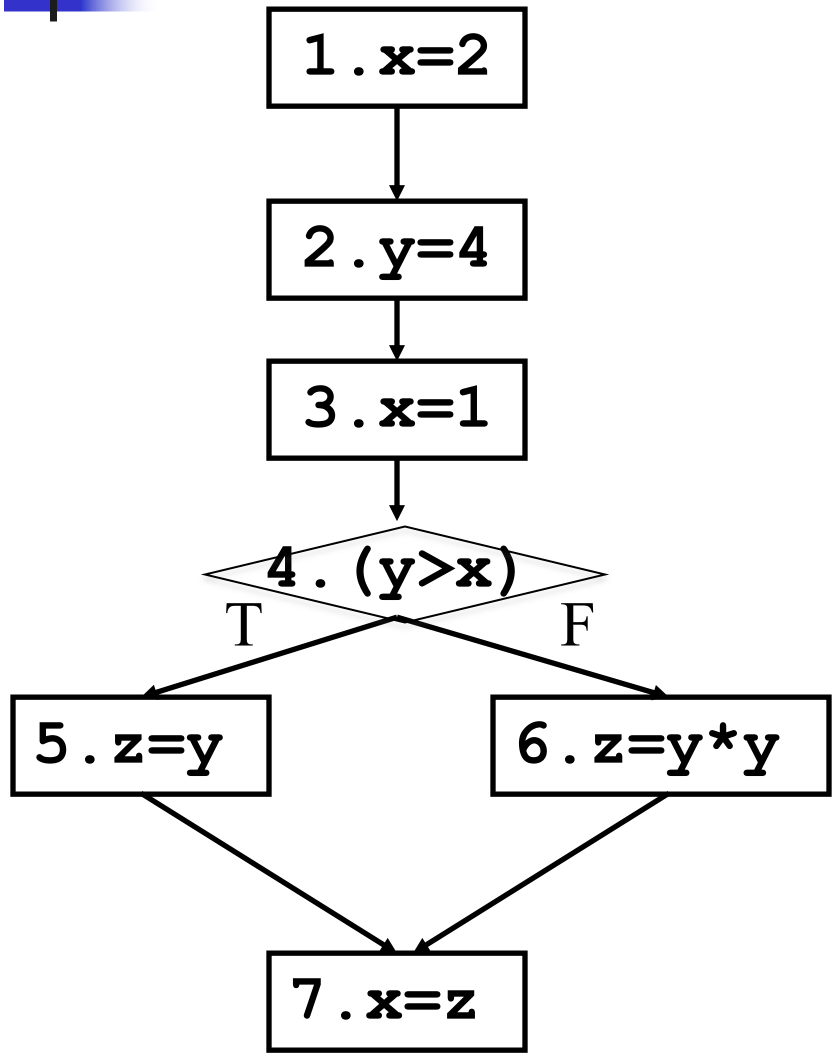


$$in_{LV}(j) = (out_{LV}(j) - kill_{LV}(j)) \cup gen_{LV}(j)$$

$$out_{LV}(j) = \{ \cup\ in_{LV}(i) \mid i \text{ is a successor of } j \}$$

Q: What are the primitive dataflow facts?
Q: What is $gen_{LV}(j)$?
Q: What is $kill_{LV}(j)$?

Live Example



Live Uses of Variables (*Live*)

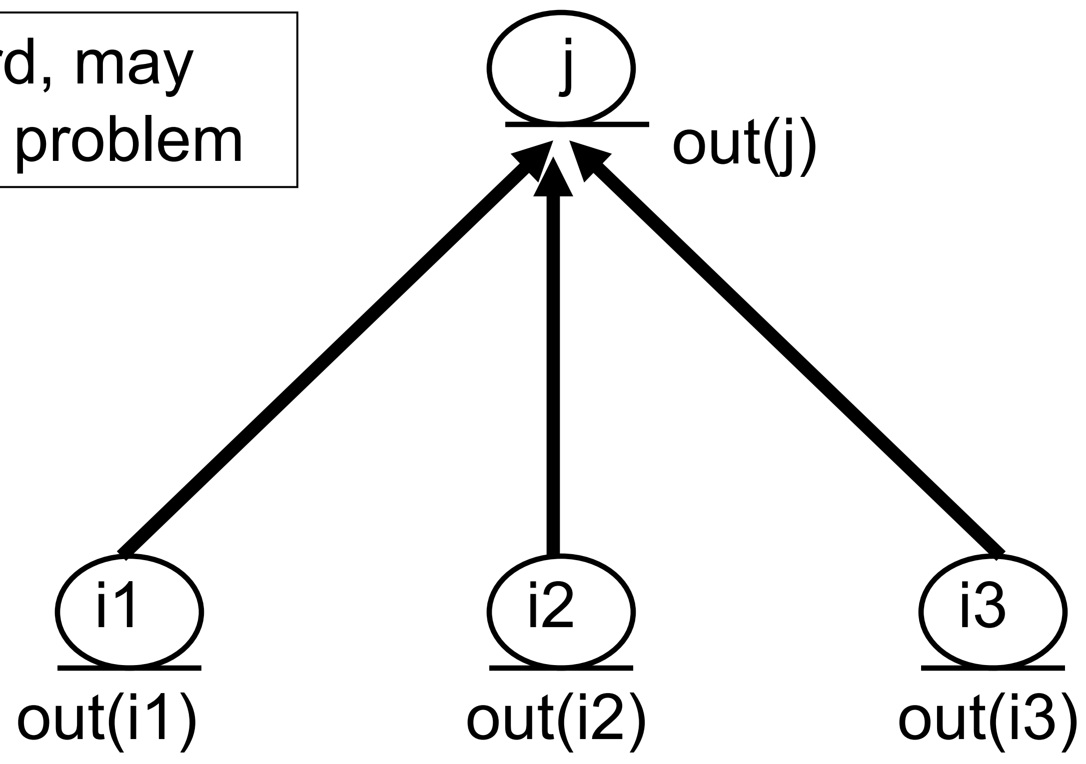
- Data
  - Primitive facts: variables **`x`**
  - Propagates sets: **`{x,y,z}`**

- Dataflow equations. At **`j: x = y+z`**
  - $kill_{LV}(j)$: **`{x}`**
  - $gen_{LV}(j)$: **`{y,z}`**

- Merge operator: set union U

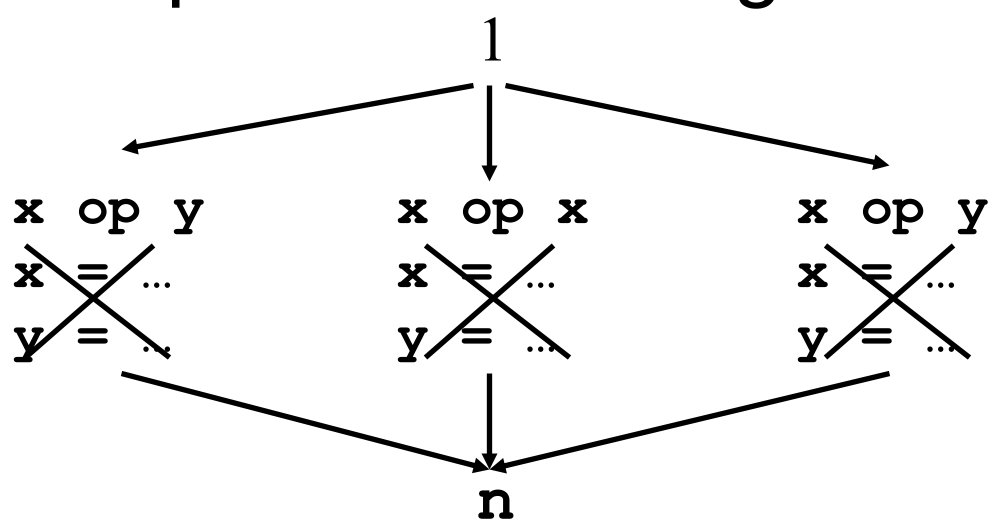
Live Uses of Variables

Backward, may dataflow problem



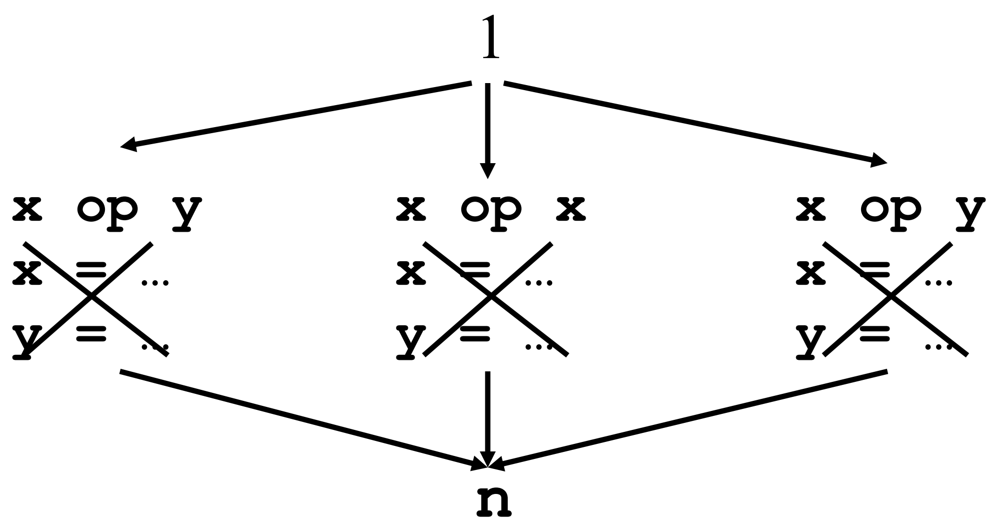
Available Expressions

- An expression `x op y` is available at program point **`n`** if **every** path from entry to **`n`** evaluates `x op y`, and there are NO subsequent assignments to `x` or `y` after evaluation and prior to reaching **`n`**.

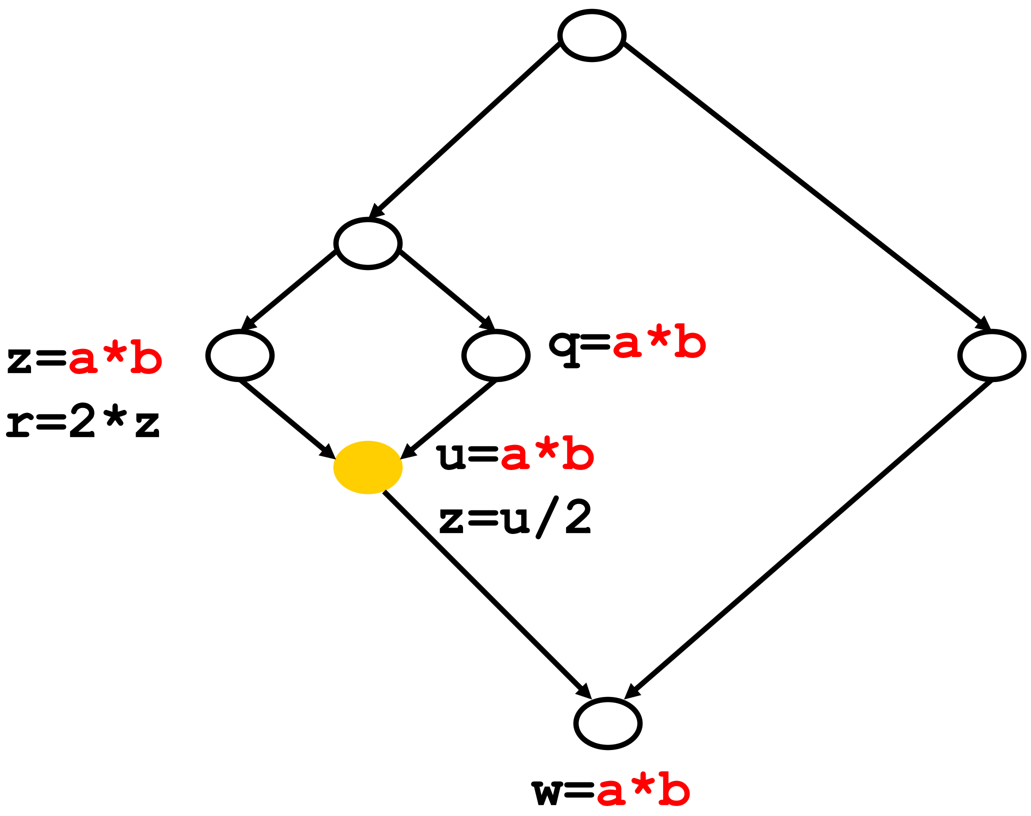


Problem 3. Available Expressions (*Avail*)

- Problem statement: For every node **n**, compute the set of expressions that are available at **n**

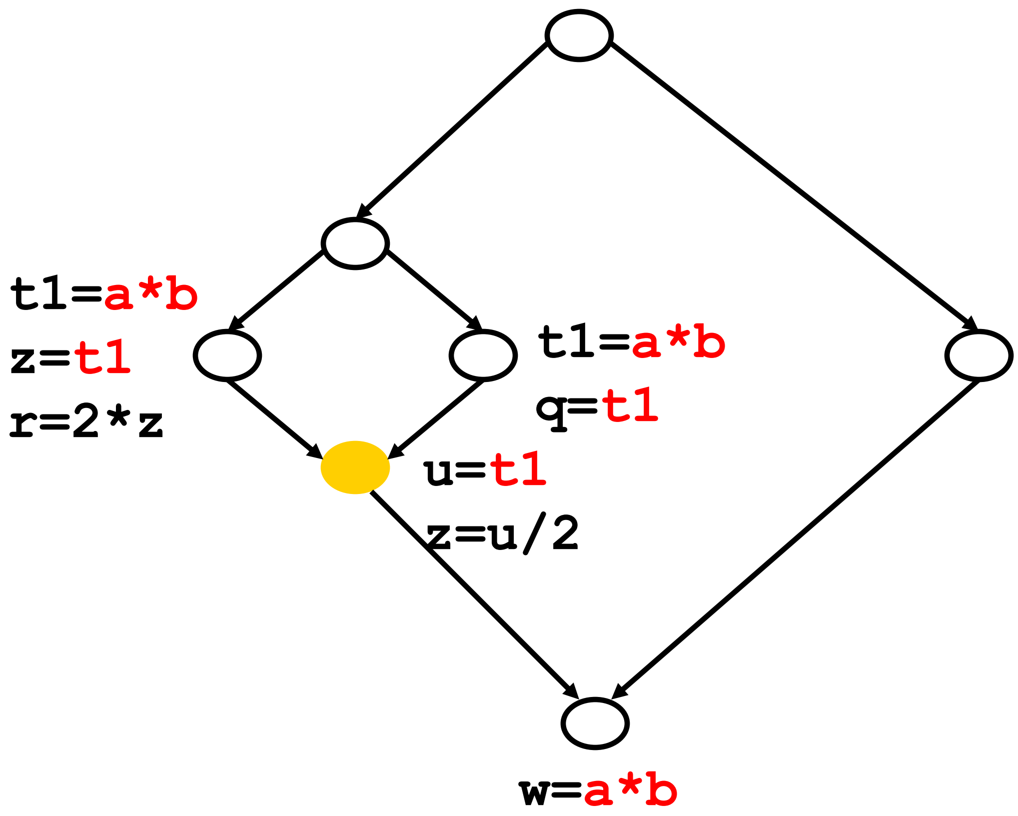


Avail Enables Global Common Subexpression Elimination



Avail Enables Global Common Subexpression Elimination

Can we eliminate **`w=a*b`**?



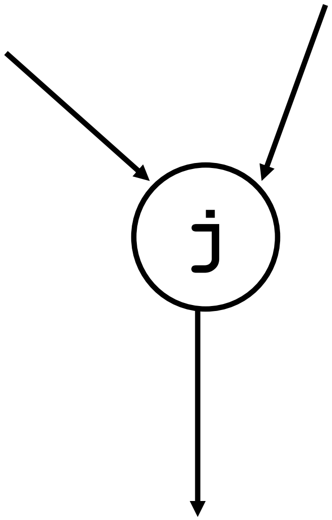
Available Expressions (*Avail*)

- Data?
  - Primitive dataflow facts are expressions, e.g., **`x+y, a*b, a+2`**
  - Analysis propagates sets of expressions, e.g., **`{x+y,a*b}`**
- Dataflow equations at **`j: x = y op z`**?
  - $out_{AE}(j) = (in_{AE}(j) - kill_{AE}(j)) \cup gen_{AE}(j)$
  - $kill_{AE}(j)$: all expressions with operand **`x`**: **`(x op _),(_ op x)`**
  - $gen_{AE}(j)$: new expression: **`{(y op z)}`**

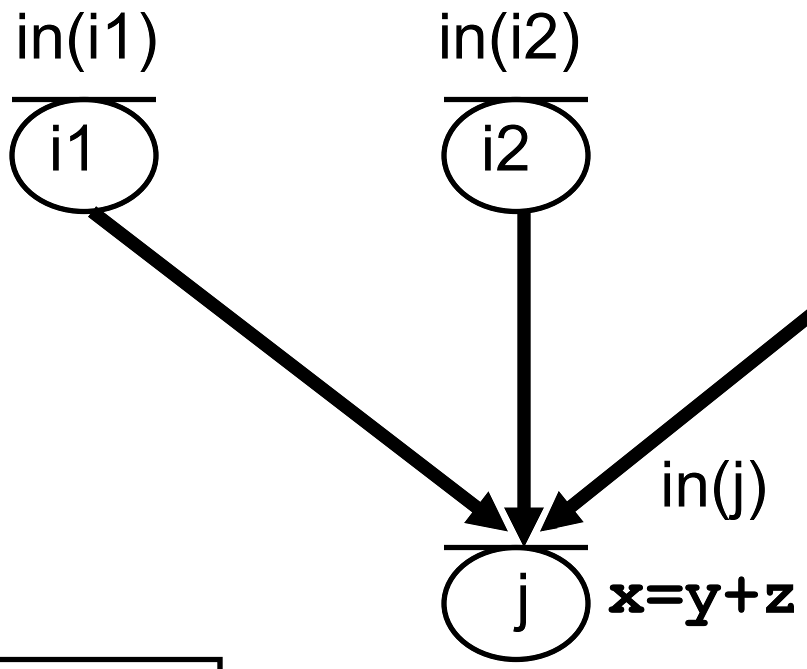
Available Expressions (*Avail*)

- Merge operator?
  - For *Avail*, it is set intersection $\bigcap$

$$in_{AE}(j) = \{ \bigcap out_{AE}(i) \mid i \text{ is predecessor of } j \}$$

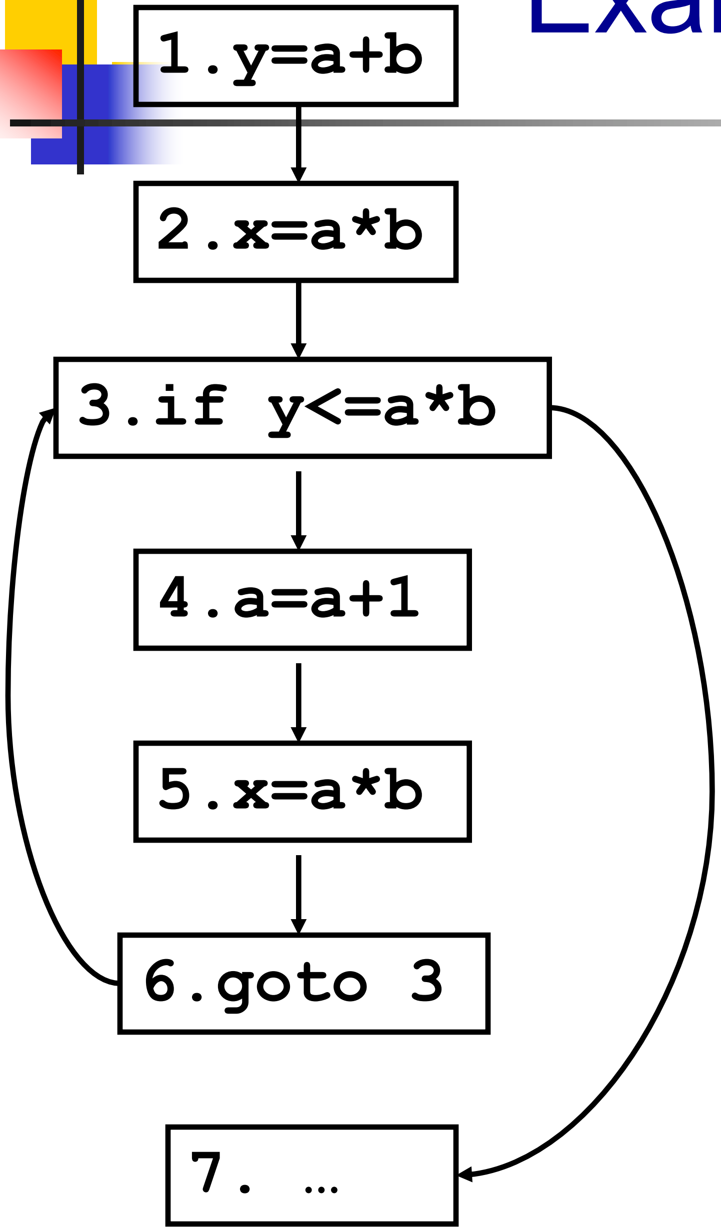


Available Expressions (*Avail*)



Forward, must dataflow problem

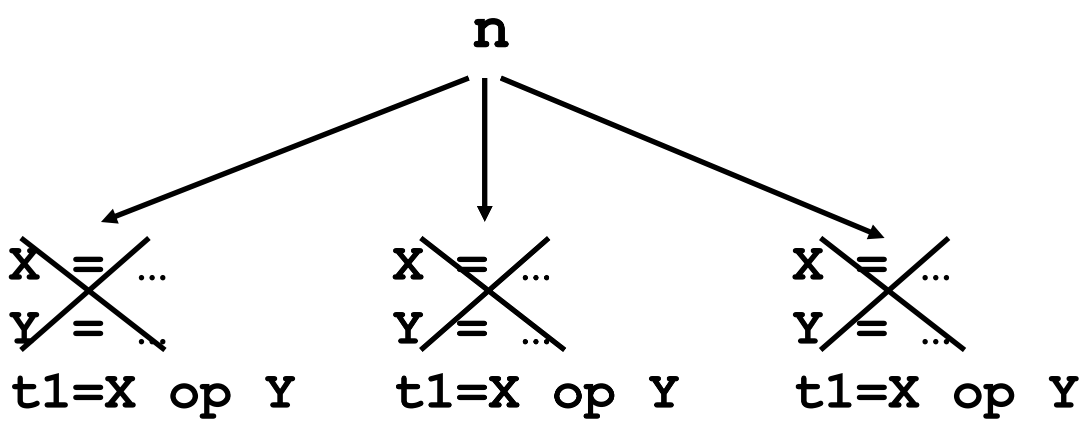
Example



Note on Homework

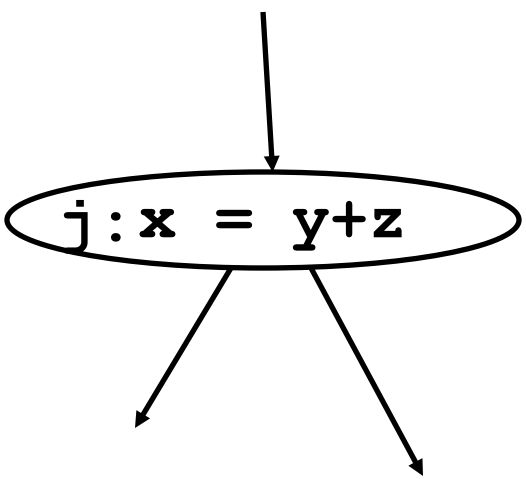
Very Busy Expressions

- An expression `x op y` is very busy at node **n**, if along EVERY path from **n** to the end of the program, we come to a computation of `x op y` BEFORE any redefinition of `x` or `y`.



Problem 4. Very Busy Expressions (*VeryB*)

- Problem Statement: For each node **`n`**, compute the set of expressions that are very busy on exit from **`n`**.



Q: What is the data?

Q: What are the equations?

Q: What is $gen_{VB}(i)$?
Q: What is $kill_{VB}(i)$?

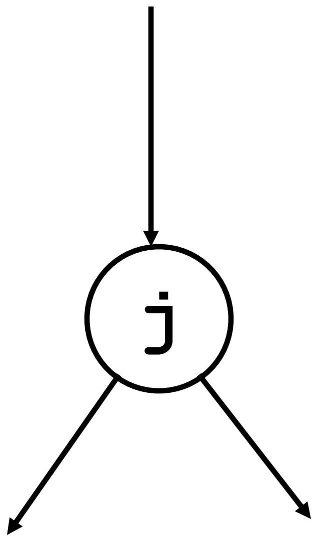
Q: What is the merge operator?

Very Busy Expressions (*VeryB*)

- Data?
  - Primitive dataflow facts are expressions, e.g., **`x+y, a*b`**
  - Analysis propagates sets of expressions, e.g., **`{x+y,a*b}`**
- Dataflow equations at **`j: x = y op z`**?
  - in(j) = gen(j) U (out(j) – kill(j))
  - kill(j): all expressions with operand **`x`**: **`(x op _),(_ op x)`**
  - gen(j): new expression: **`{ (y op z) }`**

Very Busy Expressions (*VeryB*)

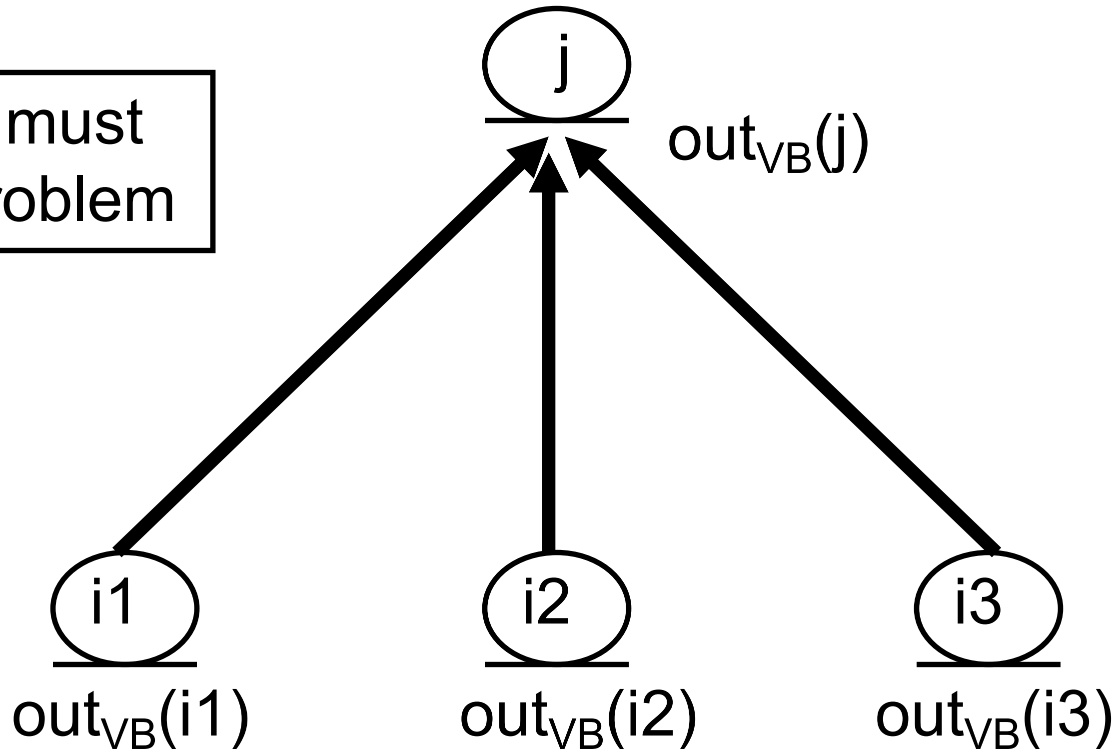
- Merge operator?
  - For *VeryB*, it is set intersection $\bigcap$



$$out_{VB}(j) = \{ \bigcap in_{VB}(i) \mid i \text{ is successor of } j \}$$

Very Busy Expressions

Backward, must dataflow problem



Dataflow Analysis Problems

	May Analyses	*Must* Analyses
Forward Analyses	Reaching Definitions	Available Expressions
Backward Analyses	Live Uses of Variables	Very Busy Expressions

Similarities

- In all cases, analysis operates on a finite set D of primitive dataflow facts:
  - *Reach*: D is the set of all definitions in the program:
    e.g., { **(x,1),(y,2),(x,4),(y,5)** }
  - *Avail* and *VeryB*: D is the set of all arithmetic expressions:
    e.g., { **a+b,a*b,a+1** }
  - *Live*: D is the set of all variables
    e.g., { **x,y,z** }
- Solution at node **n** is a subset of D (a definition either reaches node **n** or it does not reach node **n**)

Similarities

- Dataflow equations (i.e., transfer functions) for forward problems have generic form:

  out(j) = (in(j) – kill(j)) U gen(j) =

  (in(j) ∩ pres(j)) U gen(j)

  in(j) = { V out(i) | i is predecessor of j }

  Note: pres(j) is the complement of kill(j), D – kill(j)

  Note: What makes the 4 classical problems special is that sets pres(j) and gen(j) do not depend on in(j)

- Set union and set intersection can be implemented as logical OR and AND respectively