

# Lecture 11: Code Optimization

CS 540

George Mason University

# Code Optimization

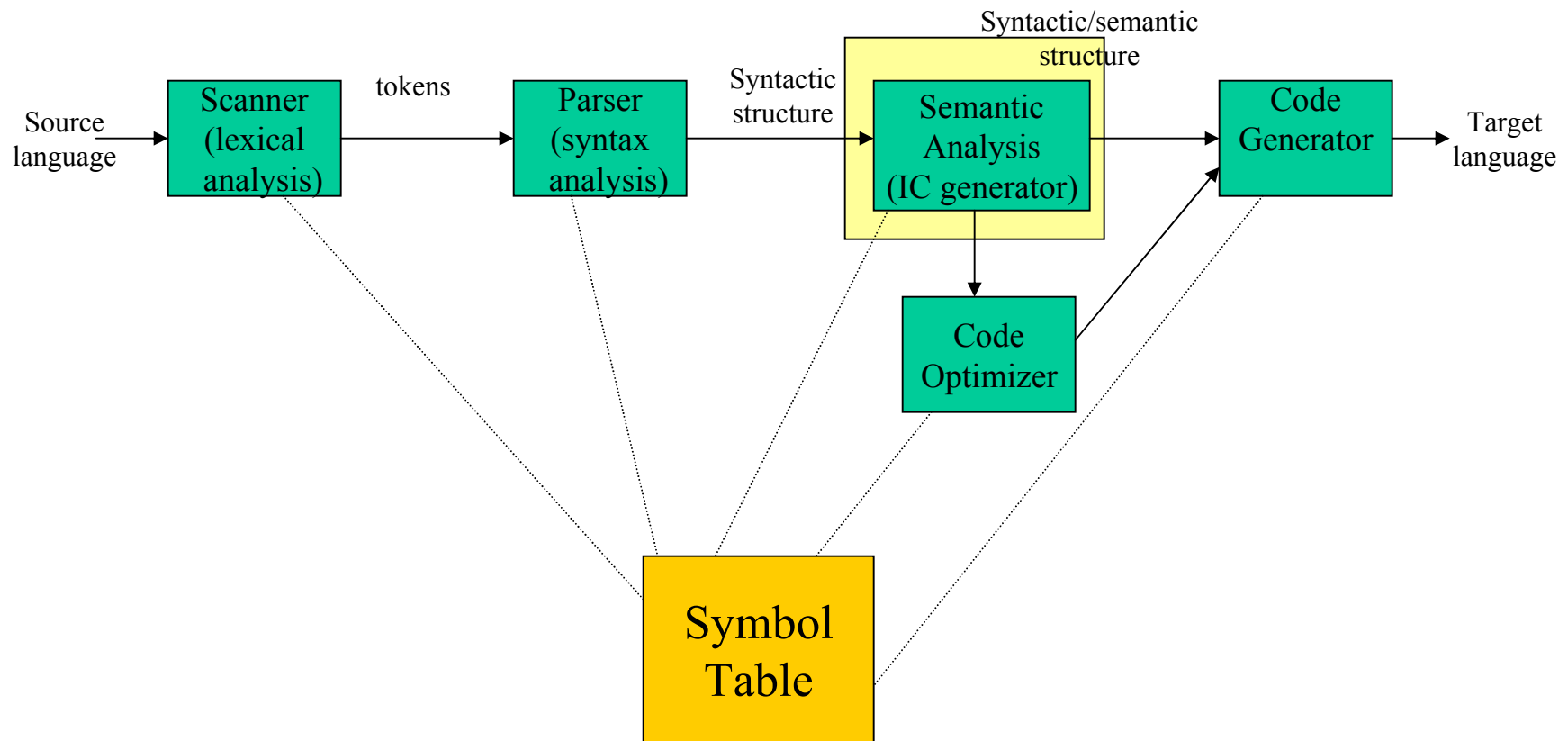
## REQUIREMENTS:

- Meaning must be preserved (correctness)
- Speedup must occur on average.
- Work done must be worth the effort.

## OPPORTUNITIES:

- Programmer (algorithm, directives)
- Intermediate code
- Target code

# Code Optimization



# Levels

- Window – peephole optimization
- Basic block
- Procedural – global (control flow graph)
- Program level – intraprocedural (program dependence graph)

# Peephole Optimizations

- Constant Folding

**x := 32** becomes **x := 64**

**x := x + 32**

- Unreachable Code

**goto L2**

**x := x + 1** ← **unneded**

- Flow of control optimizations

**goto L1** becomes **goto L2**

...

**L1: goto L2**

# Peephole Optimizations

- Algebraic Simplification

$\mathbf{x} := \mathbf{x} + 0 \leftarrow$  unneeded

- Dead code

$\mathbf{x} := 32 \leftarrow$  where  $\mathbf{x}$  not used after statement

$\mathbf{y} := \mathbf{x} + \mathbf{y} \quad \rightarrow \quad \mathbf{y} := \mathbf{y} + 32$

- Reduction in strength

$\mathbf{x} := \mathbf{x} * 2 \quad \rightarrow \quad \mathbf{x} := \mathbf{x} + \mathbf{x}$

# Peephole Optimizations

- Local in nature
- Pattern driven
- Limited by the size of the window

# Basic Block Level

- Common Subexpression elimination
- Constant Propagation
- Dead code elimination
- Plus many others such as copy propagation, value numbering, partial redundancy elimination, ...

# Simple example: $a[i+1] = b[i+1]$

- $t1 = i+1$
- $t2 = b[t1]$
- $t3 = i + 1$
- $a[t3] = t2$
- $t1 = i + 1$
- $t2 = b[t1]$
- $t3 = i + 1 \quad \leftarrow \textit{no longer live}$
- $a[t1] = t2$

Common expression can be eliminated

Now, suppose  $i$  is a constant:

- $i = 4$
  - $t1 = i+1$
  - $t2 = b[t1]$
  - $a[t1] = t2$
- $i = 4$
  - $t1 = 5$
  - $t2 = b[t1]$
  - $a[t1] = t2$
- $i = 4$
  - $t1 = 5$
  - $t2 = b[5]$
  - $a[5] = t2$

- Final Code:
- $i = 4$
  - $t2 = b[5]$
  - $a[5] = t2$

# Control Flow Graph - CFG

CFG =  $\langle V, E, \text{Entry} \rangle$ , where

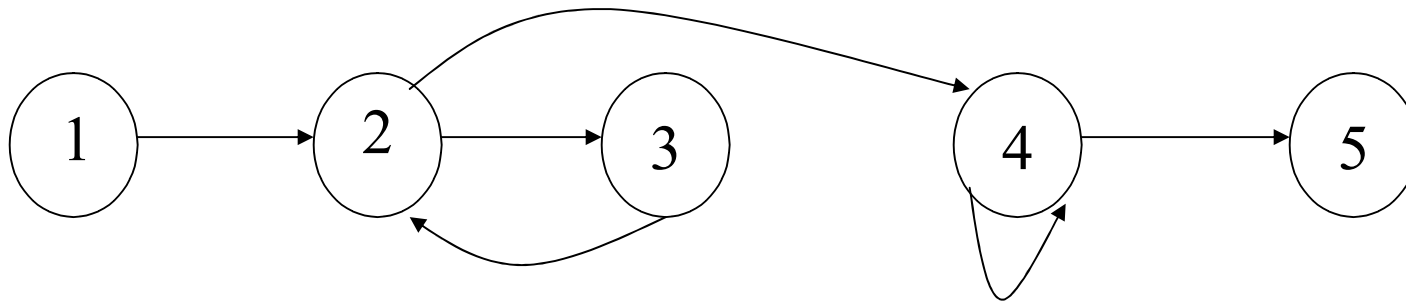
$V$  = vertices or nodes, representing an instruction or basic block (group of statements).

$E = (V \times V)$  edges, potential flow of control

Entry is an element of  $V$ , the unique program entry

Two sets used in algorithms:

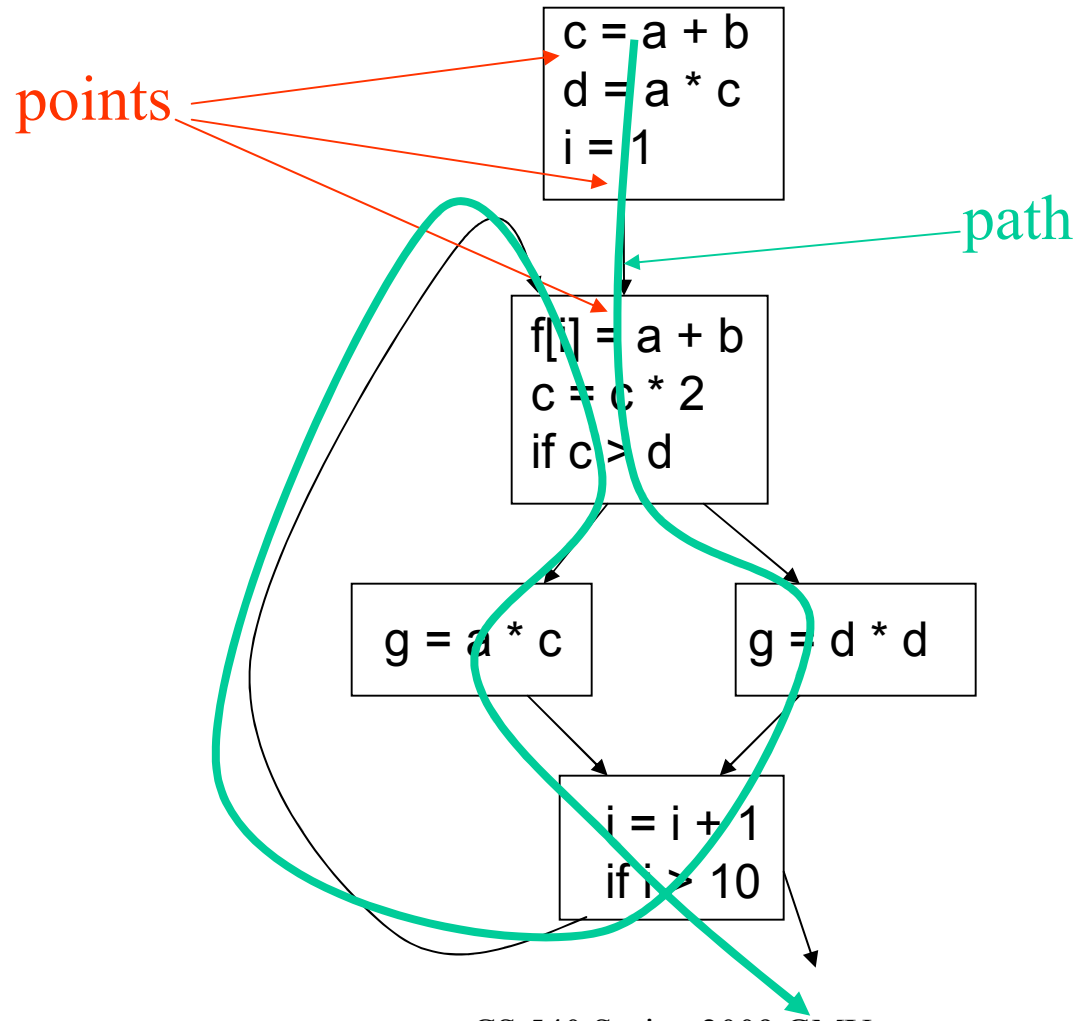
- $\text{Succ}(v) = \{x \text{ in } V \mid \text{exists } e \text{ in } E, e = v \rightarrow x\}$
- $\text{Pred}(v) = \{x \text{ in } V \mid \text{exists } e \text{ in } E, e = x \rightarrow v\}$



# Definitions

- **point** - any location between adjacent statements and before and after a basic block.
- A **path** in a CFG from point  $p_1$  to  $p_n$  is a sequence of points such that  $\forall j, 1 \leq j < n$ , either  $p_j$  is the point immediately preceding a statement and  $p_{j+1}$  is the point immediately following that statement in the same block, or  $p_j$  is the end of some block and  $p_{j+1}$  is the start of a successor block.

# CFG



# Optimizations on CFG

- Must take control flow into account
  - Common Sub-expression Elimination
  - Constant Propagation
  - Dead Code Elimination
  - Partial redundancy Elimination
  - ...
- Applying one optimization may create opportunities for other optimizations.

# Redundant Expressions

An expression  $\mathbf{x}$   $\mathbf{op}$   $\mathbf{y}$  is redundant at a point  $p$  if it has already been computed at some point(s) and no intervening operations redefine  $\mathbf{x}$  or  $\mathbf{y}$ .

$$m = 2*y*z$$

$$n = 3*y*z$$

$$o = 2*y-z$$

redundant



$$t0 = 2*y$$

$$m = t0*z$$

$$t1 = 3*y$$

$$n = t1*z$$

$$t2 = 2*y$$

$$o = t2-z$$

$$t0 = 2*y$$

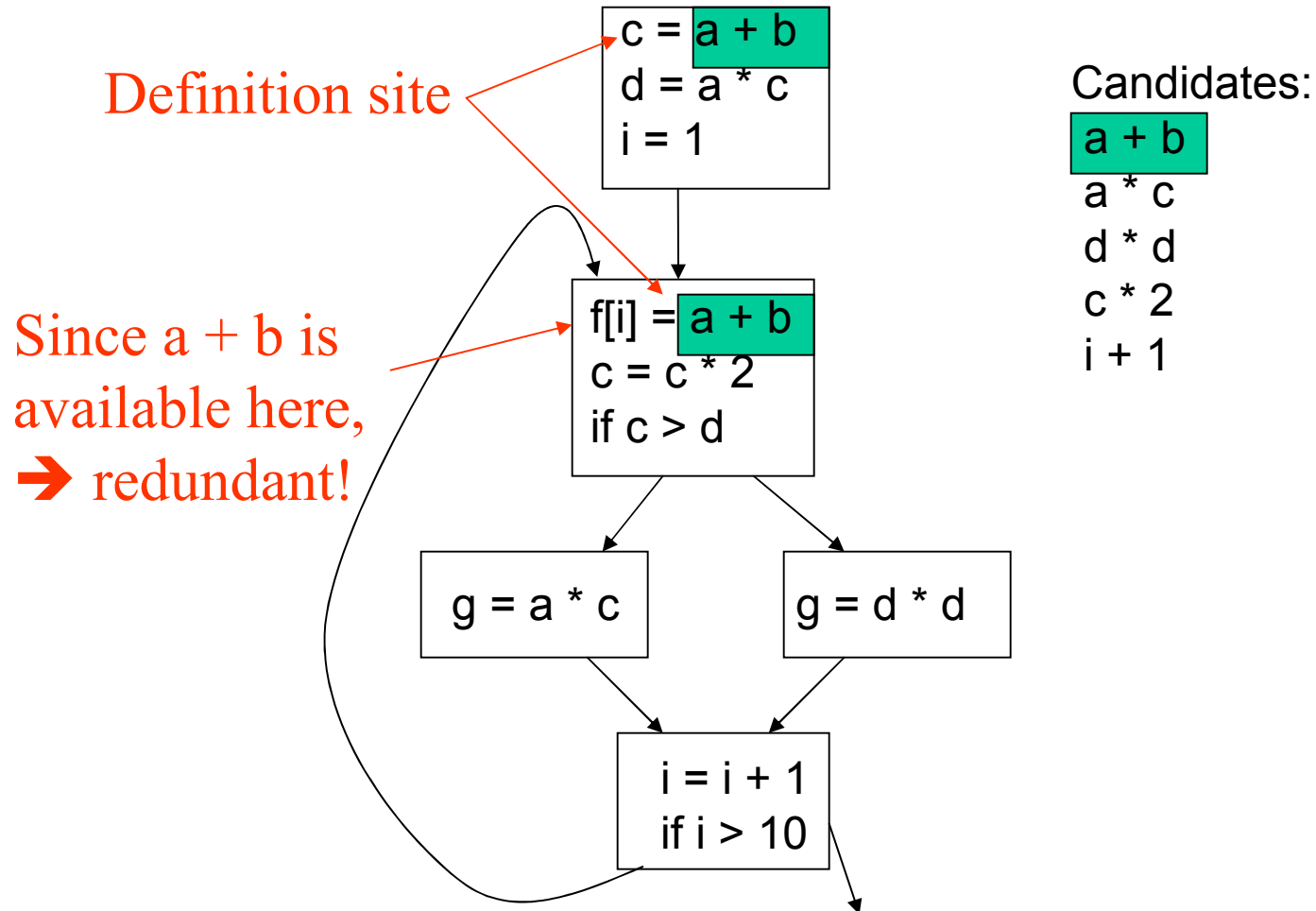
$$m = t0*z$$

$$t1 = 3*y$$

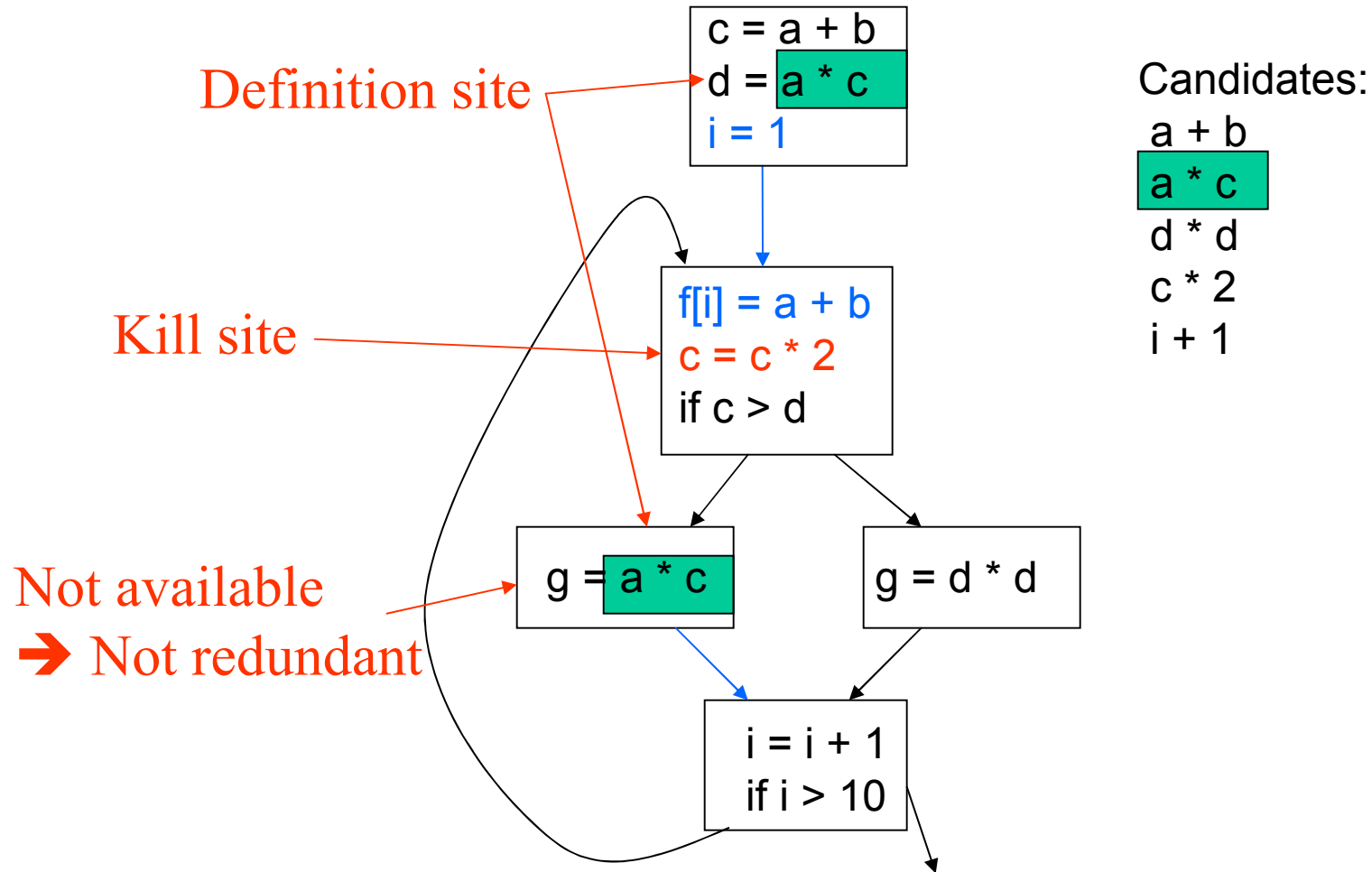
$$n = t1*z$$

$$o = t0-z$$

# Redundant Expressions



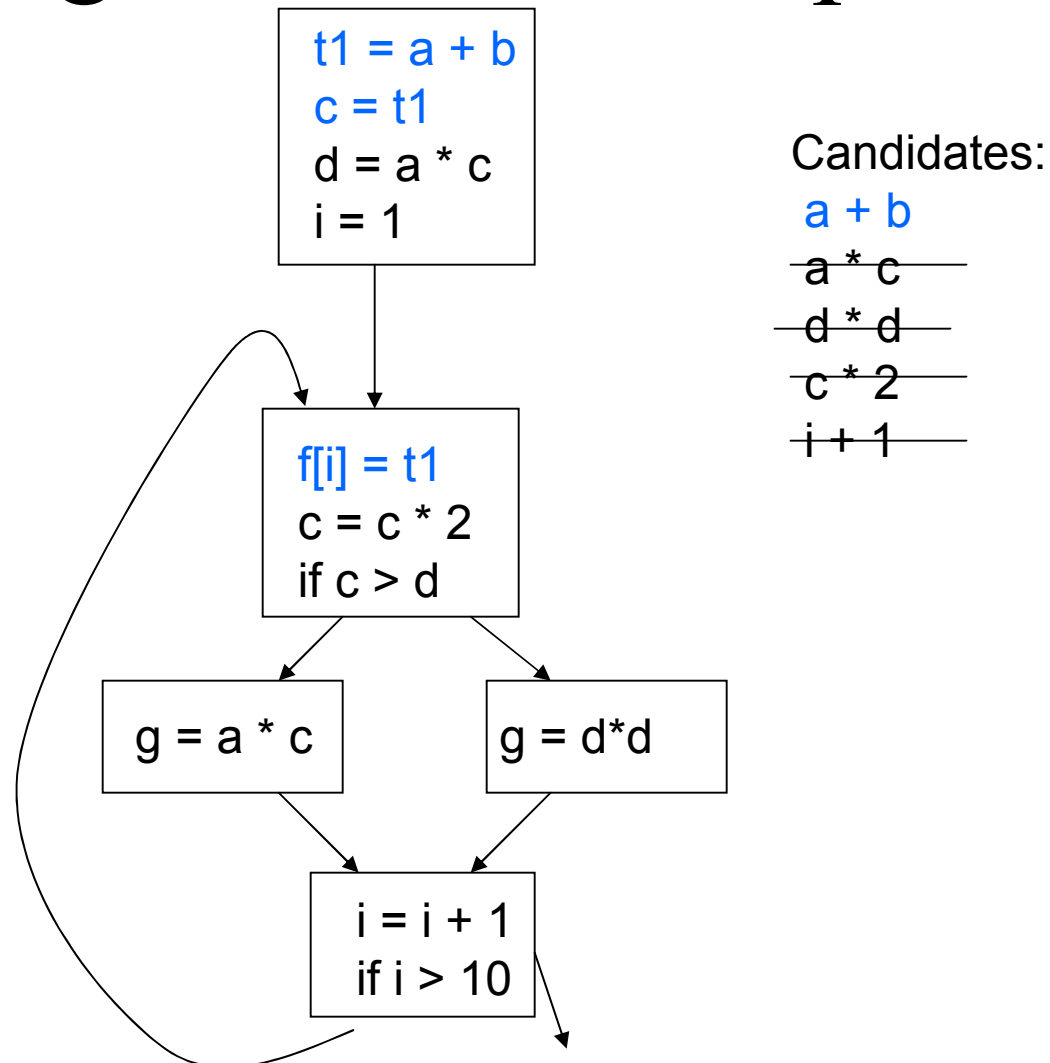
# Redundant Expressions



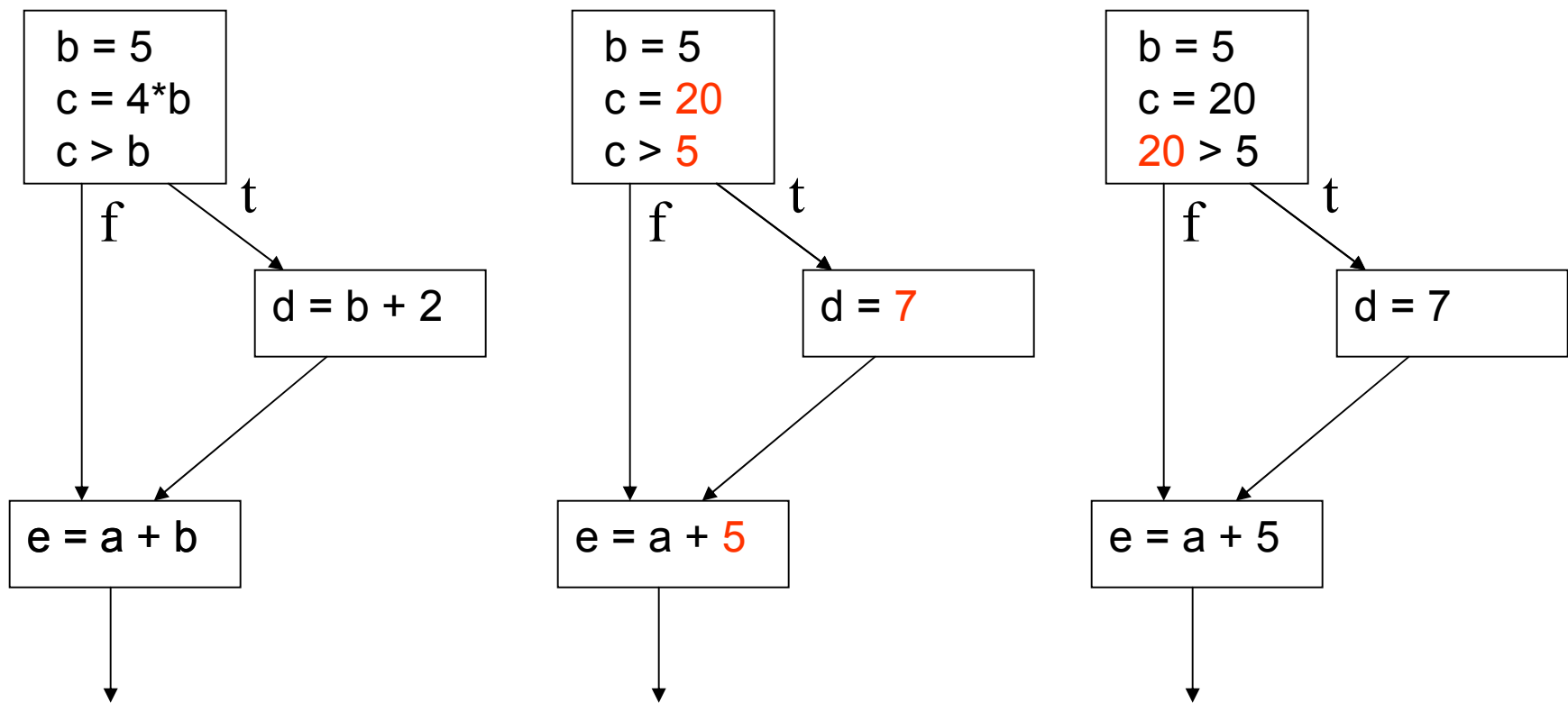
# Redundant Expressions

- An expression  $e$  is defined at some point  $p$  in the CFG if its value is computed at  $p$ . (definition site)
- An expression  $e$  is killed at point  $p$  in the CFG if one or more of its operands is defined at  $p$ . (kill site)
- An expression is *available* at point  $p$  in a CFG if every path leading to  $p$  contains a prior definition of  $e$  and  $e$  is not killed between that definition and  $p$ .

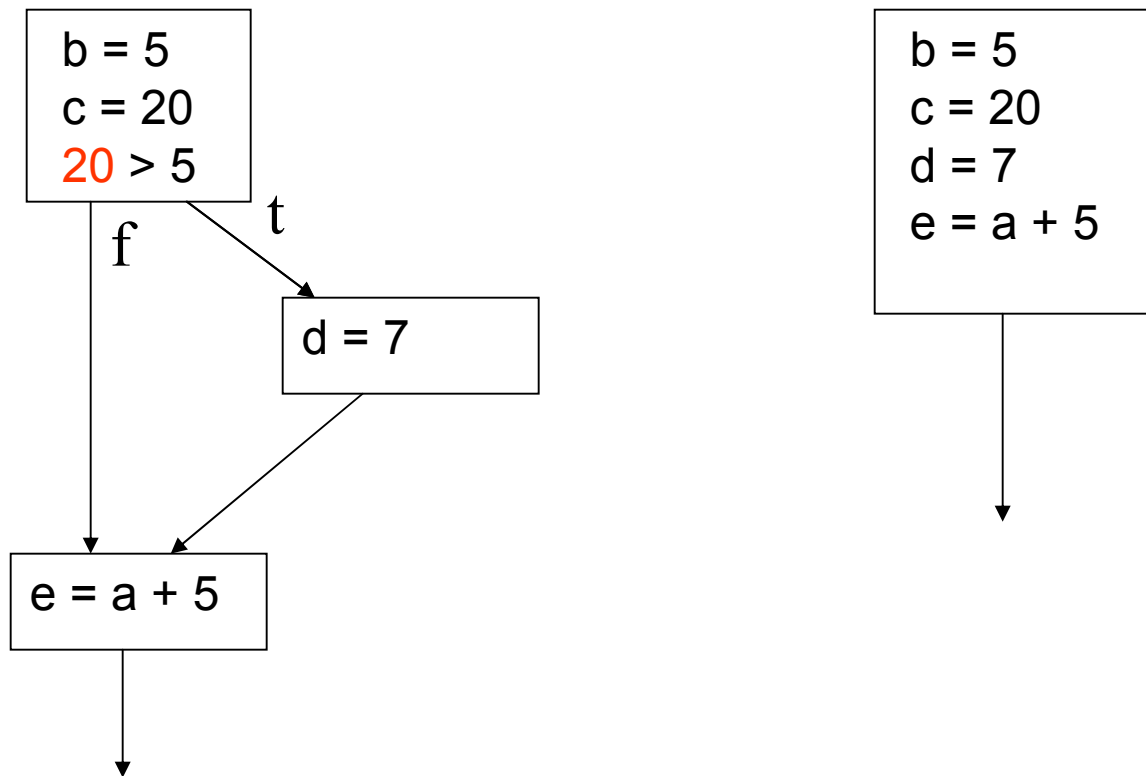
# Removing Redundant Expressions



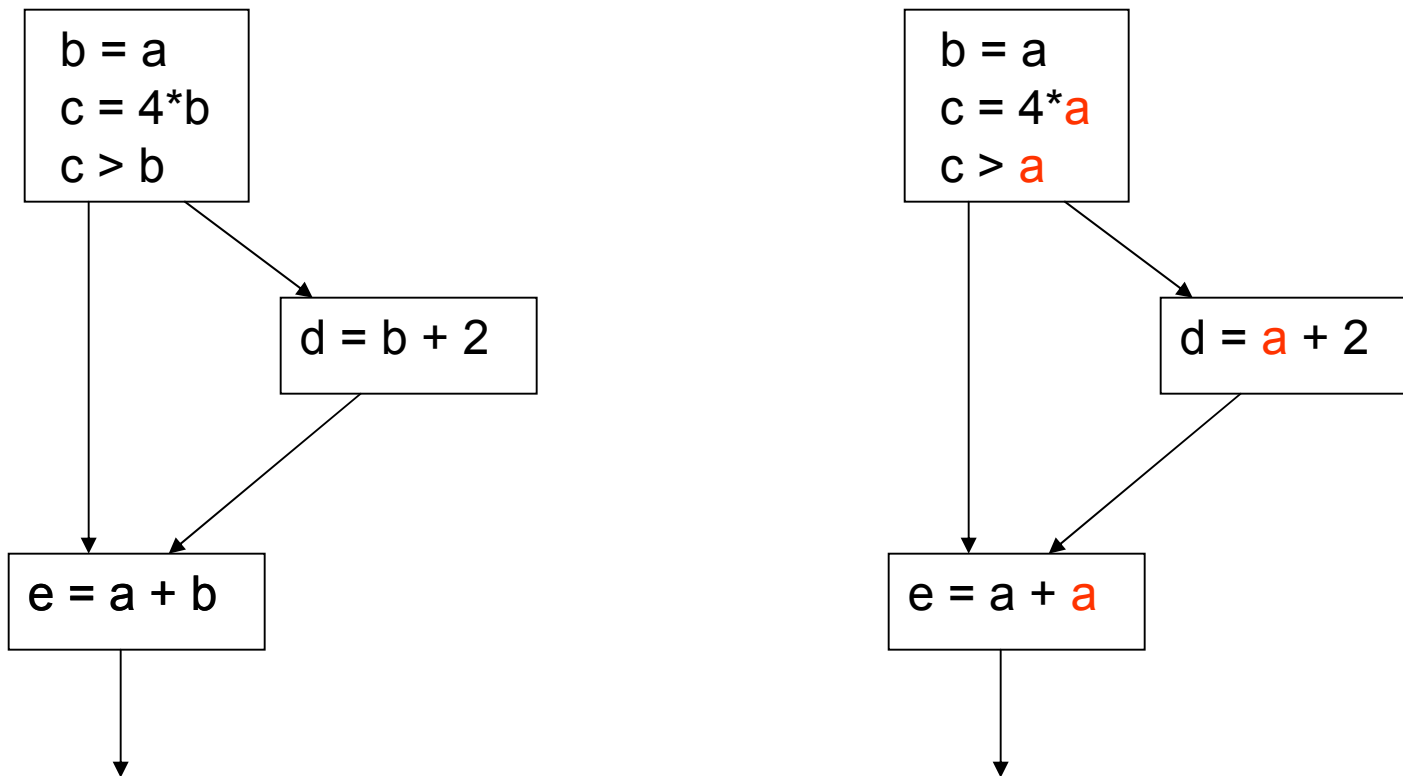
# Constant Propagation



# Constant Propagation



# Copy Propagation



# Simple Loop Optimizations: Code Motion

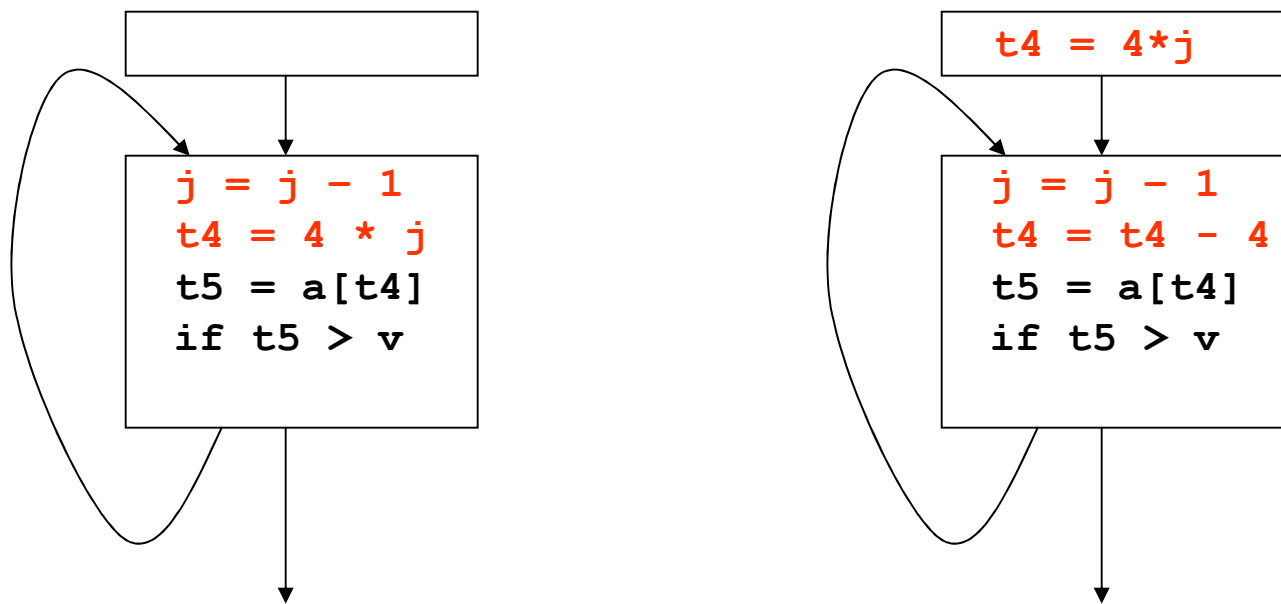
```
while (i <= limit - 2)      L1:
                             t1 = limit - 2
                             if (i > t1) goto L2
                             body of loop
                             goto L1
                             L2:
```

---

```
t := limit - 2
  while (i <= t)            t1 = limit - 2
                             L1:
                             if (i > t1) goto L2
                             body of loop
                             goto L1
                             L2:
```

# Simple Loop Optimizations: Strength Reduction

- Induction Variables control loop iterations

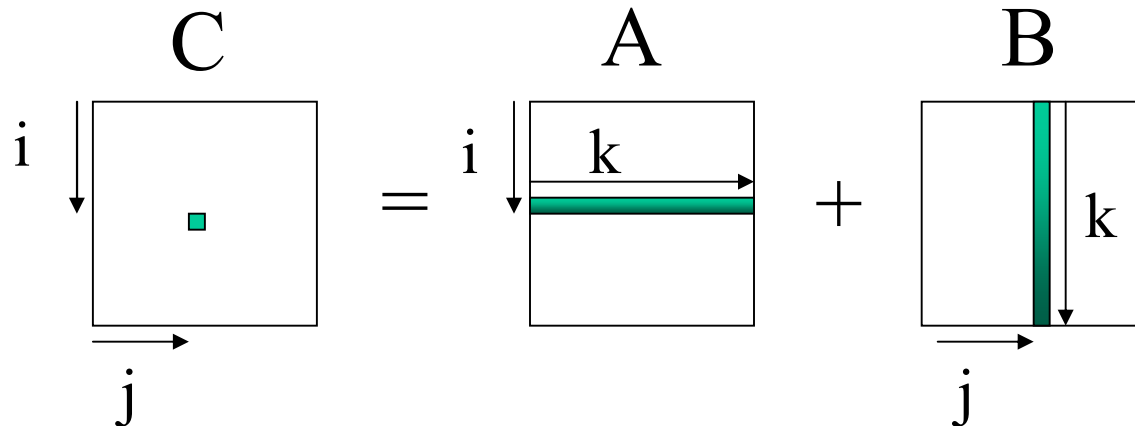


# Simple Loop Optimizations

- Loop transformations are often used to expose other optimization opportunities:
  - Normalization
  - Loop Interchange
  - Loop Fusion
  - Loop Reversal
  - ...

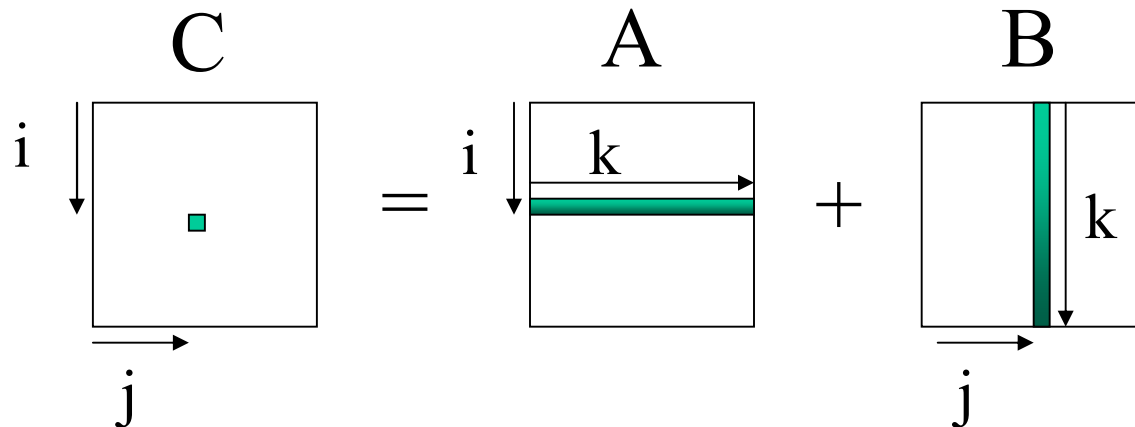
# Consider Matrix Multiplication

```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do
      C[i,j] = C[i,j] + A[i,k] + B[k,j]
    end
  end
end
```



# Memory Usage

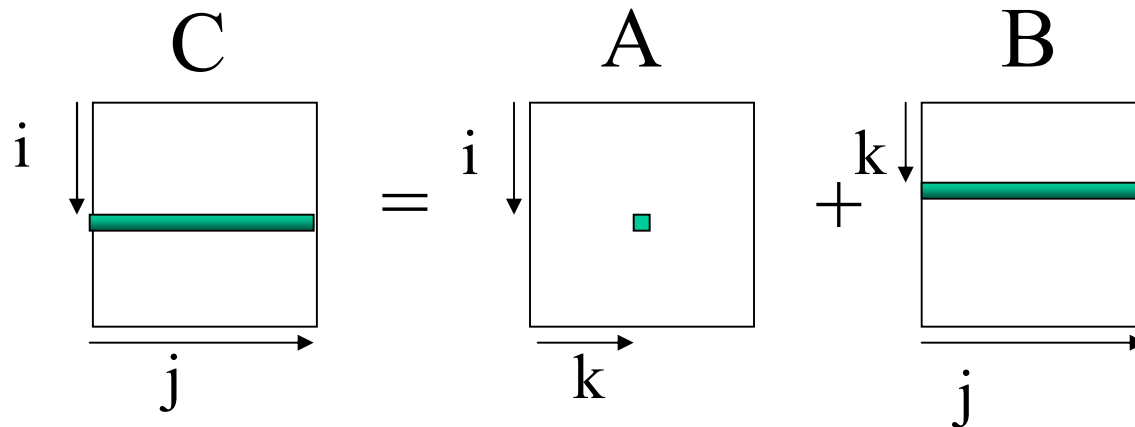
- **For A:** Elements are accessed across rows, spatial locality is exploited for cache (assuming row major storage)
- **For B:** Elements are accessed along columns, unless cache can hold all of B, cache will have problems.
- **For C:** Single element computed per loop – use register to hold



# Matrix Multiplication Version 2

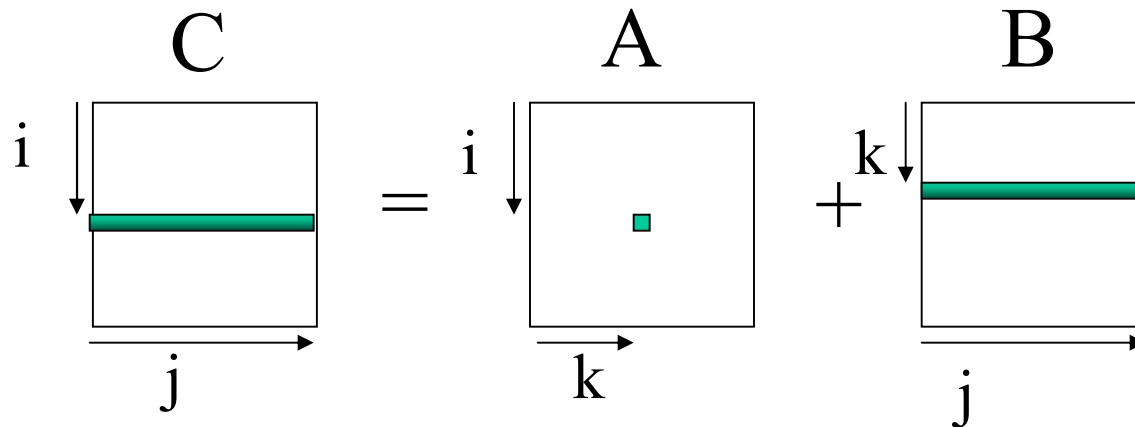
```
for i = 1 to n do
  for k = 1 to n do
    for j = 1 to n do
      C[i,j] = C[i,j] + A[i,k] + B[k,j]
    end
  end
end
```

loop interchange



# Memory Usage

- **For A:** Single element loaded for loop body
- **For B:** Elements are accessed along rows to exploit spatial locality.
- **For C:** Extra loading/storing, but across rows



# Simple Loop Optimizations

- How to determine safety?
  - Does the new multiply give the same answer?
  - Can be reversed??
    - `for (I=1 to N) a[I] = a[I+1]` – can this loop be safely reversed?

# Data Dependencies

- Flow Dependencies - write/read

$x := 4;$

$y := x + 1$

- Output Dependencies - write/write

$x := 4;$

$x := y + 1;$

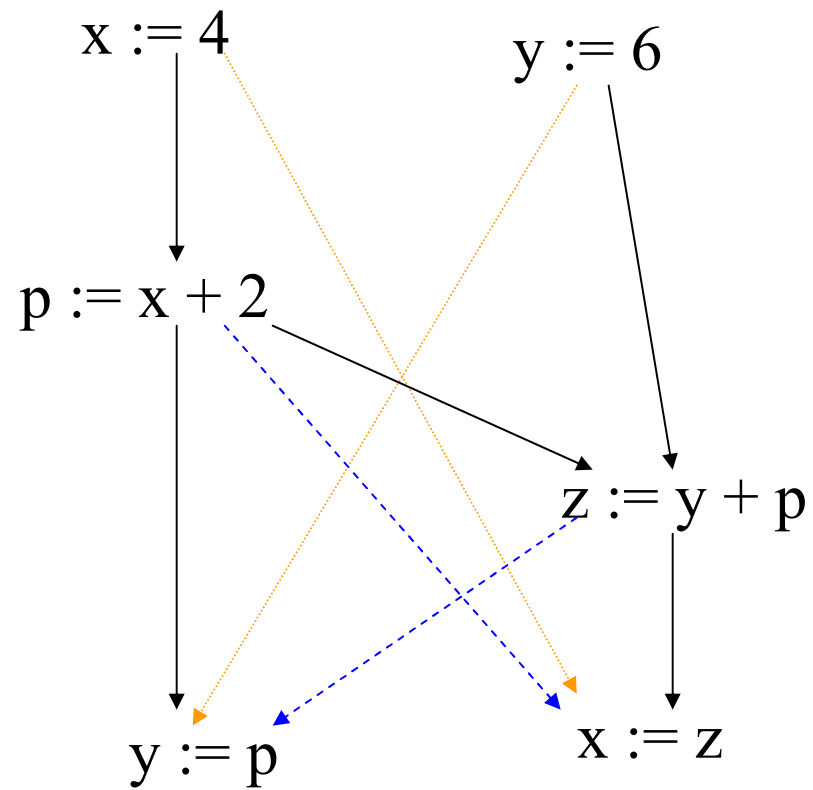
- Antidependencies - read/write

$y := x + 1;$

$x := 4;$

$x := 4$   
 $y := 6$   
 $p := x + 2$   
 $z := y + p$   
 $x := z$   
 $y := p$

—————> Flow  
 - - - - -> Output  
 - - - - -> Anti



# Global Data Flow Analysis

Collecting information about the way data is used in a program.

- Takes control flow into account
- HL control constructs
  - Simpler – syntax driven
  - Useful for data flow analysis of source code
- **General control constructs – arbitrary branching**

Information needed for optimizations such as:  
constant propagation, common sub-expressions,  
partial redundancy elimination ...

# Dataflow Analysis: Iterative Techniques

- First, compute local (block level) information.
- Iterate until no changes

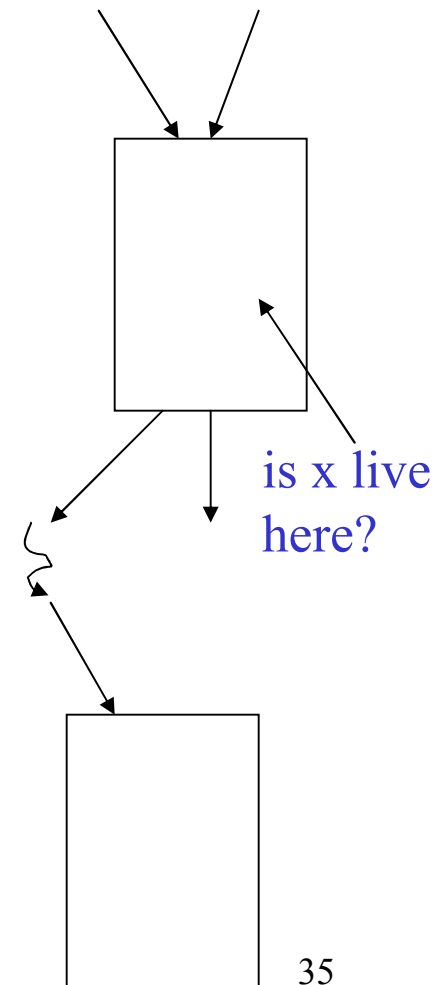
```
while change do
  change = false
  for each basic block
    apply equations updating IN and OUT
    if either IN or OUT changes, set change
  to true
end
```

# Live Variable Analysis

A variable  $x$  is *live* at a point  $p$  if there is some path from  $p$  where  $x$  is used before it is defined.

Want to determine for some variable  $x$  and point  $p$  whether the value of  $x$  *could* be used along some path starting at  $p$ .

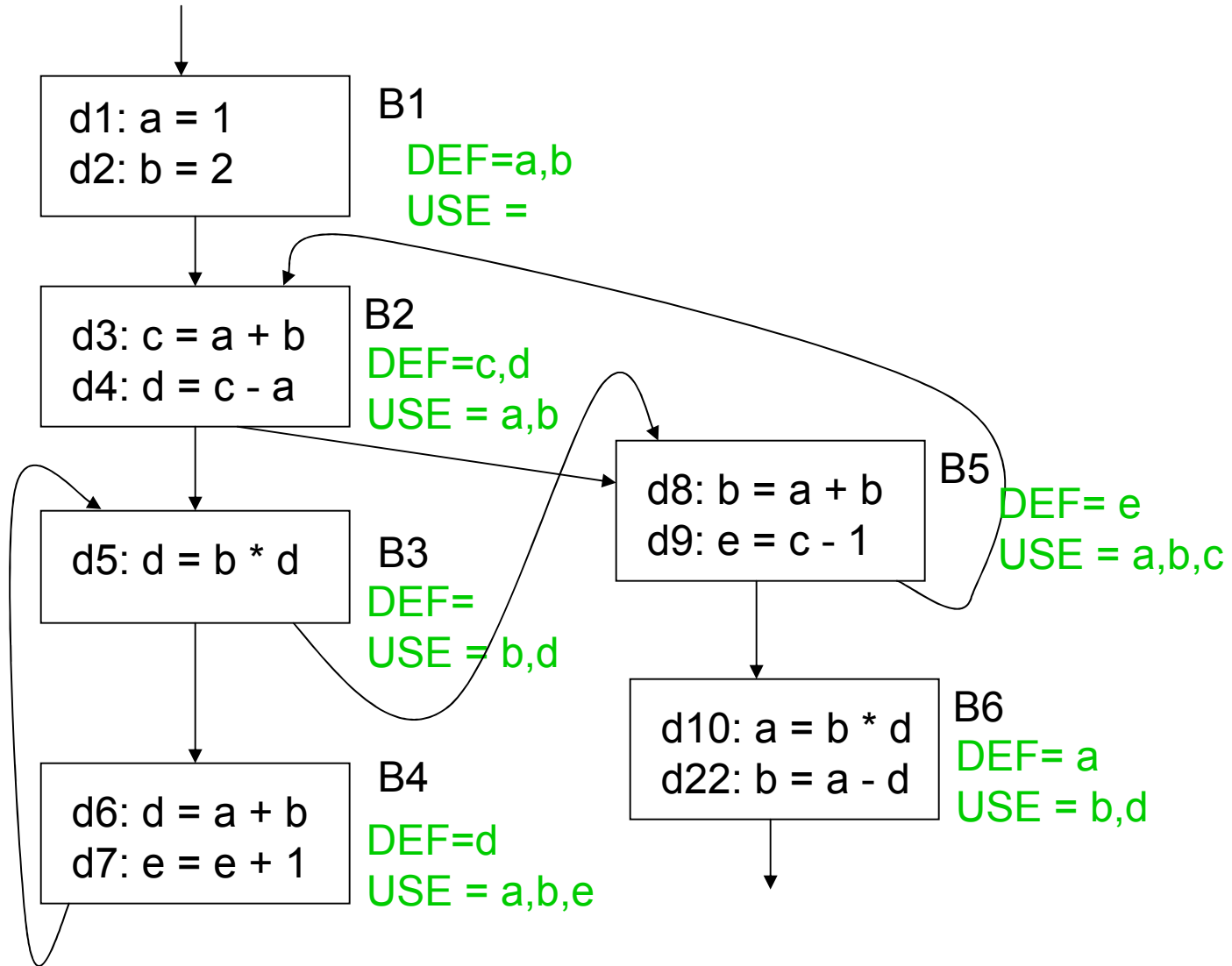
- Information flows backwards
- May – ‘along some path starting at  $p$ ’



# Global Live Variable Analysis

Want to determine for some variable  $x$  and point  $p$  whether the value of  $x$  could be used along some path starting at  $p$ .

- DEF[B] - set of variables assigned values in B prior to any use of that variable
- USE[B] - set of variables used in B prior to any definition of that variable
- OUT[B] - variables live immediately after the block  
 $OUT[B] = \bigcup IN[S]$  for all S in succ(B)
- IN[B] - variables live immediately before the block  
 $IN[B] = USE[B] + (OUT[B] - DEF[B])$



# Global Live Variable Analysis

Want to determine for some variable  $x$  and point  $p$  whether the value of  $x$  could be used along some path starting at  $p$ .

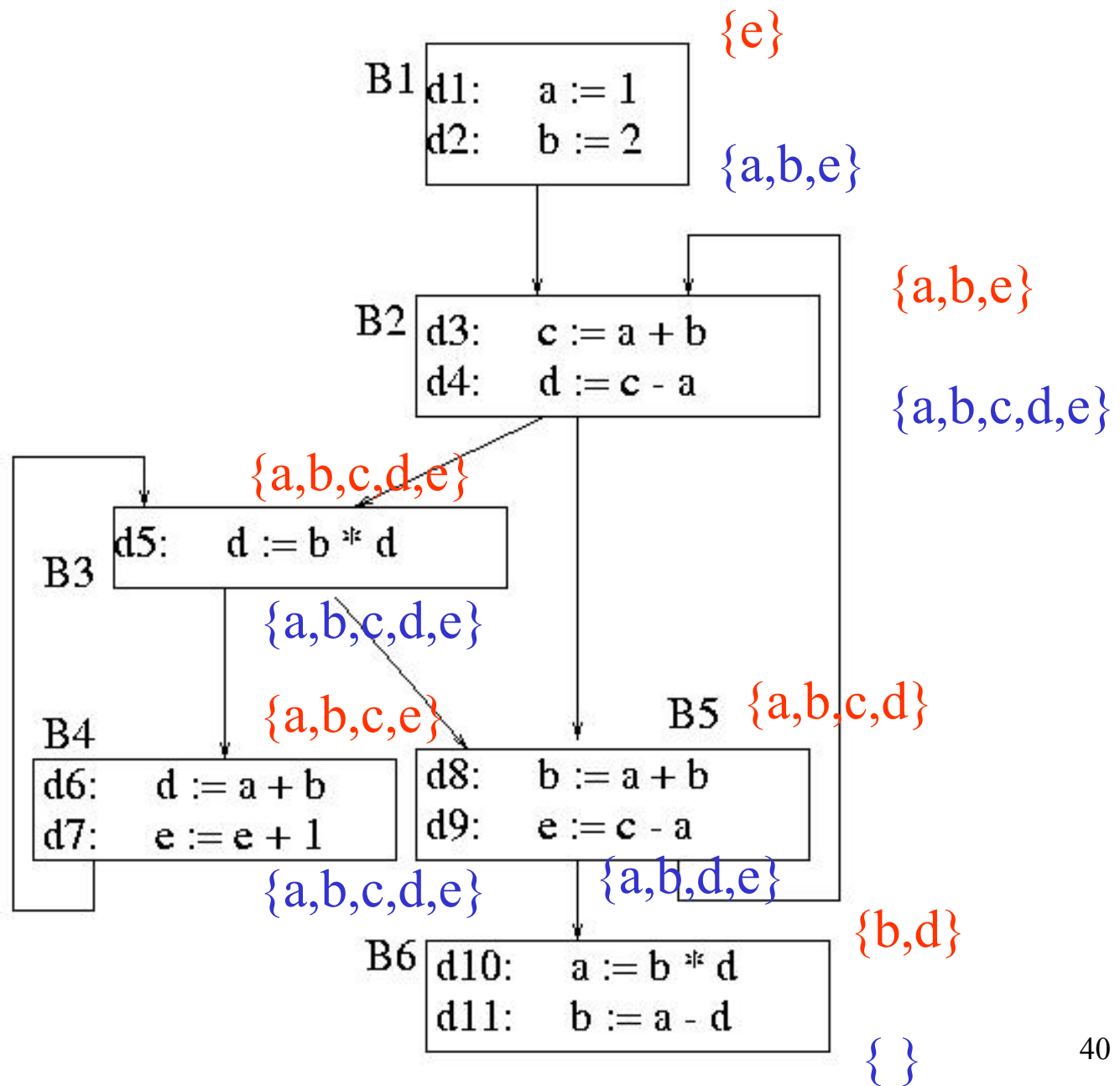
- DEF[B] - set of variables assigned values in B prior to any use of that variable
- USE[B] - set of variables used in B prior to any definition of that variable

- OUT[B] - variables live immediately after the block  
 $OUT[B] = \bigcup IN[S]$  for all  $S$  in  $succ(B)$
- IN[B] - variables live immediately before the block  
 $IN[B] = USE[B] \cup (OUT[B] - DEF[B])$

|    | IN          | OUT         | IN          | OUT         | IN        | OUT         |
|----|-------------|-------------|-------------|-------------|-----------|-------------|
| B1 | $\emptyset$ | a,b         | $\emptyset$ | a,b         | e         | a,b,e       |
| B2 | a,b         | a,b,c,d     | a,b,e       | a,b,c,d,e   | a,b,e     | a,b,c,d,e   |
| B3 | a,b,c,d,e   | a,b,c,e     | a,b,c,d,e   | a,b,c,d,e   | a,b,c,d,e | a,b,c,d,e   |
| B4 | a,b,c,e     | a,b,c,d,e   | a,b,c,e     | a,b,c,d,e   | a,b,c,e   | a,b,c,d,e   |
| B5 | a,b,c,d     | a,b,d       | a,b,c,d     | a,b,d,e     | a,b,c,d   | a,b,d,e     |
| B6 | b,d         | $\emptyset$ | b,d         | $\emptyset$ | b,d       | $\emptyset$ |

$OUT[B] = \cup IN[S]$  for all  $S$  in  $succ(B)$   
 $IN[B] = USE[B] + (OUT[B] - DEF[B])$

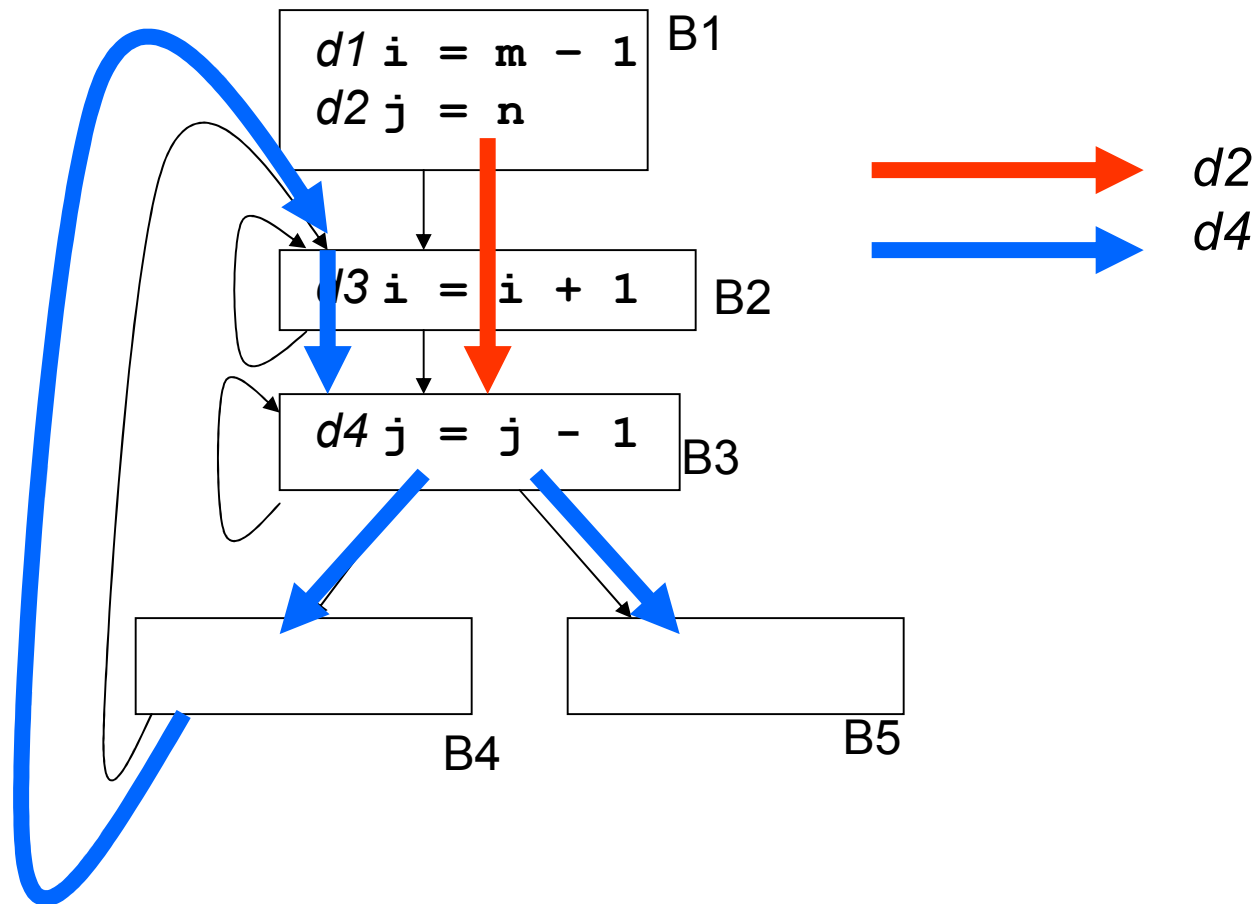
| Block | DEF   | USE     |
|-------|-------|---------|
| B1    | {a,b} | { }     |
| B2    | {c,d} | {a,b}   |
| B3    | { }   | {b,d}   |
| B4    | {d}   | {a,b,e} |
| B5    | {e}   | {a,b,c} |
| B6    | {a}   | {b,d}   |



# Dataflow Analysis Problem #2: Reachability

- A *definition of a variable  $x$*  is a statement that may assign a value to  $x$ .
- A definition may reach a program point  $p$  if there exists some path from the point immediately following the definition to  $p$  such that the assignment is not killed along that path.
- Concept: relationship between definitions and uses

# What blocks do definitions $d2$ and $d4$ reach?



# Reachability Analysis: Unstructured Input

1. Compute GEN and KILL at block—level
2. Compute IN[B] and OUT[B] for B
$$\text{IN}[B] = \cup \text{OUT}[P] \quad \text{where } P \text{ is a predecessor of } B$$
$$\text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B])$$
3. Repeat step 2 until there are no changes to OUT sets

# Reachability Analysis: Step 1

For each block, compute local (block level) information = GEN/KILL sets

- $GEN[B]$  = set of definitions generated by B
- $KILL[B]$  = set of definitions that can not reach the end of B

This information does not take control flow between blocks into account.

# Reasoning about Basic Blocks

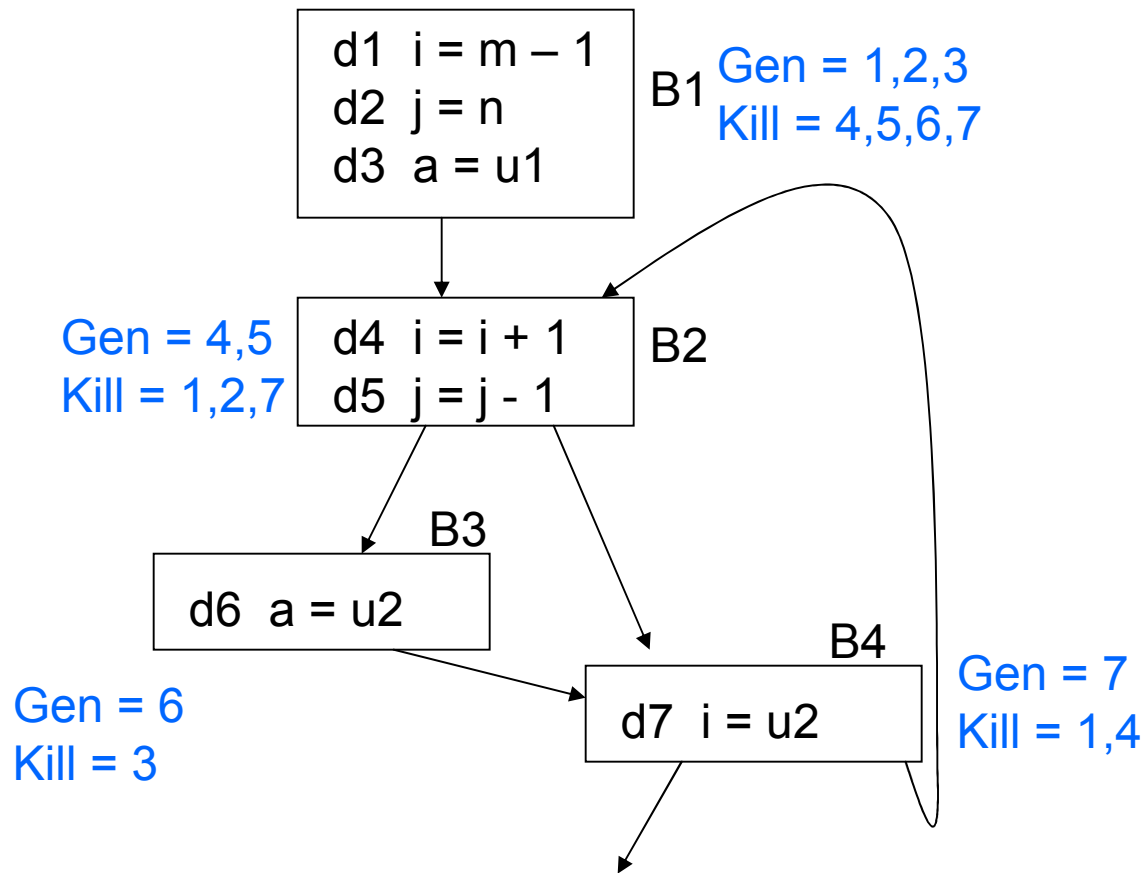
Effect of single statement:  $a = b + c$

- Uses variables  $\{b,c\}$
- **Kills all definitions of  $\{a\}$**
- **Generates new definition (i.e. assigns a value) of  $\{a\}$**

Local Analysis:

- Analyze the effect of each instruction
- Compose these effects to derive information about the entire block

# Example



# Reachability Analysis: Step 2

Compute IN/OUT for each block in a forward direction. Start with  $IN[B] = \emptyset$

- $IN[B] = \textit{set of defns reaching the start of } B$   
=  $\cup (\text{out}[P])$  for all predecessor blocks in the CFG
- $OUT[B] = \textit{set of defns reaching the end of } B$   
=  $GEN[B] \cup (IN[B] - KILL[B])$

Keep computing IN/OUT sets until a fixed point is reached.

# Reaching Definitions Algorithm

- Input: Flow graph with GEN and KILL for each block
- Output:  $in[B]$  and  $out[B]$  for each block.

For each block B do  $out[B] = gen[B]$ , (true if  $in[B] = \text{emptyset}$ )

change := true;

while change do begin

    change := false;

    for each block B do begin

$in[B] := \bigcup out[P]$ , where P is a predecessor of B;

        oldout = out[B];

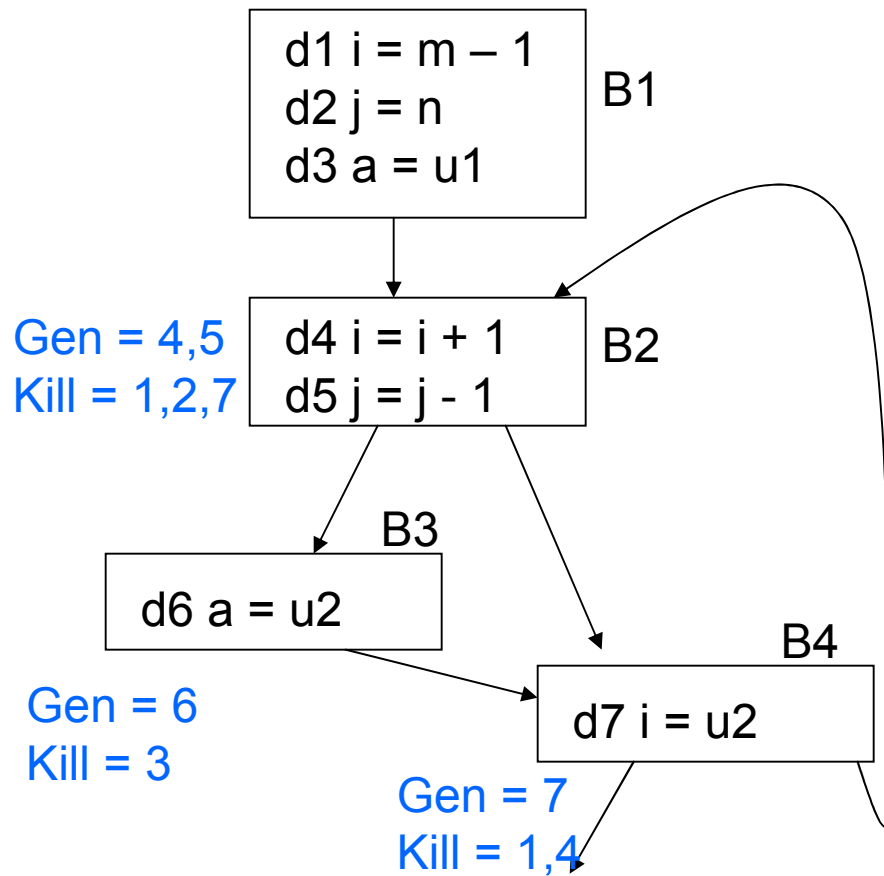
$out[B] := gen[B] \cup (in[B] - kill[B])$

        if  $out[B] \neq oldout$  then change := true;

    end

end

Gen = 1,2,3  
Kill = 4,5,6,7



|    | IN | OUT   |  |  |
|----|----|-------|--|--|
| B1 | ∅  | 1,2,3 |  |  |
| B2 | ∅  | 4,5   |  |  |
| B3 | ∅  | 6     |  |  |
| B4 | ∅  | 7     |  |  |

$IN[B] = \cup(out[P])$  for all predecessor blocks in the CFG

$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$

|    | IN          | OUT   | IN                         | OUT                                   |  |  |
|----|-------------|-------|----------------------------|---------------------------------------|--|--|
| B1 | $\emptyset$ | 1,2,3 | $\emptyset$                | 1,2,3                                 |  |  |
| B2 | $\emptyset$ | 4,5   | OUT[1]+OUT[4]<br>= 1,2,3,7 | 4,5 + (1,2,3,7<br>- 1,2,7)<br>= 3,4,5 |  |  |
| B3 | $\emptyset$ | 6     | OUT[2] = 3,4,5             | 6 + (3,4,5 - 3)<br>= 4,5,6            |  |  |
| B4 | $\emptyset$ | 7     | OUT[2]+OUT[3]<br>= 3,4,5,6 | 7 + (3,4,5,6 - 1,4) = 3,5,6,7         |  |  |

IN[B] =  $\cup(\text{out}[P])$  for all predecessor  
blocks in the CFG

OUT[B] = GEN[B] + (IN[B] - KILL[B])

|    | IN          | OUT   | IN          | OUT     | IN   | OUT  |
|----|-------------|-------|-------------|---------|--|--|
| B1 | $\emptyset$ | 1,2,3 | $\emptyset$ | 1,2,3   | $\emptyset$                                      | 1,2,3                                      |
| B2 | $\emptyset$ | 4,5   | 1,2,3,7     | 3,4,5   | $\text{OUT}[1] + \text{OUT}[4] =$<br>1,2,3,5,6,7 | $4,5 + (1,2,3,5,6,7 - 1,2,7) =$<br>3,4,5,6 |
| B3 | $\emptyset$ | 6     | 3,4,5       | 4,5,6   | $\text{OUT}[2] = 3,4,5,6$                        | $6 + (3,4,5,6 - 3)$<br>$= 4,5,6$           |
| B4 | $\emptyset$ | 7     | 3,4,5,6     | 3,5,6,7 | $\text{OUT}[2] + \text{OUT}[3] =$<br>3,4,5,6     | $7 + (3,4,5,6 - 1,4)$<br>$= 3,5,6,7$       |

$\text{IN}[B] = \cup(\text{out}[P])$  for all predecessor  
blocks in the CFG

$\text{OUT}[B] = \text{GEN}[B] + (\text{IN}[B] - \text{KILL}[B])$

# Forward vs. Backward

## Forward flow vs. Backward flow

Forward: Compute OUT for given IN, GEN, KILL

- Information propagates from the predecessors of a vertex.
- Examples: **Reachability**, available expressions, constant propagation

Backward: Compute IN for given OUT, GEN, KILL

- Information propagates from the successors of a vertex.
- Example: **Live variable Analysis**

# Forward vs. Backward Equations

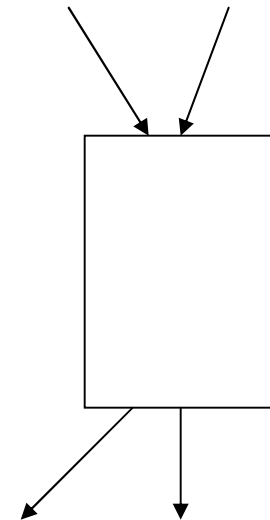
## Forward vs. backward

### – Forward:

- $IN[B]$  - process  $OUT[P]$  for all  $P$  in  $predecessors(B)$
- $OUT[B] = local\ U\ (IN[B] - local)$

### – Backward:

- $OUT[B]$  - process  $IN[S]$  for all  $S$  in  $successor(B)$
- $IN[B] = local\ U\ (OUT[B] - local)$



# May vs. Must

## May vs. Must

Must – true on **all paths**

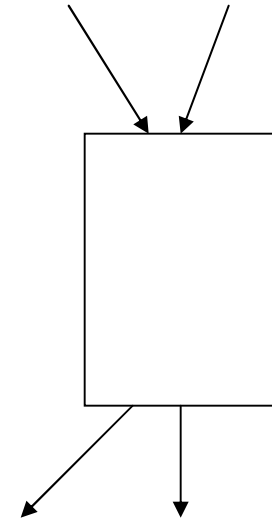
Ex: constant propagation – variable must provably hold appropriate constant on all paths in order to do a substitution

May – true on **some path**

Ex: **Live variable analysis** – a variable is live if it could be used on some path; **reachability** – a definition reaches a point if it can reach it on some path

# May vs. Must Equations

- May vs. Must
  - May –  $IN[B] = \cup(out[P])$  for all  $P$  in  $pred(B)$
  - Must –  $IN[B] = \cap(out[P])$  for all  $P$  in  $pred(B)$



- Reachability
  - $IN[B] = \cup(\text{out}[P])$  for all  $P$  in  $\text{pred}(B)$
  - $OUT[B] = \text{GEN}[B] + (IN[B] - \text{KILL}[B])$
- Live Variable Analysis
  - $OUT[B] = \cup(IN[S])$  for all  $S$  in  $\text{succ}(B)$
  - $IN[B] = \text{USE}[B] \cup (OUT[B] - \text{DEF}[B])$
- Constant Propagation
  - $IN[B] = \cap(\text{out}[P])$  for all  $P$  in  $\text{pred}(B)$
  - $OUT[B] = \text{DEF\_CONST}[B] \cup (IN[B] - \text{KILL\_CONST}[B])$

# Discussion

- Why does this work?
  - Finite set – can be represented as bit vectors
  - Theory of lattices
- Is this guaranteed to terminate?
  - Sets only grow and since finite in size ...
- Can we find ways to reduce the number of iterations?

# Choosing visit order for Dataflow Analysis

In forward flow analysis situations, if we visit the blocks in depth first order, we can reduce the number of iterations.

Suppose definition  $d$  follows block path  $3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$  where the block numbering corresponds to the preorder depth-first numbering.

Then we can compute the reach of this definition in 3 iterations of our algorithm.

