

Optimization

- **Intermediate representation**
 - 3 address code
- **Def-use information**
 - Reaching defs and live uses
 - Data-flow equations
 - Constant propagation
- **Common optimizations, by example**
- **Code motion and natural loops**

Three Address Code

- *Result, operand, operand, operator*
 - **$x := y \text{ op } z$** , where **op** is a binary operator and **x, y, z** can be variables, constants or compiler-generated temporaries (intermediate results)
- Can write this as a shorthand
 - $\langle op, arg1, arg2, result \rangle$ -- quadruples
 - Let line number of instruction stand for the result (saves space)
 - $\langle op, arg1, arg2 \rangle$ -- triples

Three Address Code

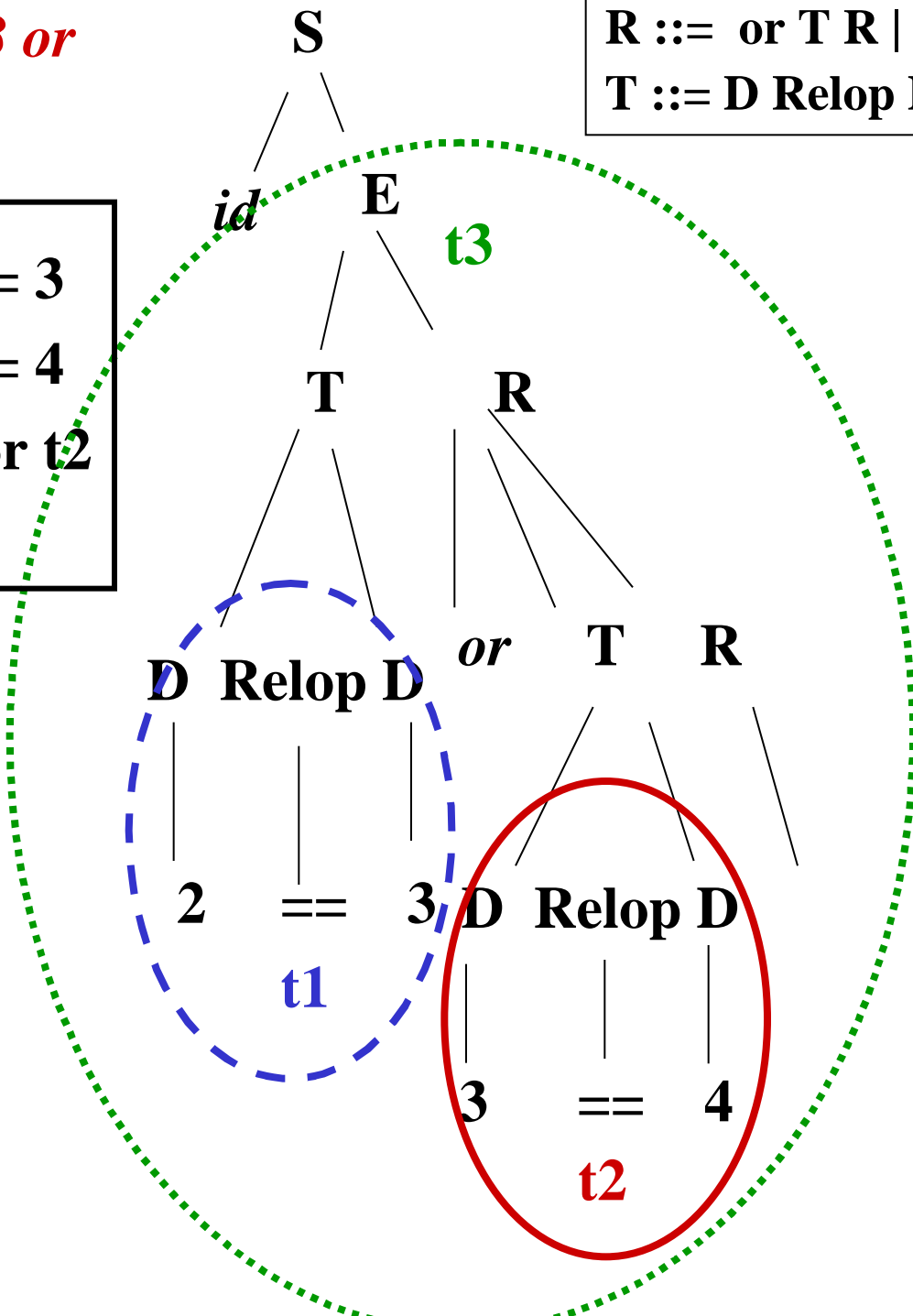
- **Set of statements allowed**
 - **Assignment** $x := y \text{ op } z$
 - **Copy stmts** $x := y$
 - **Goto L**
 - **if x relop y goto L**
 - **Indexed assignments** $x := y[j]$ or $s[j] := z$
 - **Address and pointer assignments (for C)**
 $x := \&y, x := *p; *x := y$
 - **Parm x; call p, n; return y;**

How this works?

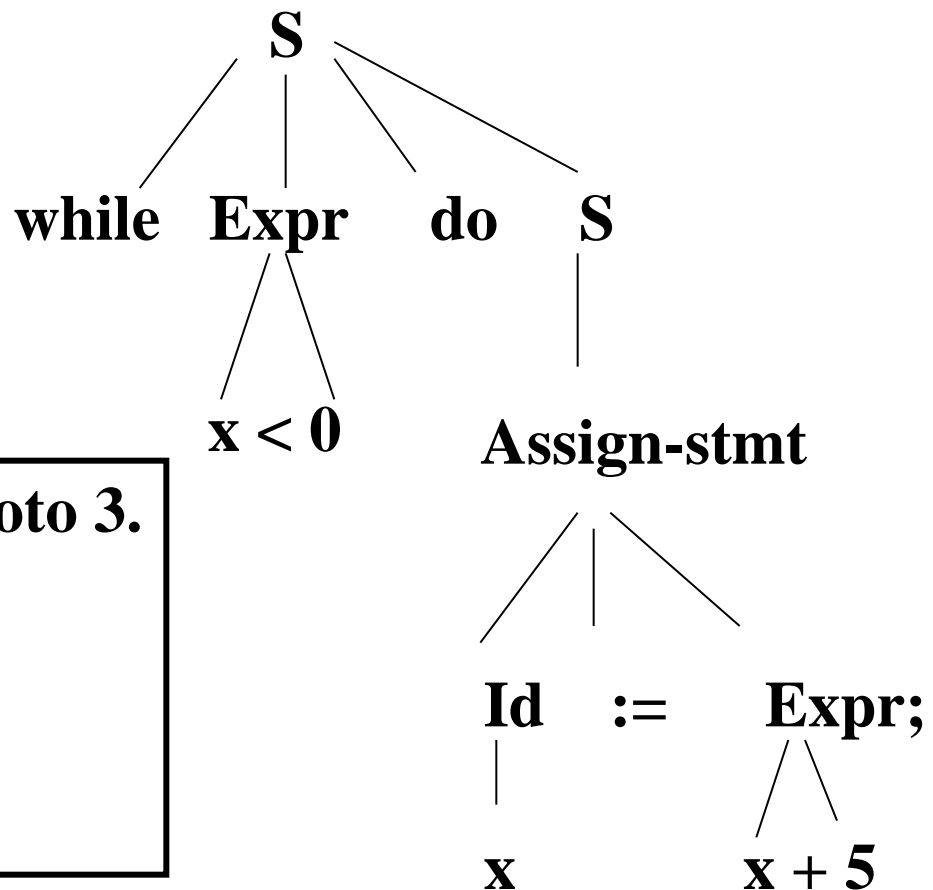
$S ::= id := E$
 $E ::= T R \mid R$
 $R ::= or \ T R \mid$
 $T ::= D \ Relop \ D$

$z := 2 == 3 \ or$
 $3 == 4;$

$t1 := 2 == 3$
 $t2 := 3 == 4$
 $t3 := t1 \ or \ t2$
 $z := t3$



E.g., While Stmt



1. if $x < 0$ goto 3.
2. goto 5.
3. $x := x + 5$
4. goto 1.
- 5.

Definitions

Flow analysis:

Fact finding about a program before its execution

Control-flow analysis:

Discerning possible execution paths.

Data-flow analysis:

Determining information about modification, preservation, and use of data entities in a program.

Two classic data-flow problems

Reaching definitions (REACH), Live uses of variables (LIVE)

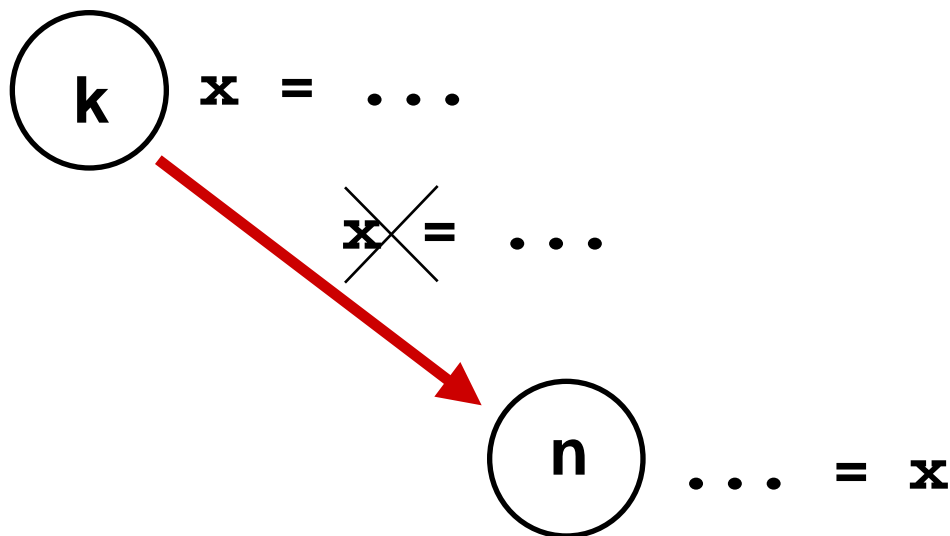
Def-use and Use-def chains, built from REACH and LIVE, used for many optimizations

Reaching Definitions (REACH)

Definition:

A statement that can modify the value of a variable.

A definition of a variable x at node k reaches node n if there is a definition-clear path from k to n .

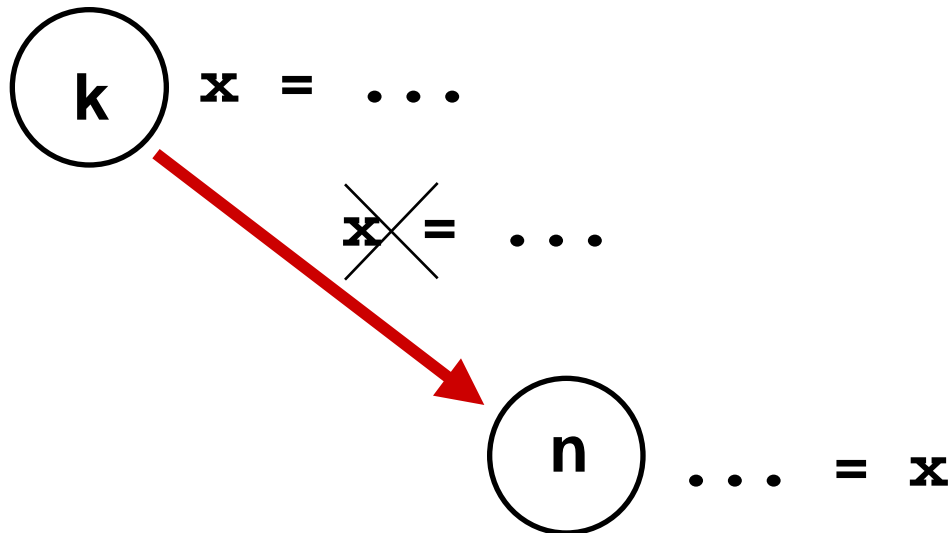


Live Uses of Variables (LIVE)

Use:

An appearance of a variable as an operand in 3 address code.

A use of a variable x at node n is live on exit from node k if there is a definition-clear path for x from k to n .



REACH and LIVE

UD- Chain:

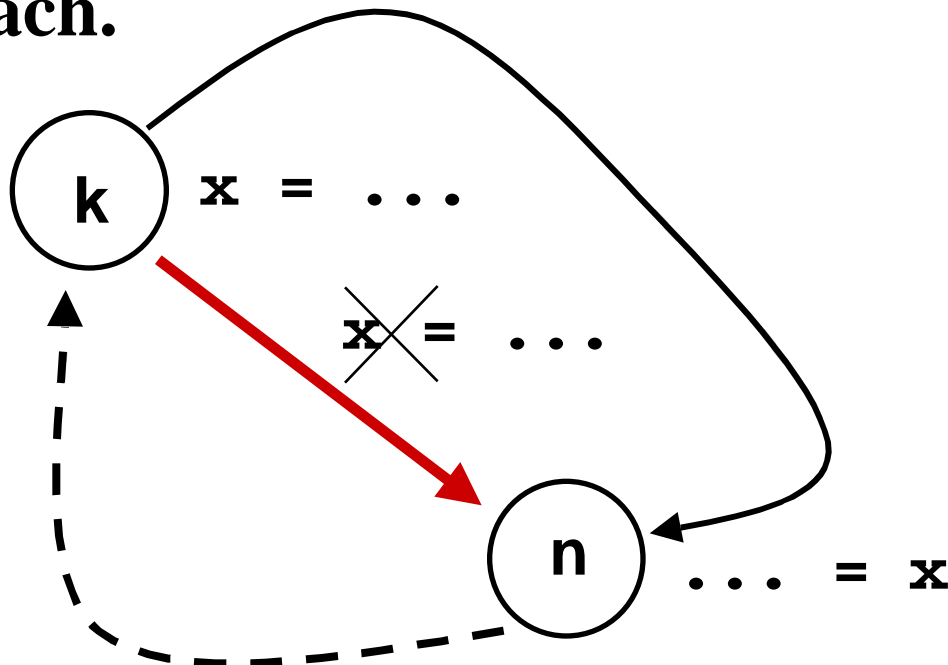


Links each use of variable x to definition(s) which reach that use.

DU - Chain:



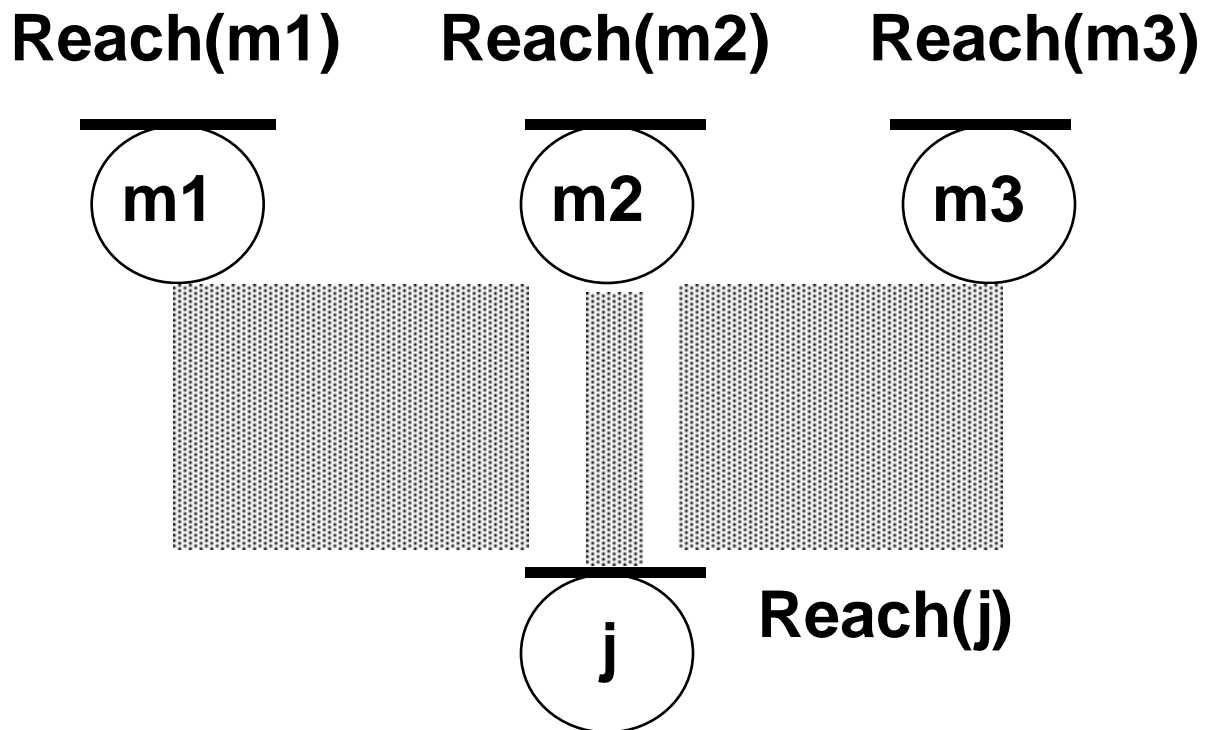
Links each definition of variable x to those uses which that definition can reach.



Global Optimizations

- **Live ranges for global register allocation(DU)**
- **Dead code elimination (DU)**
- **Code motion (UD)**
- **Strength reduction (UD)**
- **Test elision (UD)**
- **Constant propagation (UD)**
- **Copy propagation (DU)**

Reaching Definitions



forward
data-flow
problem

Data-Flow Equations

REACH

$$\text{Reach}(j) = \{ \text{Reach}(m) \text{ pres}(m) \text{ dgen}(m) \}$$

$m \text{ pred}(j)$

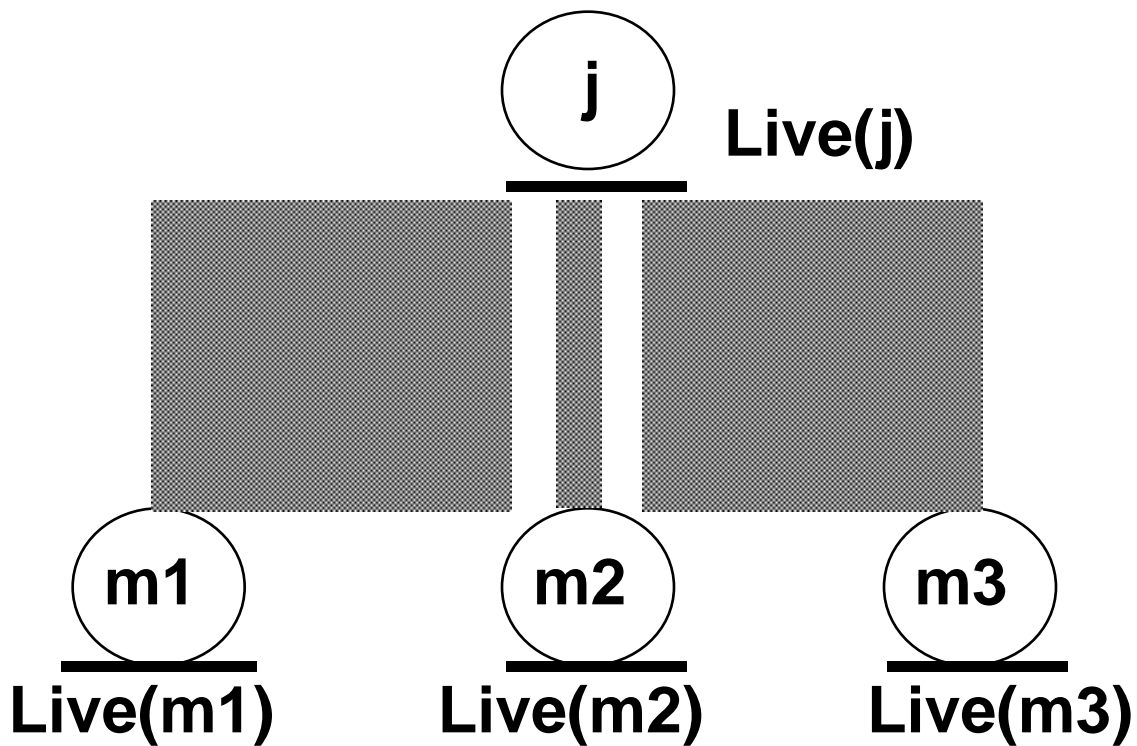
where:

pres(m) is the set of defs preserved through node m

dgen(m) is the set of defs generated at node m

pred(j) is the set of immediate predecessors of node j

Live Uses of Variables



backward
data-flow
problem

Data-Flow Equations

LIVE

Live(j) =

$$\{ \text{Live}(m) \cup \text{upres}(m) \cup \text{ugen}(m) \}$$

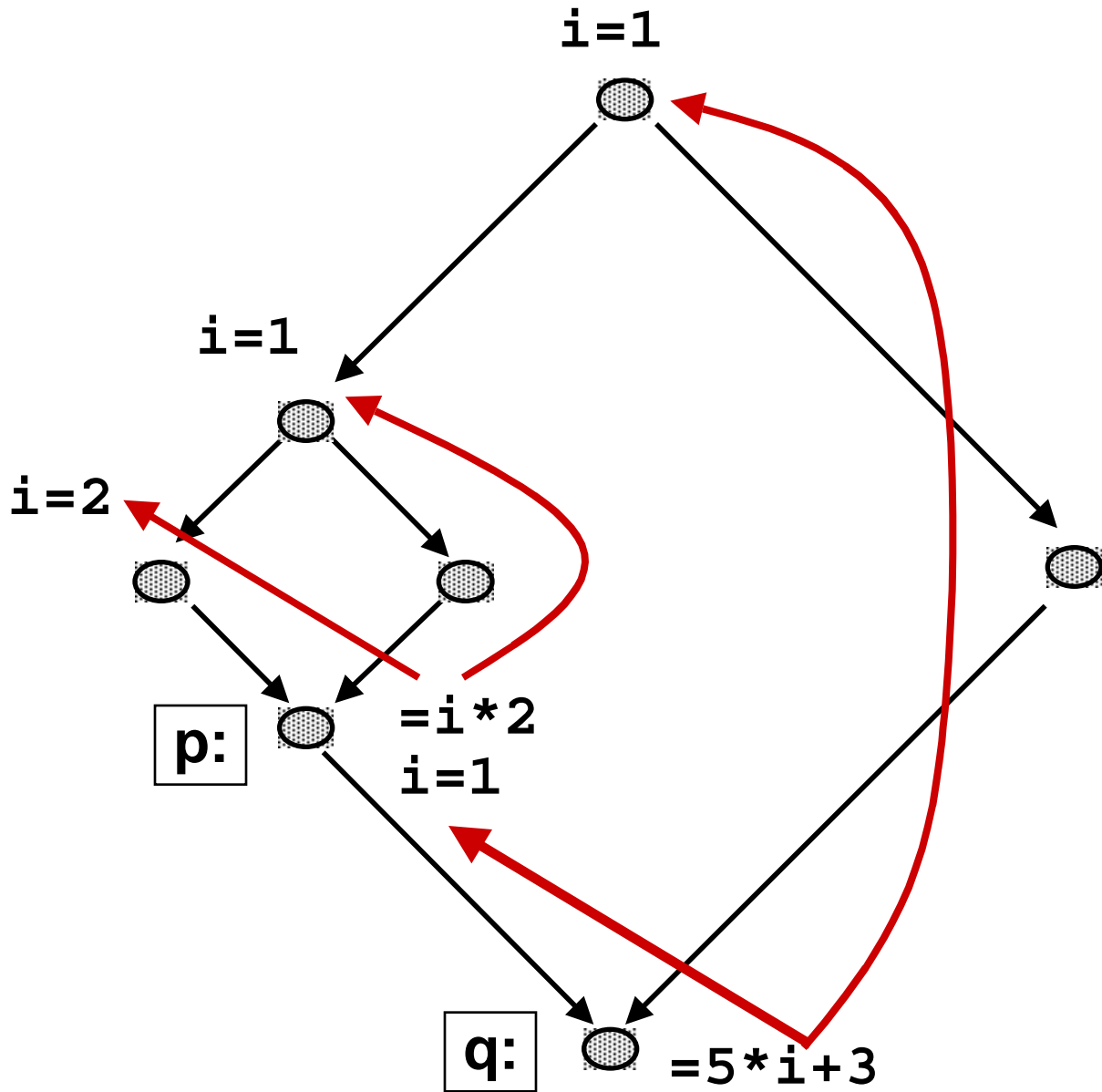
m succ(j)
where:

upres(m) is the set of uses preserved through node m

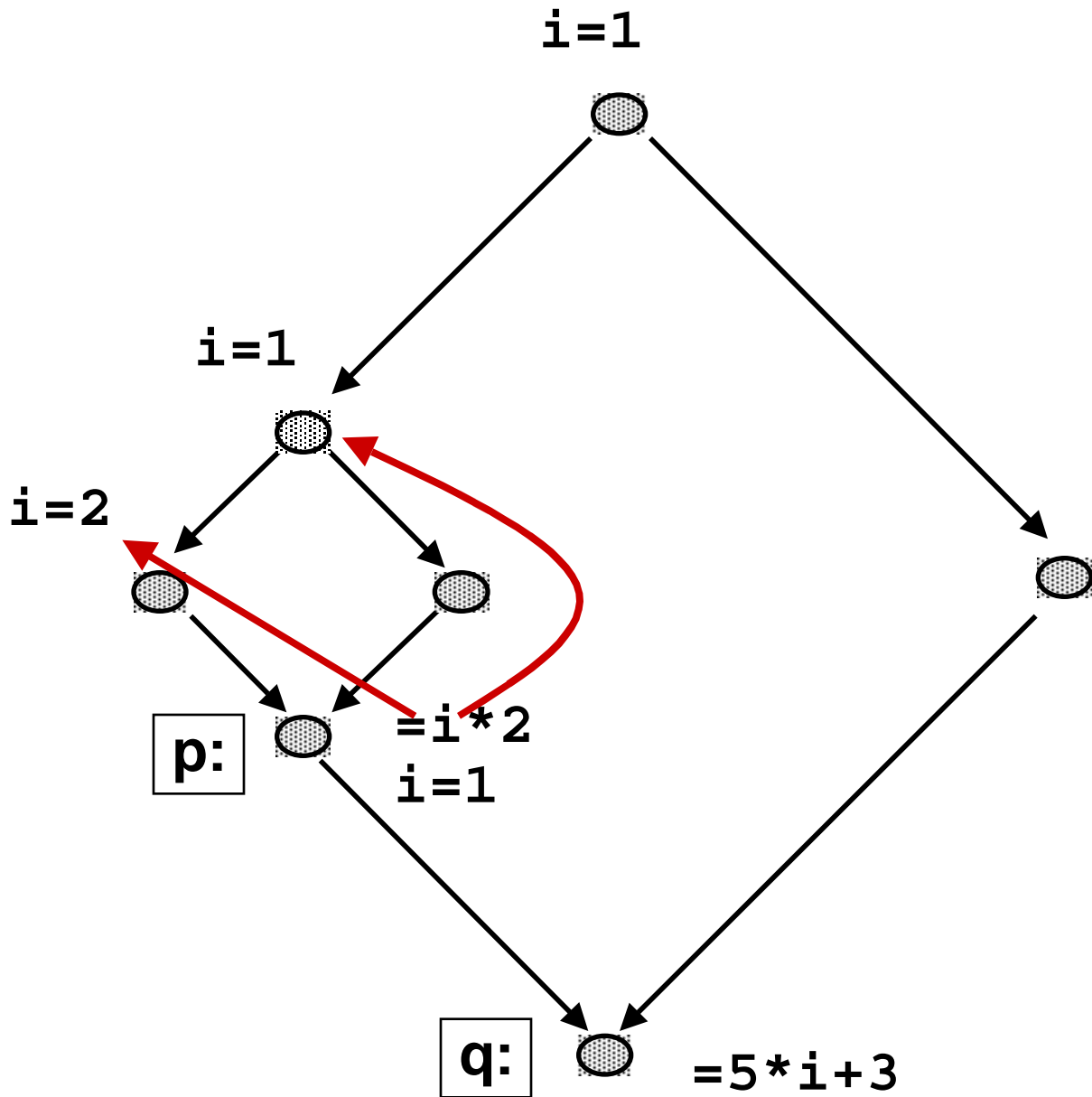
ugen(m) is the set of uses generated at node m

succ(j) is the set of immediate successors of node j

Constant Propagation

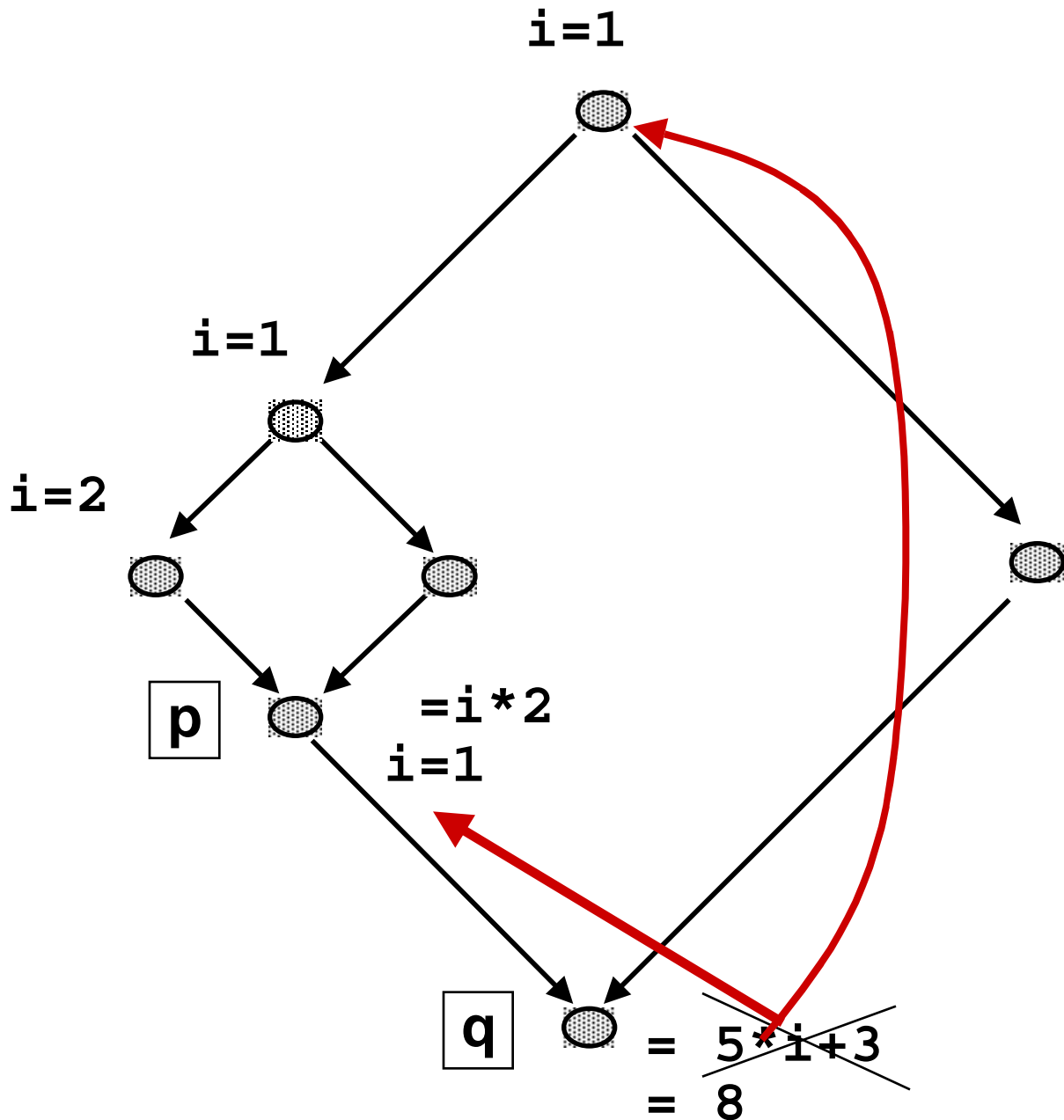


Constant Propagation



At program point p , UD chain shows all definitions reaching this use are constant - but not the same constant.
No propagation.

Constant Propagation



At program point q , UD chain shows all defs reaching this use are constant - and the same constant; can substitute value.

***Optimization is
a semantics
preserving
operation***

Example

Fortran Source Code:

```
•  
•  
•  
sum = 0  
do 10 i = 1, n  
10 sum = sum + a(i) *  
a(i)  
•  
•  
•
```

3 Address Code

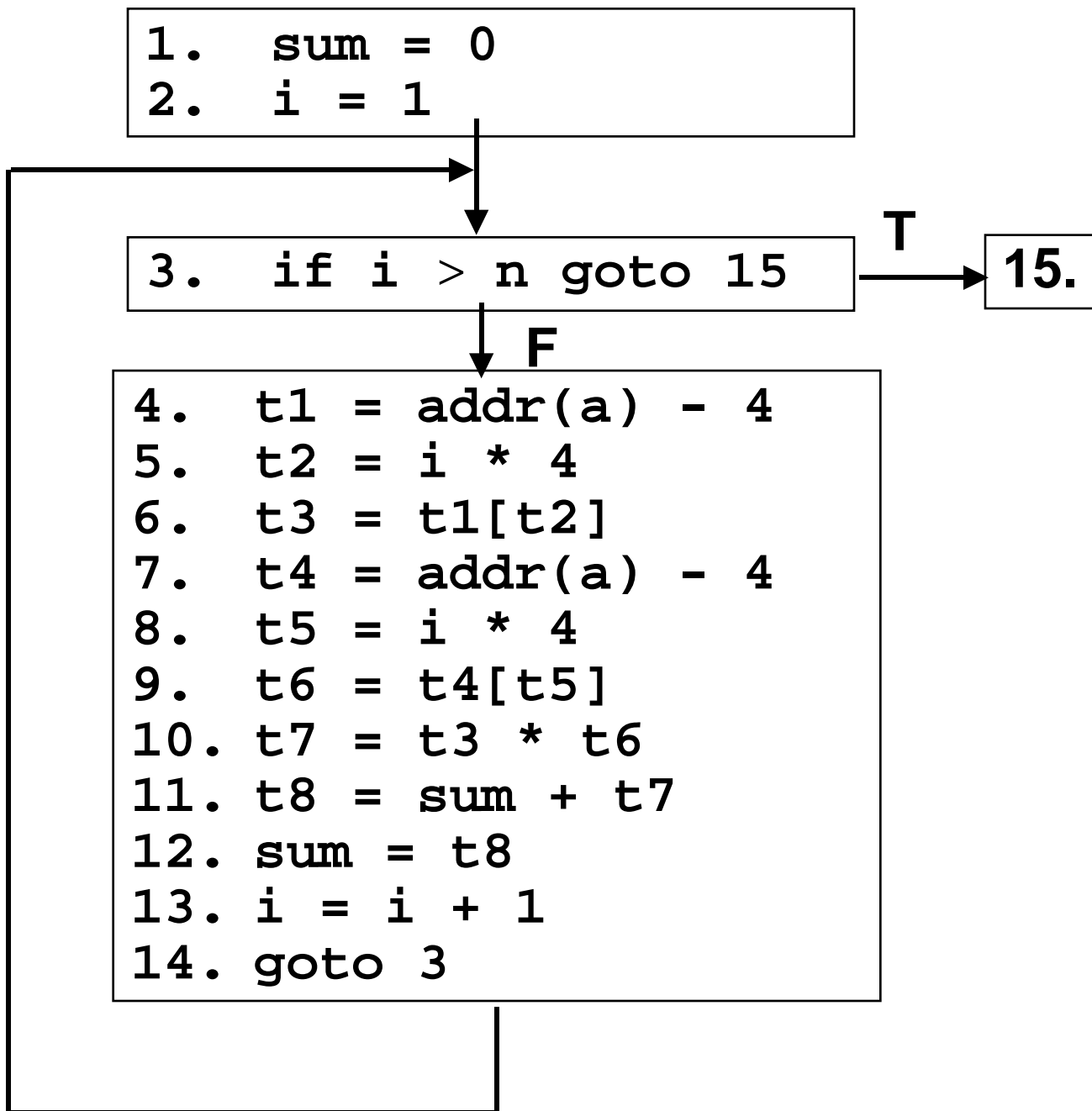
```
1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
```

Optimization

<u>3-Address Code</u>	<u>Source</u>
1. <code>sum = 0</code>	<code>sum = 0</code>
2. <code>i = 1</code>	<code>init for loop</code>
3. <code>if i > n goto 15</code>	<code>and check limit</code>
4. <code>t1 = addr(a) - 4</code>	<code>a[i]</code>
5. <code>t2 = i * 4</code>	
6. <code>t3 = t1[t2]</code>	
7. <code>t4 = addr(a) - 4</code>	<code>a[i]</code>
8. <code>t5 = i * 4</code>	
9. <code>t6 = t4[t5]</code>	
10. <code>t7 = t3 * t6</code>	<code>a[i] * a[i]</code>
11. <code>t8 = sum + t7</code>	<code>increment sum</code>
12. <code>sum = t8</code>	
13. <code>i = i + 1</code>	<code>increment loop counter</code>
14. <code>goto 3</code>	
15.	

•

Control Flow Graph



Example

```
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 = t4[t5]
10.   t7 = t3 * t6
10a.  t7 = t3 * t3
11.   t8 = sum + t7
11a.  sum = sum + t7
12.   sum = t8
13.   i = i + 1
14.   goto 3
15.
```

Local Common Subexpression Elimination

Example

1. sum = 0

2. i = 1

2a. t1 = addr[a] - 4

3. if i > n goto 15

~~4. t1 = addr(a) - 4~~

5. t2 = i * 4

6. t3 = t1[t2]

10a. t7 = t3 * t3

11a. sum = sum + t7

13. i = i + 1

14. goto 3

15.

Invariant Code Motion

Example

1. sum = 0

2. i = 1

2a. t1 = addr[a] - 4

2b. t2 = i * 4

3. if i > n goto 15

~~5. t2 = i * 4~~

6. t3 = t1[t2]

10a. t7 = t3 * t3

11a. sum = sum + t7

12a. t2 = t2 + 4

13. i = i + 1

14. goto 3

15.

Reduction in Strength

Example

```
1.  sum = 0
2.  i = 1
2a. t1 = addr[a] - 4
2b. t2 = i * 4
2c. t9 = 4 * n
3.  if i > n goto 15
3a. if t2 > t9 goto 15
6.  t3 = t1[t2]
10a. t7 = t3 * t3
11a. sum = sum + t7
12a. t2 = t2 + 4
13. i = i + 1
14. goto 3a
```

15. **Test Elision
and
Elimination of Induction Variables**

Example

1. sum = 0

~~2. i = 1~~

2a. t1 = addr[a] - 4

~~2b. t2 = i * 4~~

~~2d. t2 = 4~~

2c. t9 = 4 * n

3a. if t2 > t9 goto 15

6. t3 = t1[t2]

10a. t7 = t3 * t3

11a. sum = sum + t7

12a. t2 = t2 + 4

14. goto 3a

15. **Constant Propagation
and
Dead Code Elimination**

Example

Optimized Code (renumbered)

```
1.  sum = 0
2.  t1 = addr[a] - 4
3.  t2 = 4
4.  t9 = 4 * n
5.  if t2 > t9 goto 11
6.  t3 = t1[t2]
7.  t7 = t3 * t3
8.  sum = sum + t7
9.  t2 = t2 + 4
10. goto 5
11.
```

<p>Unoptimized: 8 temps, 11 stmts in loop</p> <p>Optimized: 5 temps, 5 stmts in loop</p>
--

Example

```
1.  sum = 0
2.  t1 = addr[a] - 4
3.  t2 = 4
4.  t9 = 4 * n
```

F

```
5.  if t2 > t9 goto 11
```

T

```
11.
```

```
6.  t3 = t1[t2]
7.  t7 = t3 * t3
8.  sum = sum + t7
9.  t2 = t2 + 4
10. goto 5
```

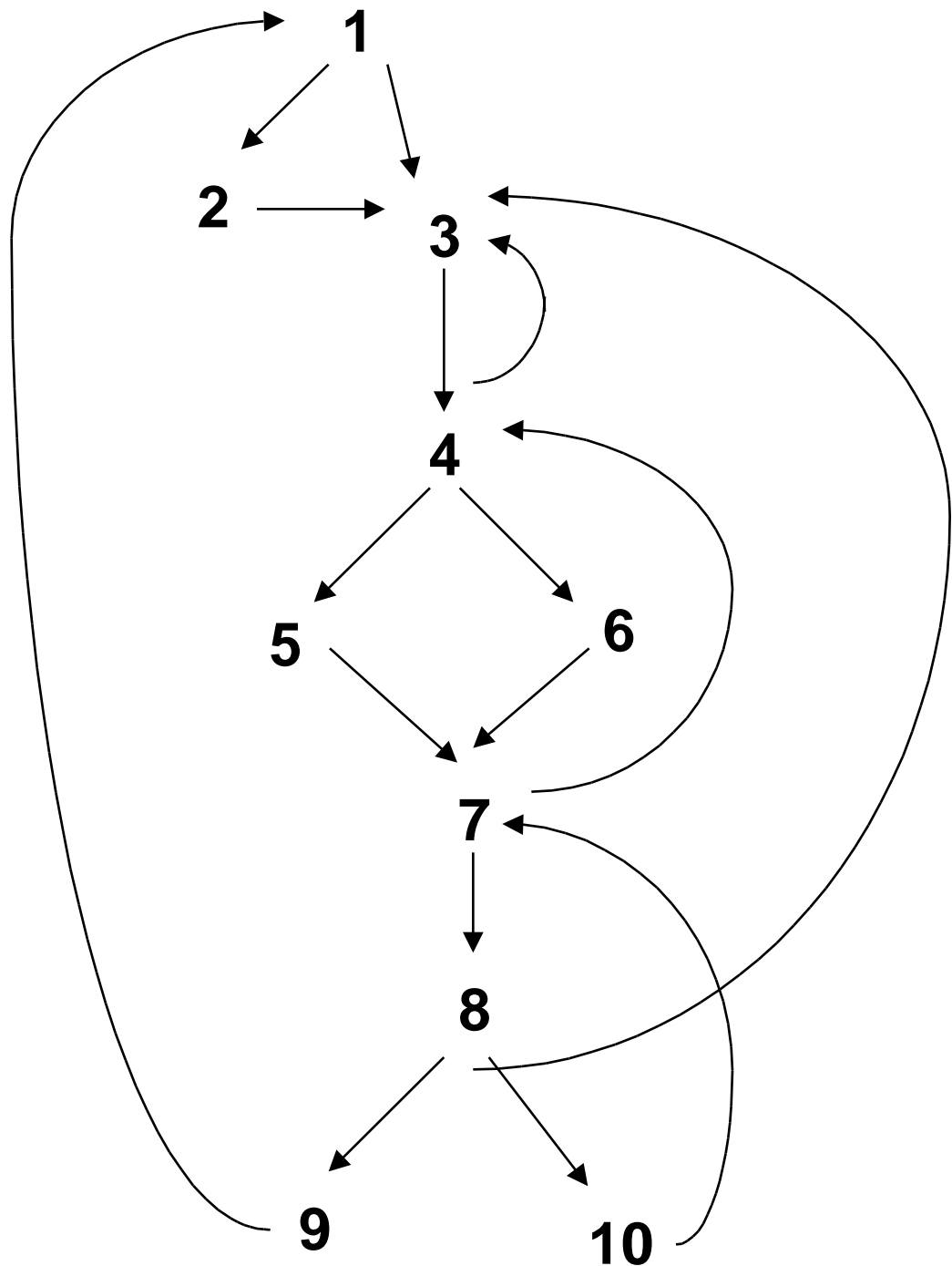
Natural Loops

- **Single entry node (header) that dominates all other nodes in loop**
- **From every node there is at least one path back to the header**
- **Back edge: edge whose target node dominates its source node.**

Natural Loops

- **Loop construction:**
 - **Find back edge. Traverse edges in reverse execution direction until back edge target is reached. All nodes encountered in traversal are in corresponding natural loop. (ASU Alg 10.1)**
- **If 2 back edges go to same header then all nodes in natural loop sets for these edges are in same loop**
- **If each pair of nodes, one in $\text{loop}(k)$ and one in $\text{loop}(n)$ are reachable one from the other and $\text{header}(n)$ dominates $\text{header}(k)$, then $\text{loop}(k)$ is nested within $\text{loop}(n)$.**

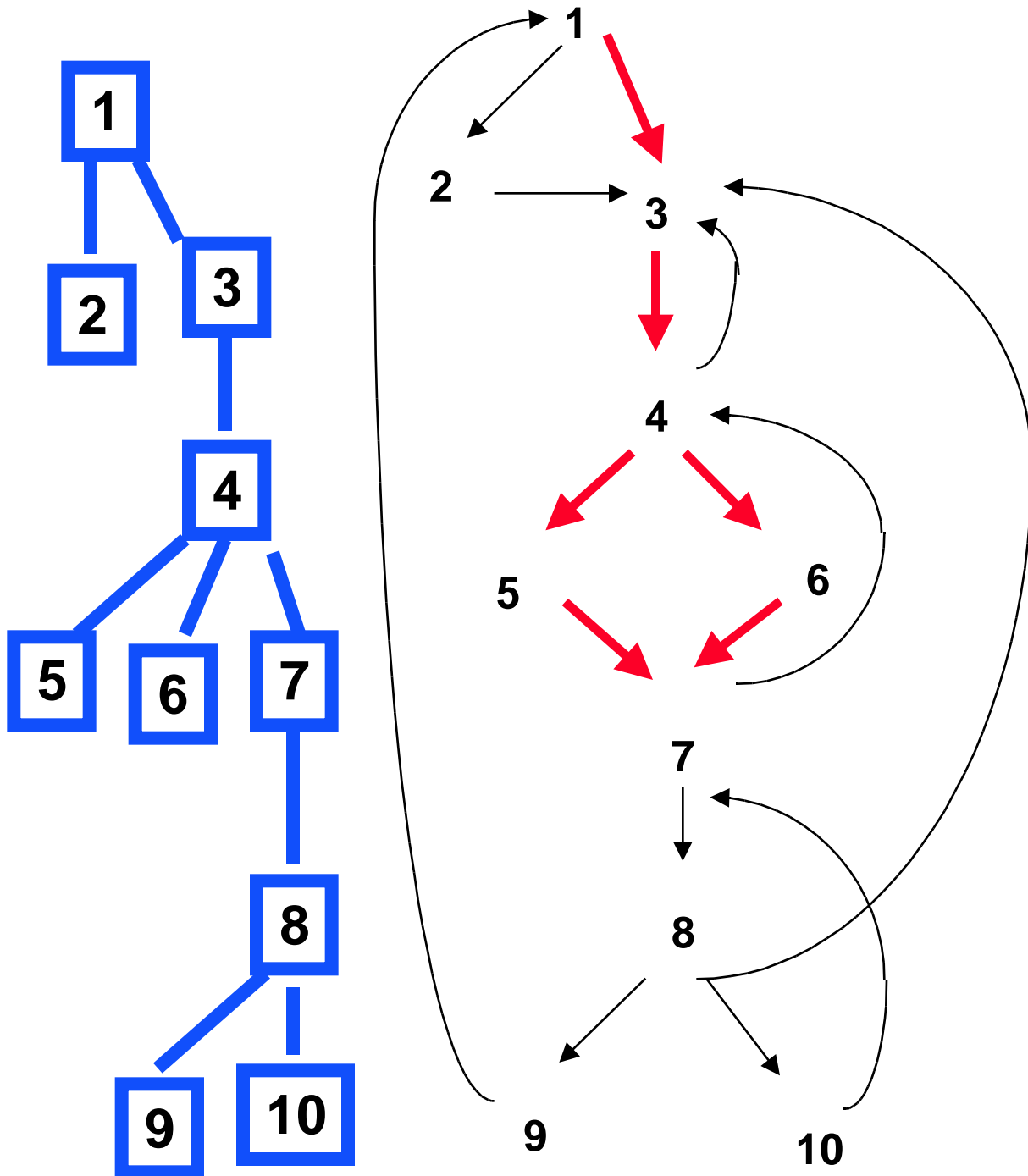
Natural Loops



ASU, P 603

Natural Loops

1 dominates 7



Dominator Tree

Invariant Code Motion

- Computation is **loop invariant** if its value does not change while control stays within the loop
- Algm (needs use-def chains)
 - As move invariants, other code becomes invariant.

Invariant Code Motion

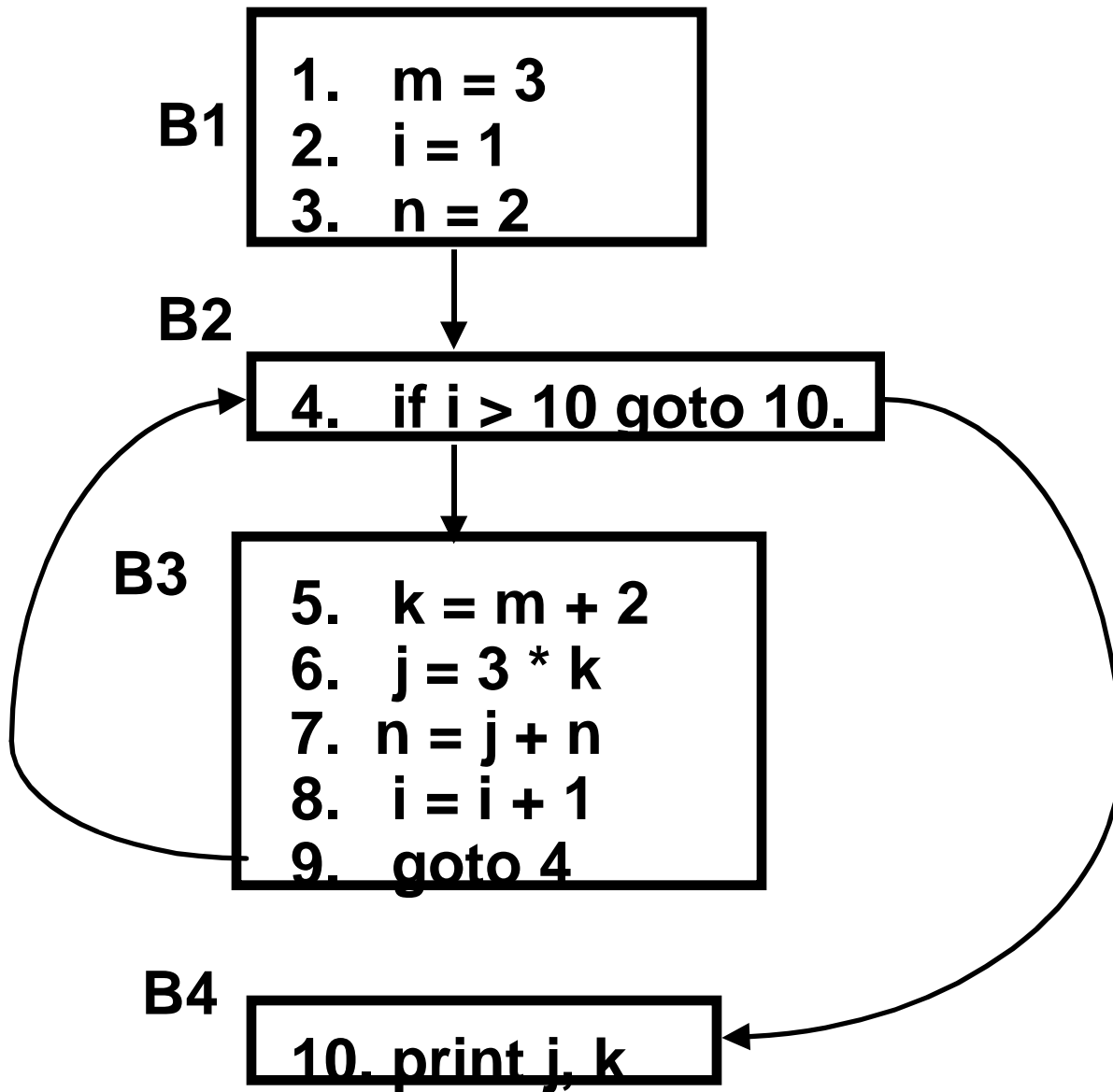
1. $m = 3$
2. $i = 1$
3. $n = 2$

4. if $i > 10$ goto 10.

5. $k = m + 2$
6. $j = 3 * k$
7. $n = j + n$
8. $i = i + 1$
9. goto 4

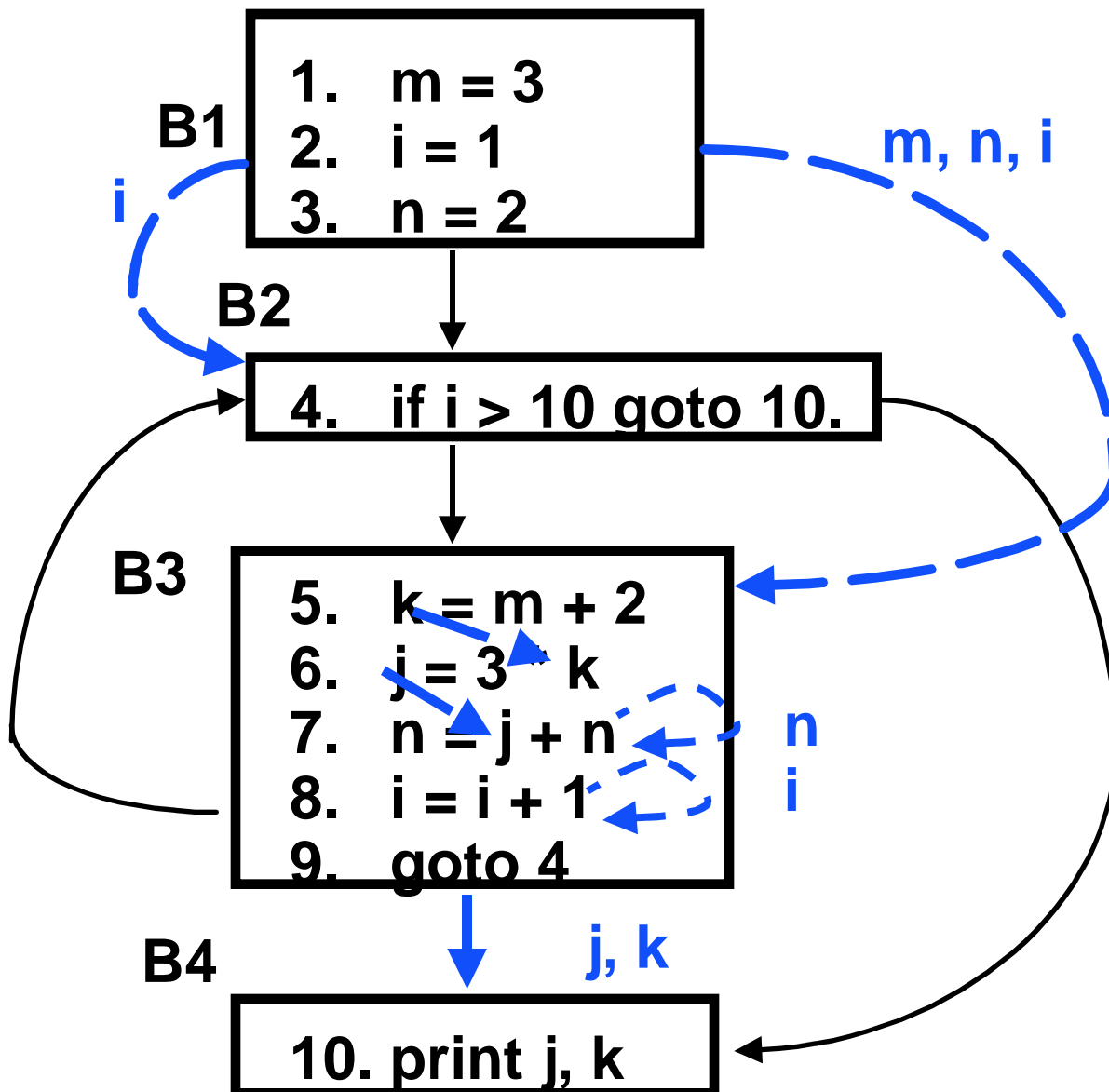
10. print j, k

Invariant Code Motion



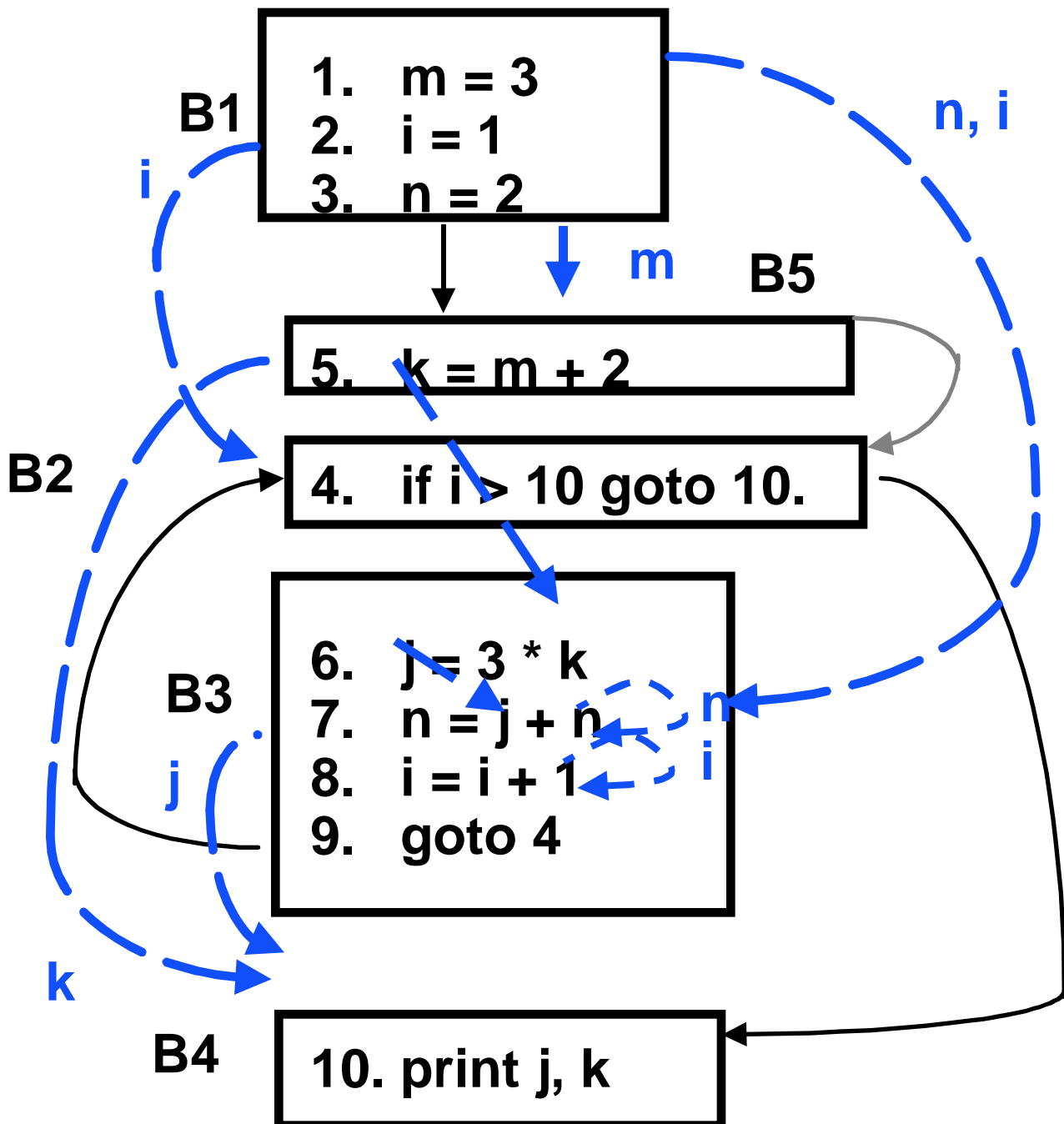
Control Flow Graph

Def-Use Links



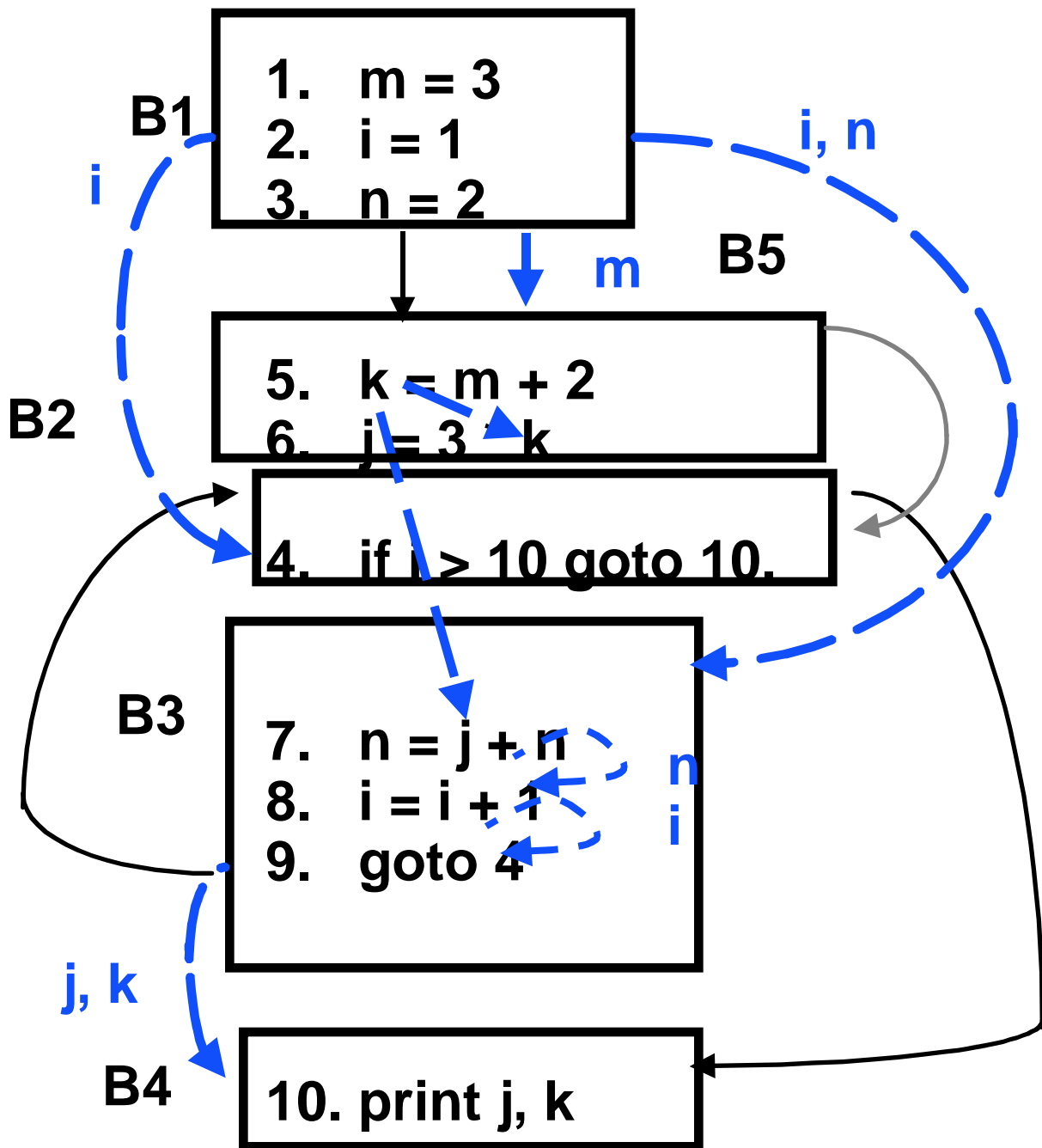
Def-Use Links - invariant code is code which has no def-use links from within the loop (B2,B3)

Invariant Code Motion



$m+2$ is invariant so statement 5 can be moved to new cfg node (loop preheader) B5.
now statement 6 can be moved to B5 as well, because it is now invariant in the loop.

Invariant Code Motion



No more code motion is possible because neither n nor i are invariant in the loop.