Compilerconstructie

najaar 2017

`http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/`

Rudy van Vliet
kamer 140 Snellius, tel. 071-527 2876
rvvliet(at)liacs(dot)nl

college 9, vrijdag 24 november 2017
+ 'werkcollege'

Code Optimization (1)

8.5 Optimization of Basic Blocks

To improve running time of code

- Local optimization: within block
- Global optimization: across blocks

Local optimization benefits from DAG representation of basic block

A slide from lecture 5

6.1 Variants of Syntax Trees

Directed Acyclic Graphs for Expressions

$a + a * (b - c) + (b - c) * d$

Syntax tree vs DAG. . .

Pros DAG. . .

8.5.1 DAG Representation of Basic Blocks

1. A node for initial value of each variable appearing in block
2. A node N for each statement s in block
   Children of N are nodes corresponding to last definitions of operands used by s
3. Node N is labeled by operator applied at s
   N has list of variables for which s is last definition in block
4. *Output nodes* $\approx$ live on exit

Example:

```
a = b + c
b = a - d
c = b + c
d = a - d
```

8.5.2 Finding Local Common Subexpressions

- Use value-number method to detect common subexpressions

- Remove redundant computations

Example:

```
a = b + c
b = a - d
c = b + c
d = a - d
```

Local Common Subexpression Elimination

- Use value-number method to detect common subexpressions

- Remove redundant computations

Example:

```
a = b + c          a = b + c
b = a - d          b = a - d
c = b + c          c = b + c
d = a - d          d = b
```

8.5.3 Dead Code Elimination

- Remove roots with no live variables attached
- If possible, repeat

Example:

```
a = b + c
b = b - d
c = c + d
e = b + c
```

No common subexpression

If c and e are not live. . .

Dead Code Elimination

- Remove roots with no live variables attached
- If possible, repeat

Example:

```
a = b + c          a = b + c
b = b - d          b = b - d
c = c + d
e = b + c
```

No common subexpression

If c and e are not live. . .

8.5.5 Representation of Array References

```
x = a[i]
y = x+z
z = a[i]
```

DAG. . .

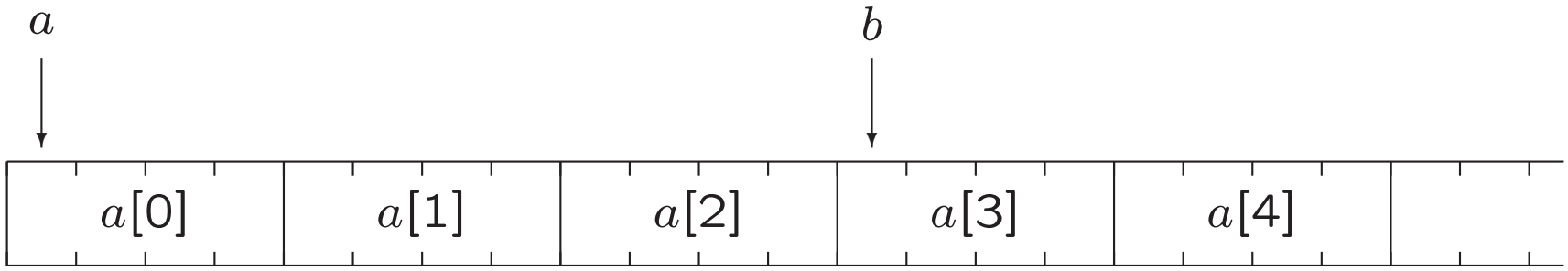
Representation of Array References

```
x = a[i]
a[j] = y
z = a[i]
```

DAG. . .

Representation of Array References

```
b = 12 + a
x = b[i]
a[j] = y
z = b[i]
```



DAG. . .

8.5.6 Pointer Assignments and Procedure Calls

```
a = b + c
e = a - d
c = b + c
b = a - d
```

DAG. . .

Pointer Assignments vs. Common Subexpressions

```
p = &a
a = b + c
e = a - d
*p = y
c = b + c
b = a - d
```

DAG. . .

Pointer Assignments vs. Common Subexpressions

```
a = b + c
e = a - d
*p = y
c = b + c
b = a - d
```

DAG. . .

Pointer Assignments vs. Dead Code

```
a = b + c
b = b - d
c = c + d
e = b + c
x = *p
```

DAG. . .

If c and e are not live. . .

To summarize:

```
*q = y
x = *p
```

Procedure calls. . .

8.5.4 The Use of Algebraic Identities

and other algebraic transformations

(cf. assignment 3)

Algebraic identities:

$$\begin{aligned} x + 0 &= 0 + x &&= x \\ x * 1 &= 1 * x &&= x \end{aligned}$$

Reduction in strength:

$$\begin{aligned} x^2 &= x * x && \text{(cheaper)} \\ 2 * x &= x + x && \text{(cheaper)} \\ x/2 &= x * 0.5 && \text{(cheaper)} \end{aligned}$$

Constant folding:

$$2 * 3.14 = 6.28$$

Algebraic Transformations

Common subexpressions resulting from commutativity / associativity of operators:

$$\begin{aligned} x * y &= y * x \\ c + d + b &= (b + c) + d \end{aligned}$$

Common subexpressions generated by relational operators:

$$x > y \;\Leftrightarrow\; x - y > 0$$

8.7 Peephole Optimization

- Examines short sequence of instructions in a window (peephole) and replace them by faster/shorter sequence
- Applied to intermediate code or target code
- Typical optimizations
  - Redundant instruction elimination
  - Eliminating unreachable code
  - Flow-of-control optimization
  - Algebraic simplification
  - Use of machine idioms

8.7.1 Eliminating Redundant Loads and Stores

Naive code generator may produce

```
ST  a, R0
LD  R0, a
```

N.B.: optimize only within basic block

8.7.2 Eliminating Unreachable Code

Example:

```
      if debug == 1 goto L1
      goto L2
L1:   print debugging information
L2:
```

Jump over jump

Eliminating Unreachable Code

Example:

```
      if debug != 1 goto L2
      print debugging information
  L2:
```

How to recognize that label `L1` can be removed?

Eliminating Unreachable Code

Example:

```
      if debug != 1 goto L2
      print debugging information
  L2:
```

If `debug` is set to 0 at beginning of program, . . .

8.7.2 Eliminating Unreachable Code

Example:

```
      if debug == 1 goto L1
      goto L2
L1:   print debugging information
L2:
```

Even without jump over jump. . .

8.7.3 Flow-of-Control Optimizations

Example 1:

```
      goto L1
      ...
L1:   goto L2
```

Example 3:

```
      goto L1
      . . .
L1:   if a < b goto L2
L3:
```

8.7.3 Flow-of-Control Optimizations

Example 1:

```
     goto L1
     ...
L1:  goto L2
```

```
     goto L2
     ...
L1:  goto L2
```

Example 3:

```
     goto L1
     . . .
L1:  if a < b goto L2
L3:
```

```
     if a < b goto L2
     goto L3
     ...
L3:
```

9.1 The Principal Sources of Optimization

Causes of redundancy

- At source level
- Side effect of high-level programming language, e.g., $A[i][j]$

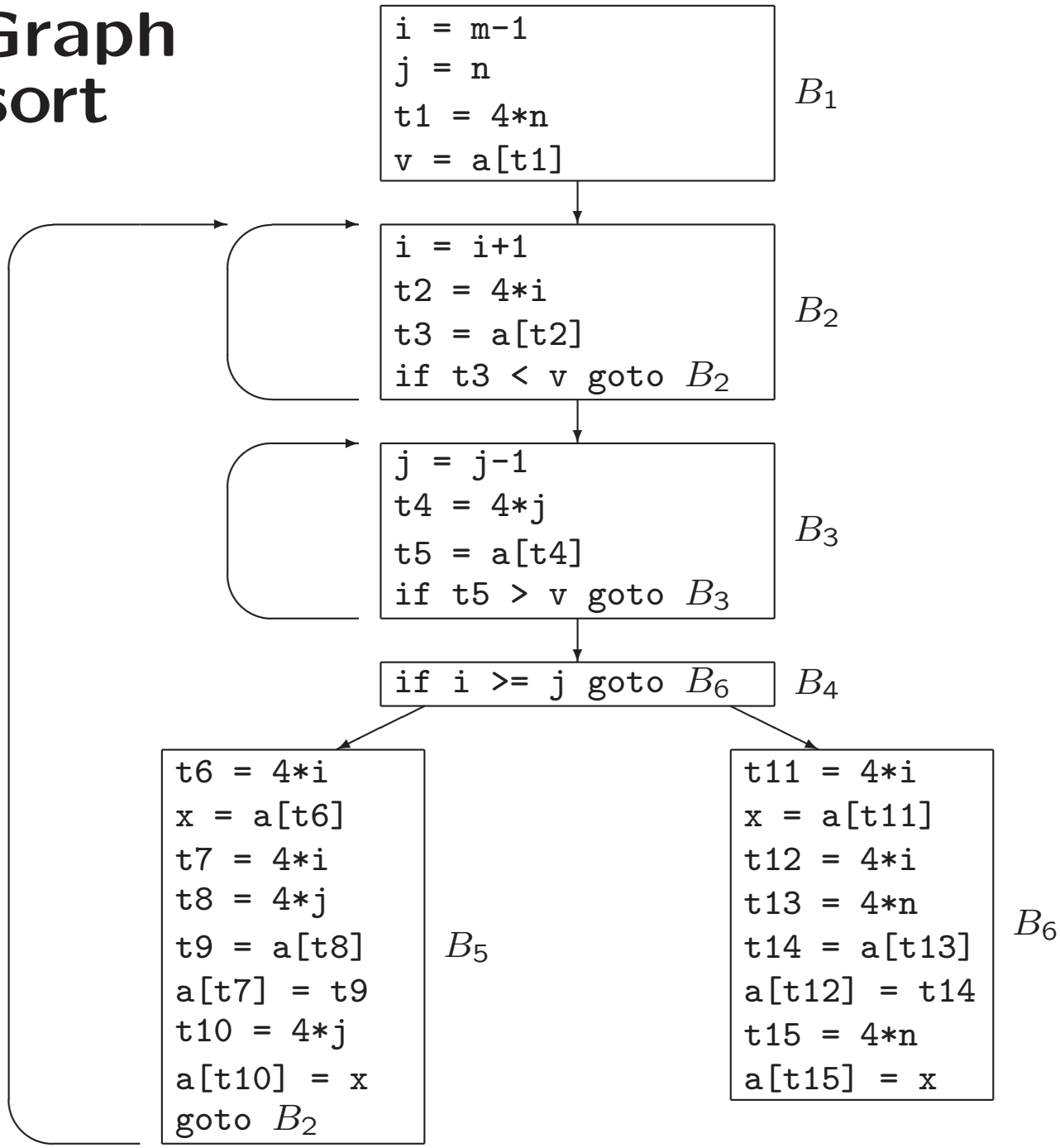
9.1.2 A Running Example: Quicksort

```
void quicksort (int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;

    if (n <= m) return;

    i = m-1; j = n; v = a[n];
    while (1)
    {   do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */

    quicksort(m,j); quicksort(i+1,n);
}
```

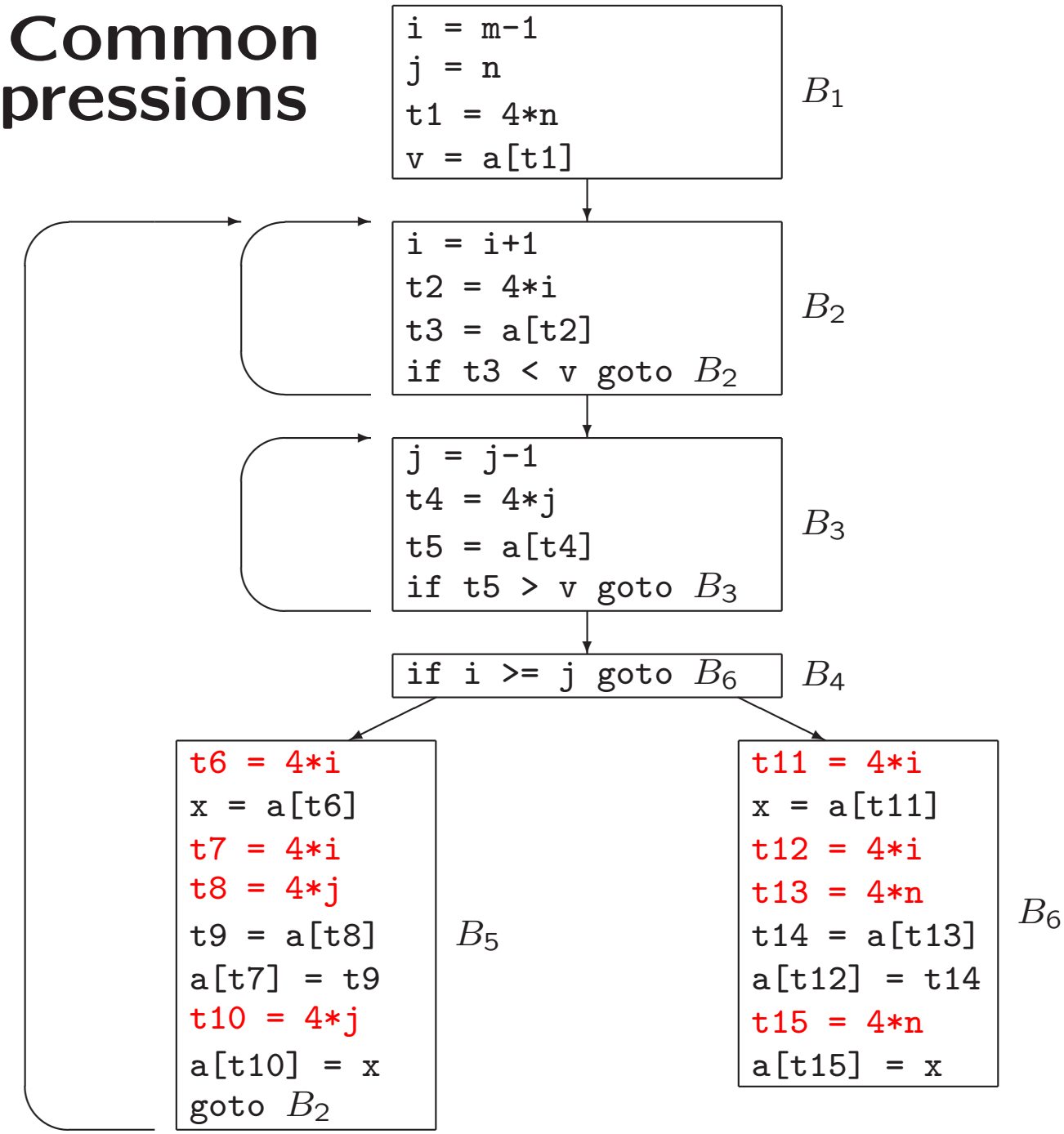
Three-Address Code Quicksort

```
⟶   (1)   i = m-1
     (2)   j = n
     (3)   t1 = 4*n
     (4)   v = a[t1]
⟶   (5)   i = i+1
     (6)   t2 = 4*i
     (7)   t3 = a[t2]
     (8)   if t3<v goto (5)
⟶   (9)   j = j-1
    (10)   t4 = 4*j
    (11)   t5 = a[t4]
    (12)   if t5>v goto (9)
⟶  (13)   if i>=j goto (23)
⟶  (14)   t6 = 4*i
    (15)   x = a[t6]
    (16)   t7 = 4*i
    (17)   t8 = 4*j
    (18)   t9 = a[t8]
    (19)   a[t7] = t9
    (20)   t10 = 4*j
    (21)   a[t10] = x
    (22)   goto (5)
⟶  (23)   t11 = 4*i
    (24)   x = a[t11]
    (25)   t12 = 4*i
    (26)   t13 = 4*n
    (27)   t14 = a[t13]
    (28)   a[t12] = t14
    (29)   t15 = 4*n
    (30)   a[t15] = x
```

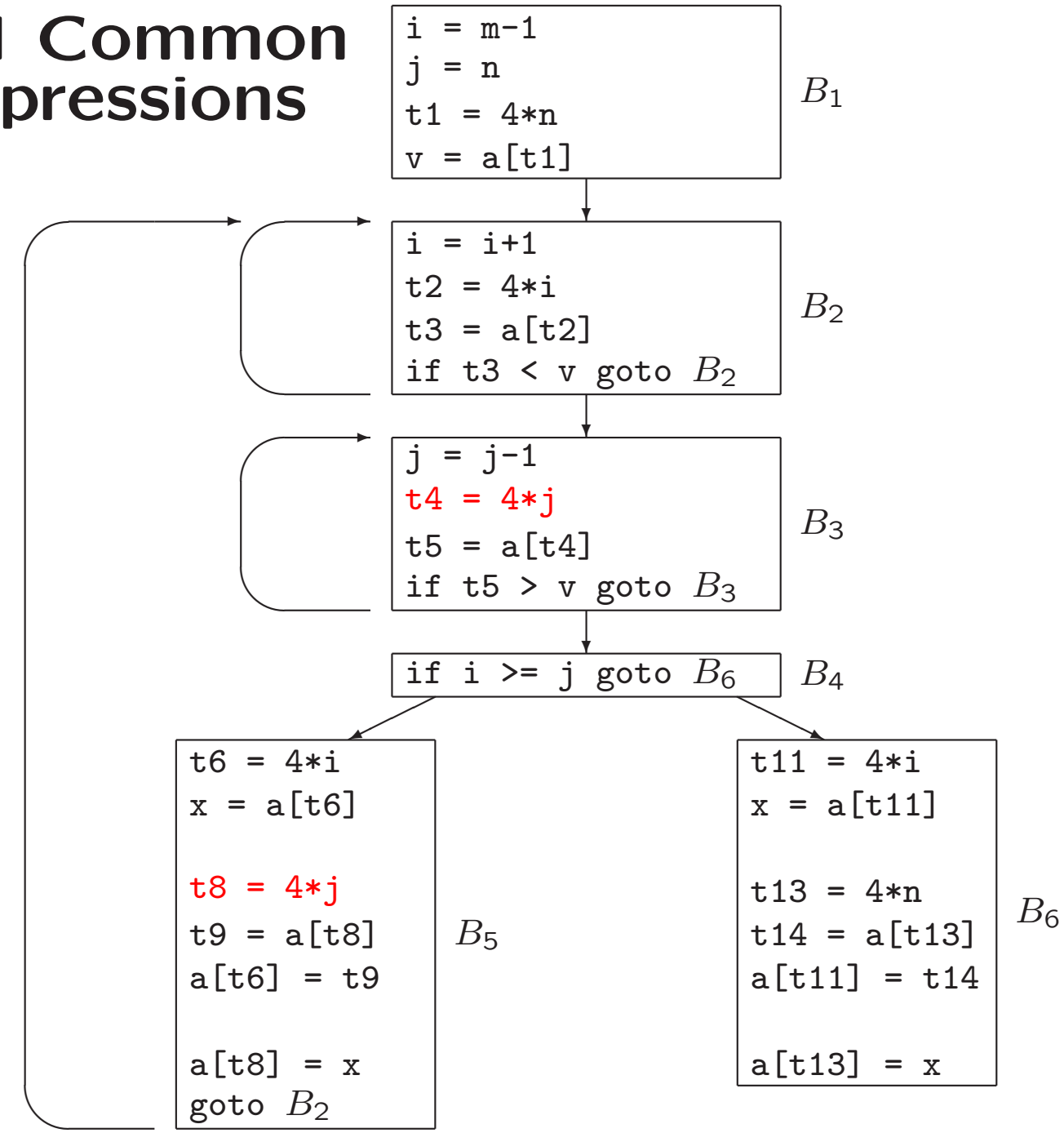
Flow Graph Quicksort



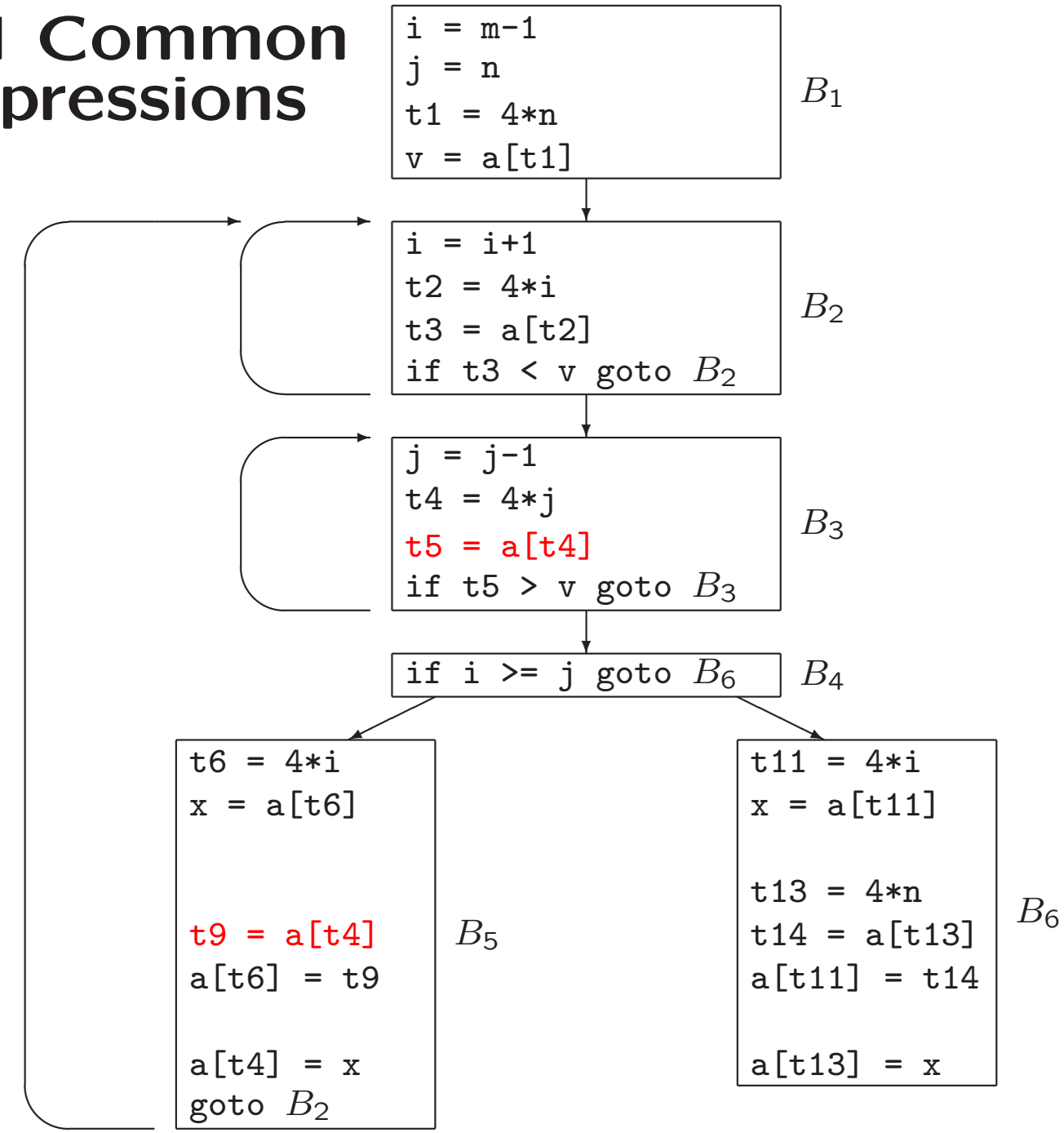
Local Common Subexpressions



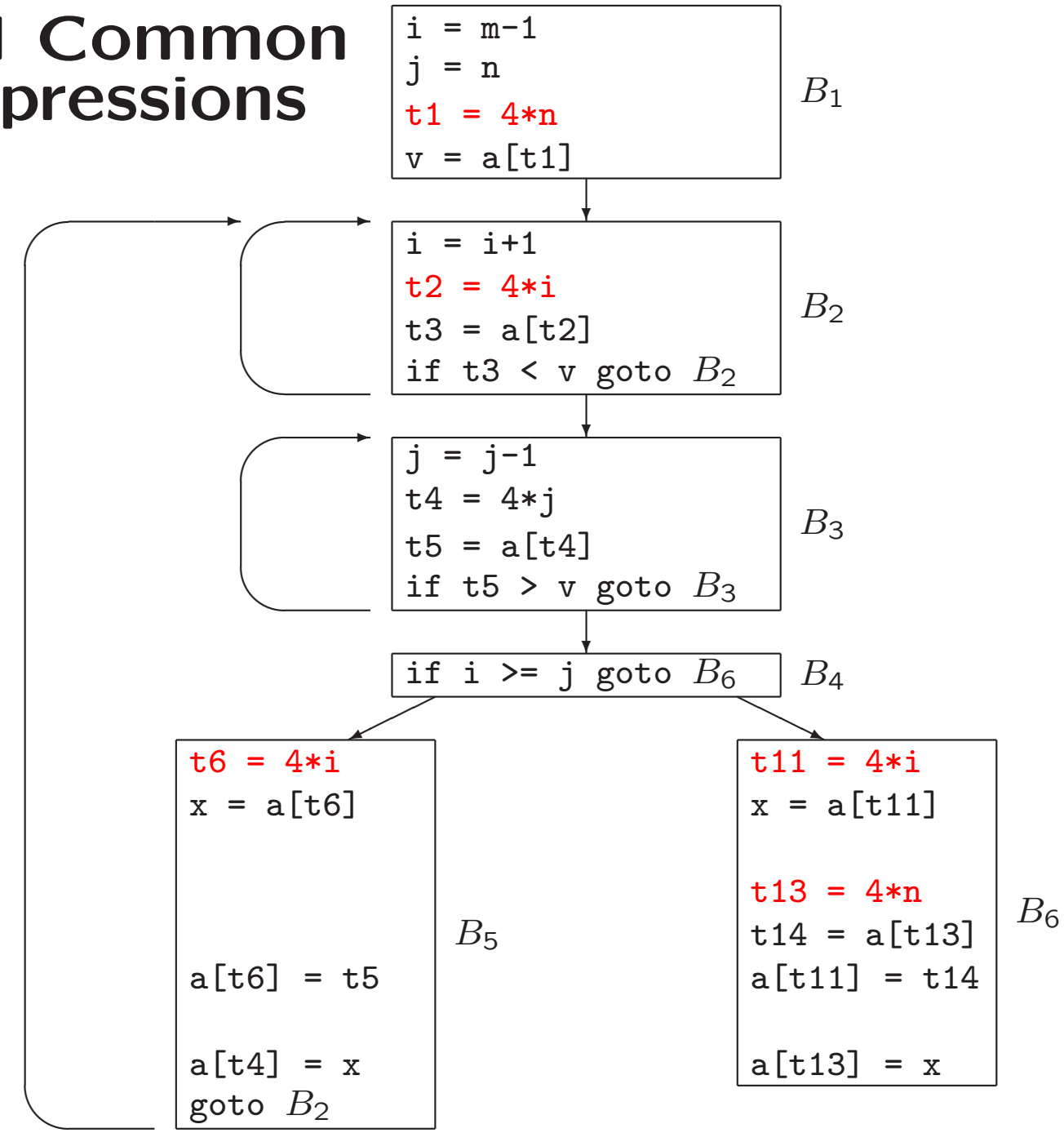
Global Common Subexpressions



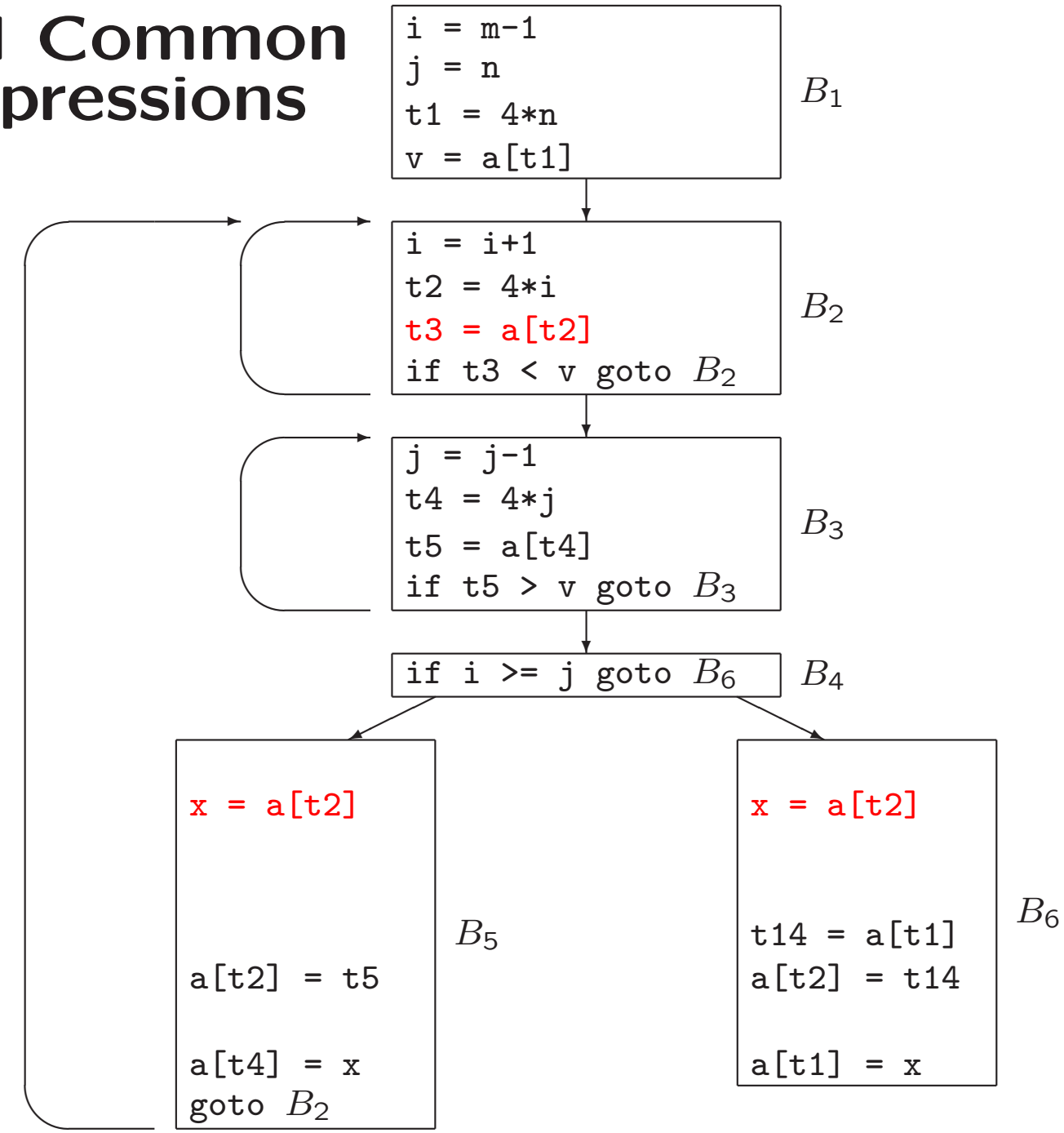
Global Common Subexpressions



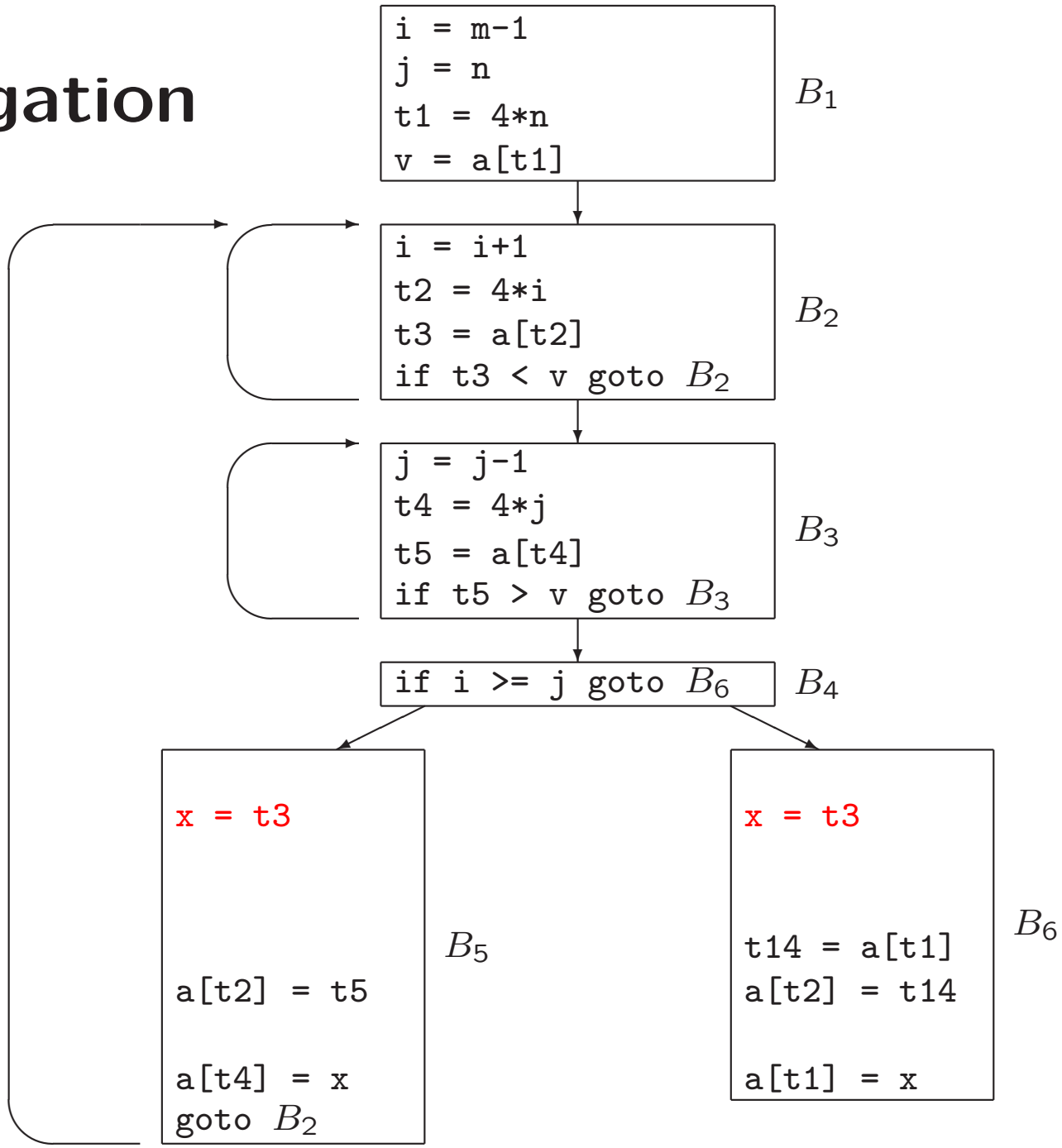
Global Common Subexpressions



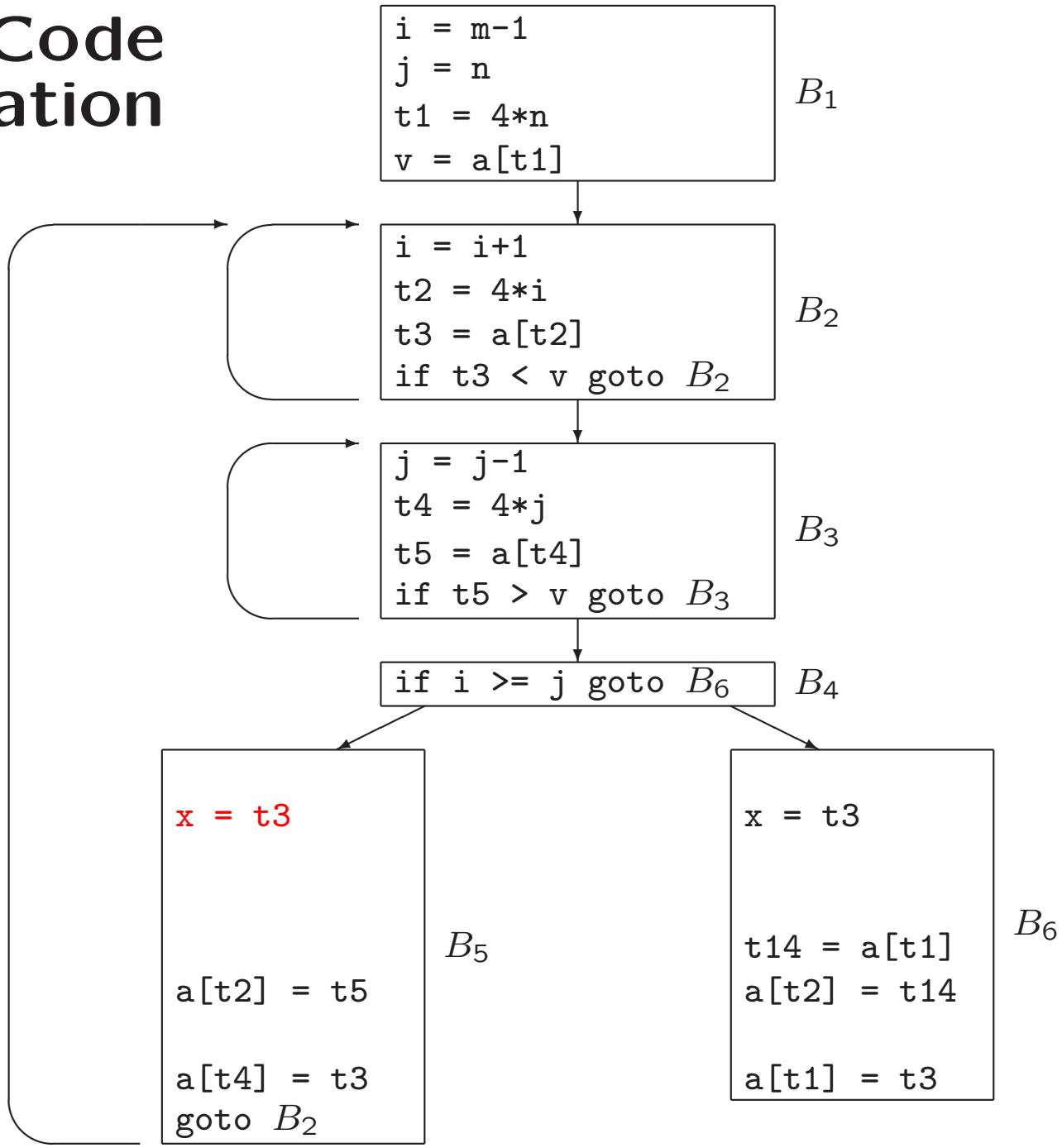
Global Common Subexpressions



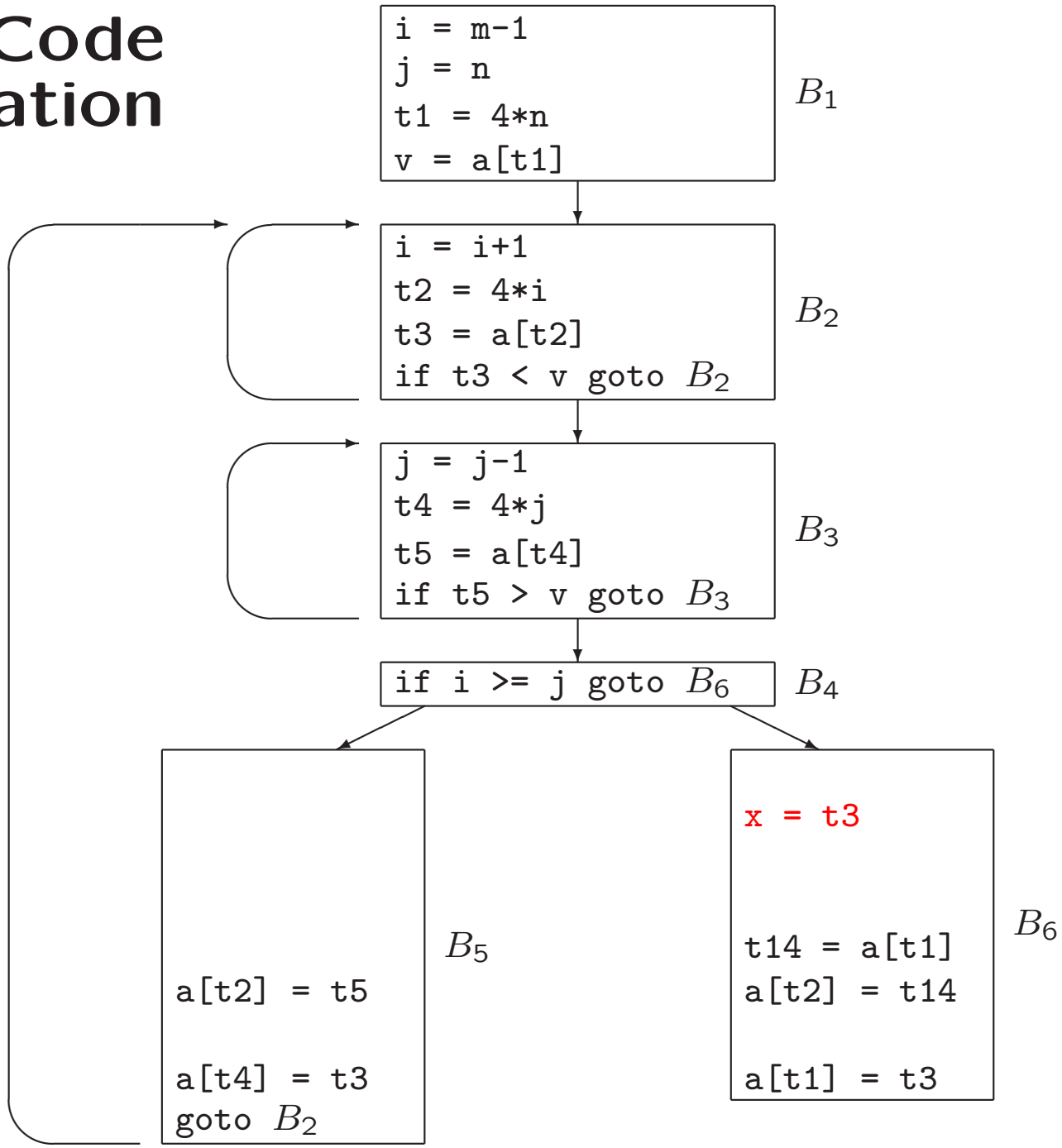
Copy Propagation



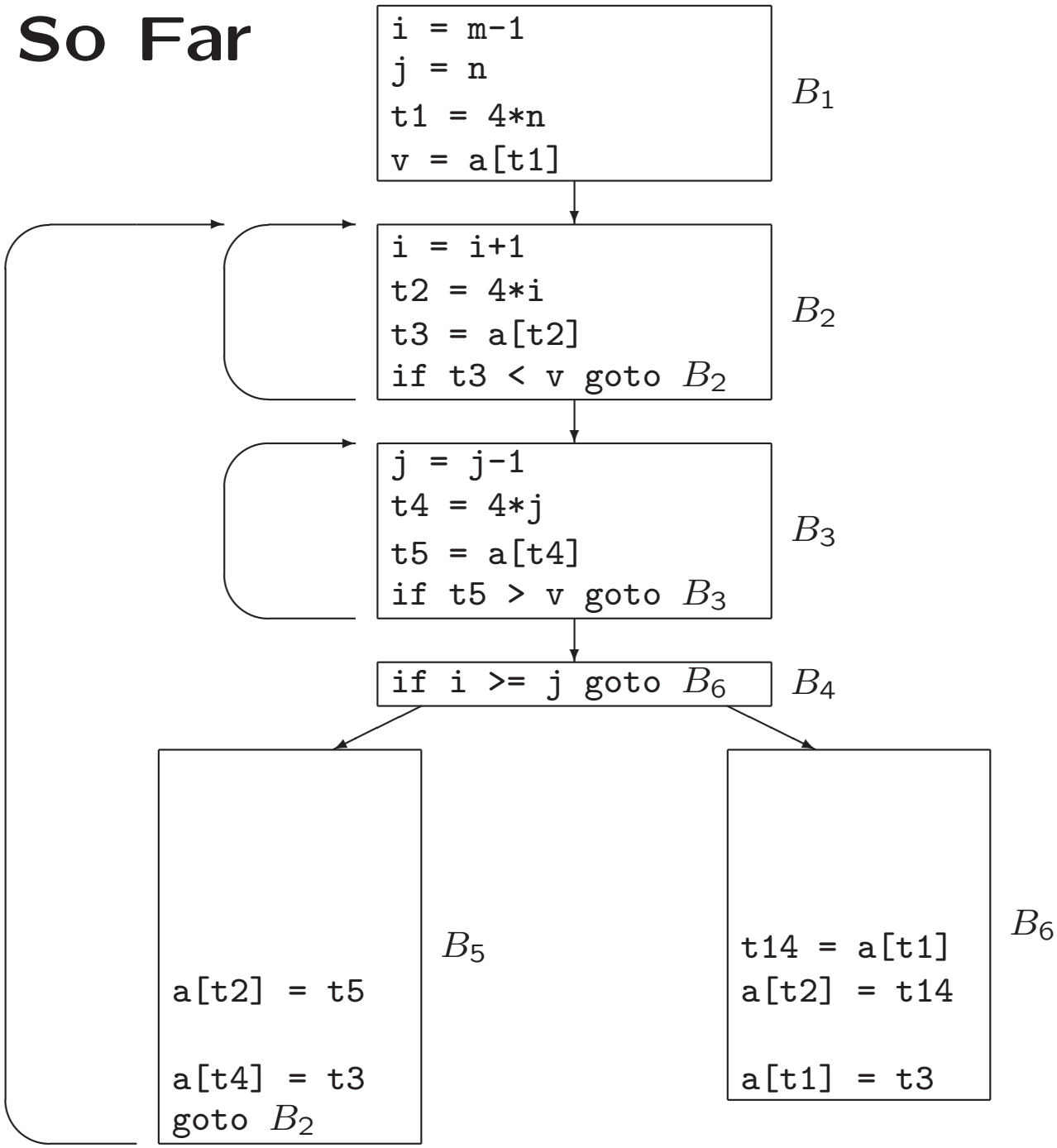
Dead-Code Elimination



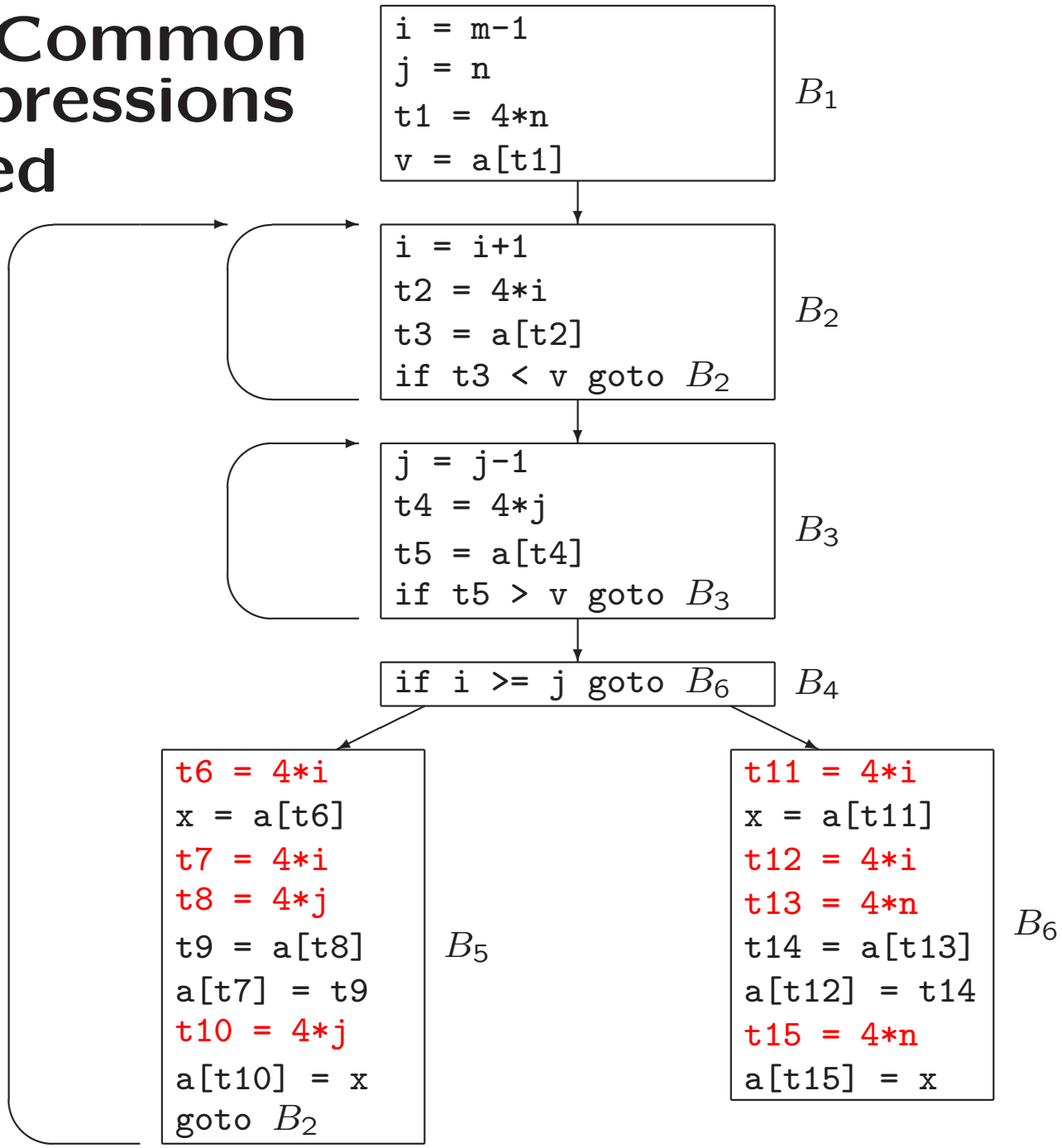
Dead-Code Elimination



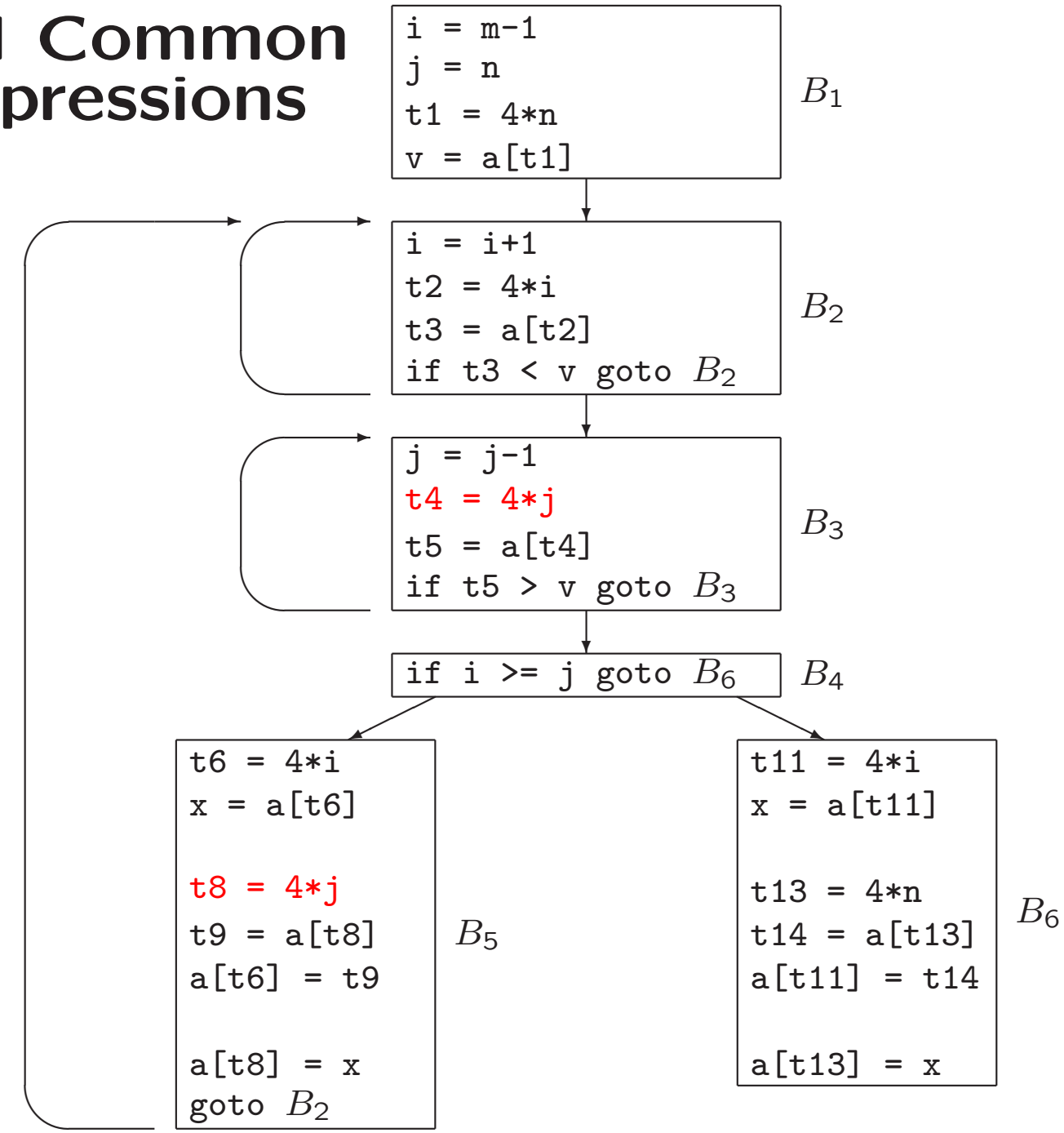
Result So Far



Local Common Subexpressions revisited



Global Common Subexpressions



Code Motion

- loop-invariant computation
- compute before loop
- Example:

```
while (i <= limit-2)  /* statement does not change limit */
```

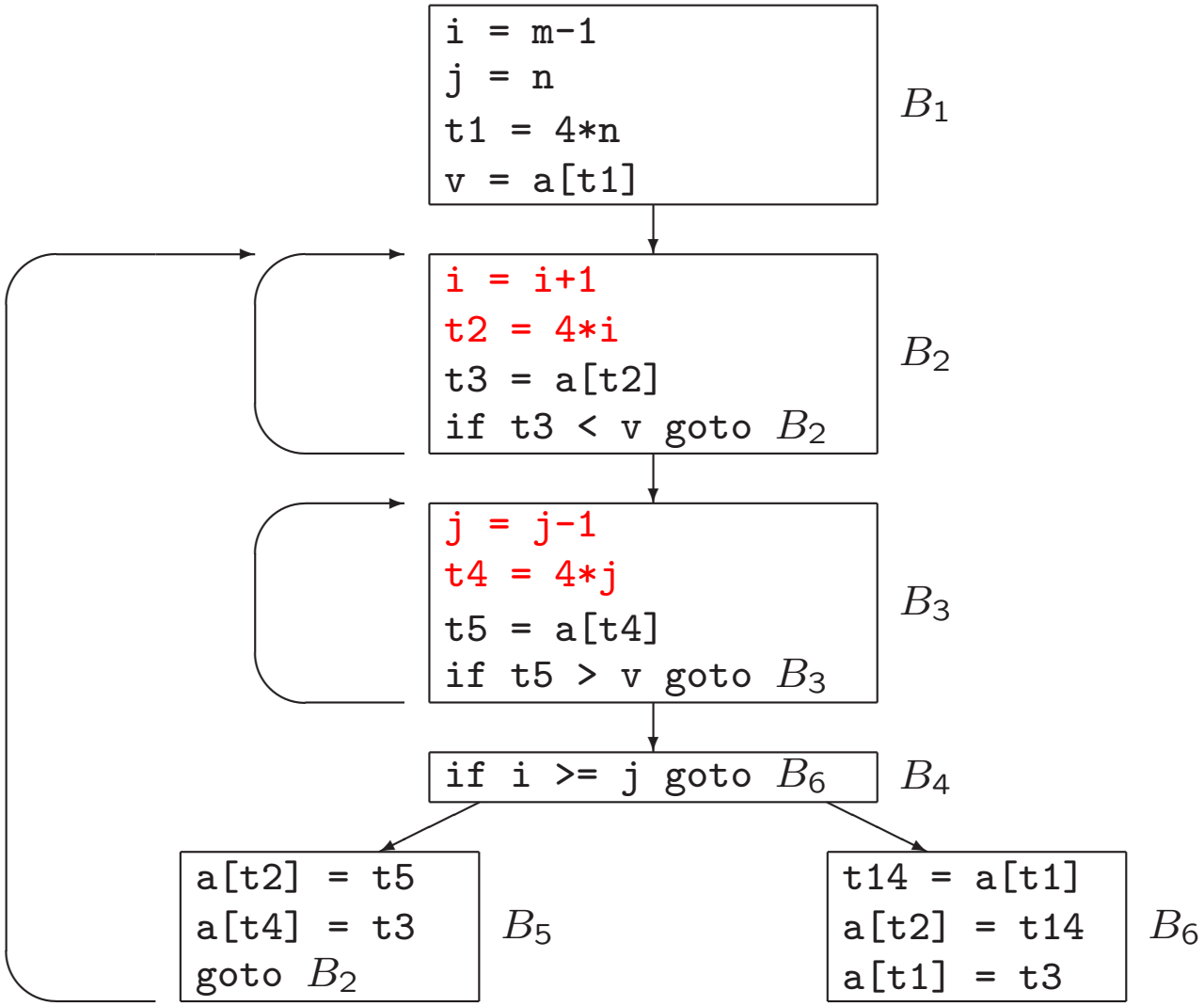
After code-motion

```
t = limit-2
while (i <= t)  /* statement does not change limit or t */
```

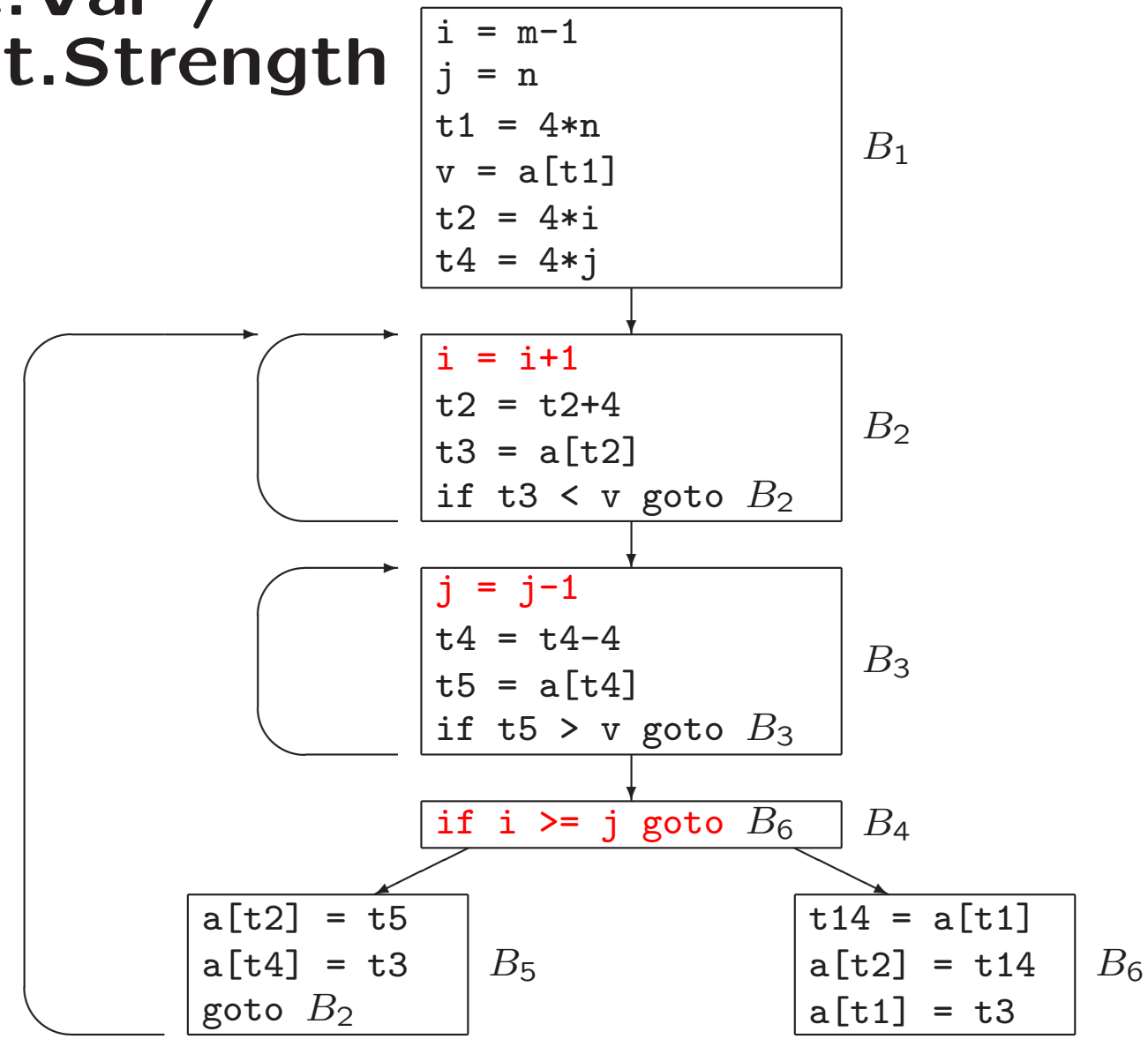
Induction Variables and Reduction in Strength

- Induction variable: each assignment to x of form $x = x + c$
- Reduction in strength: replace expensive operation by cheaper one

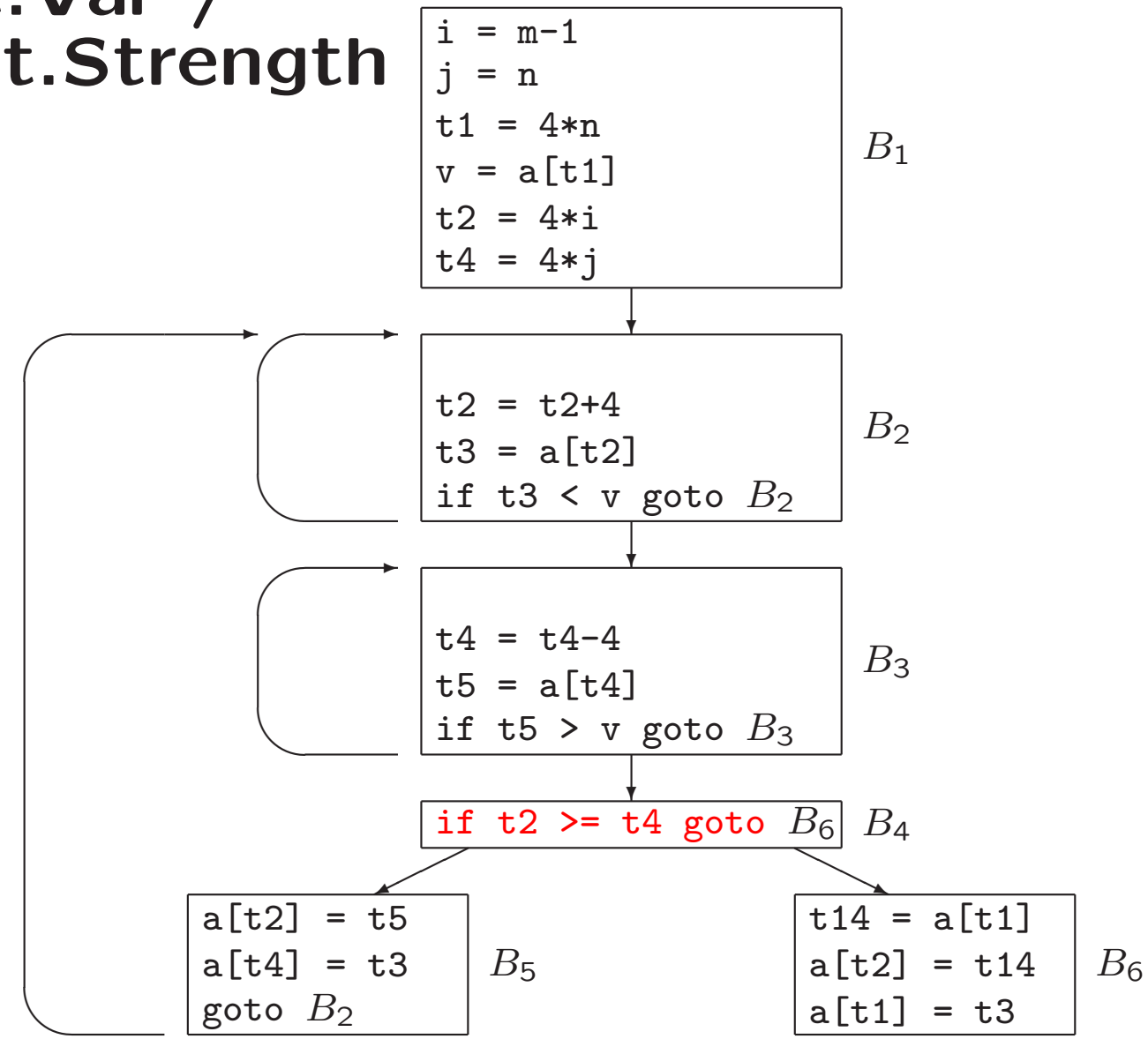
Induct.Var / Reduct.Strength



Induct.Var / Reduct.Strength

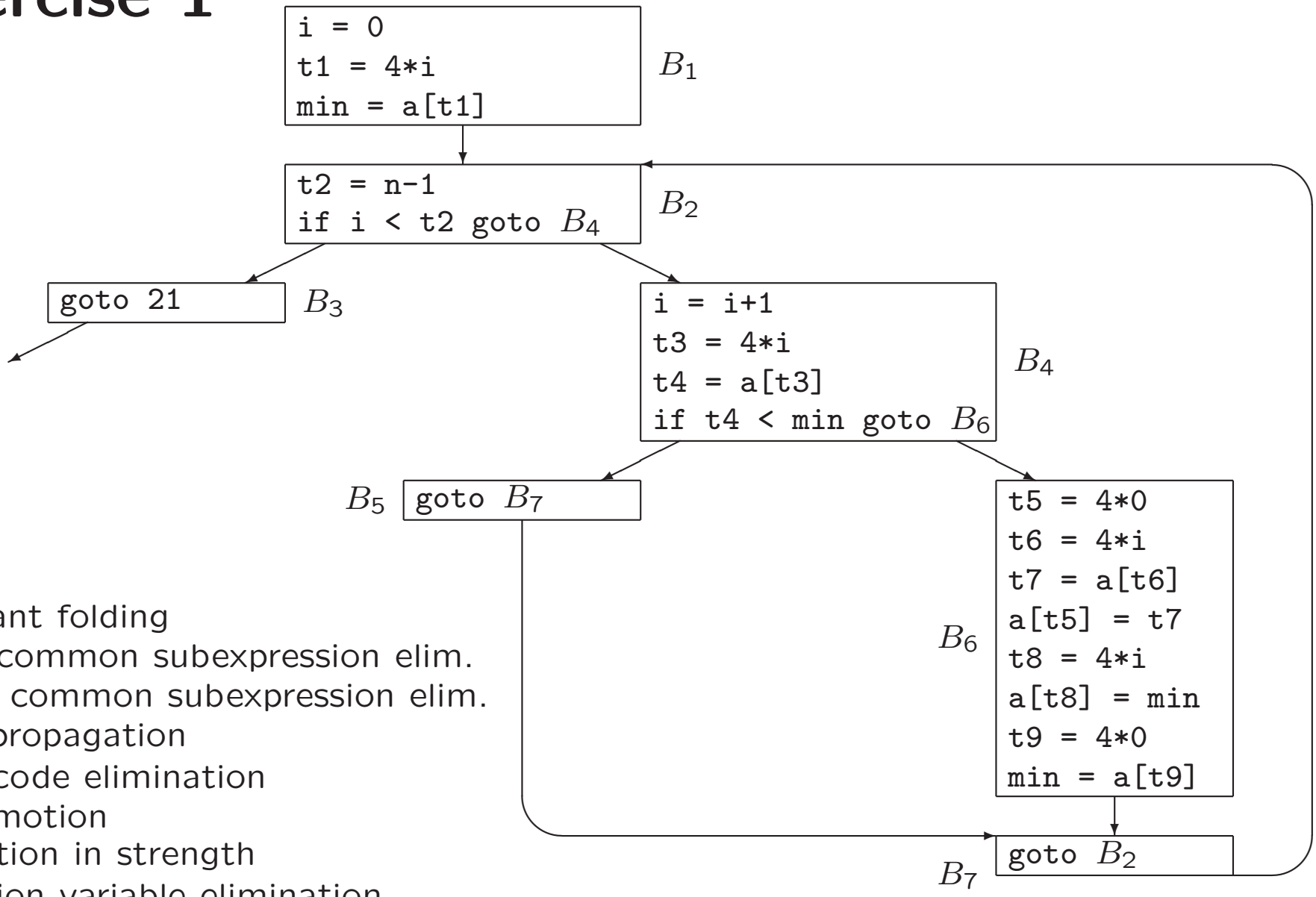


Induct.Var / Reduct.Strength



Exercise 1

Flow Graph Exercise 1



Constant folding
Local common subexpression elim.
Global common subexpression elim.
Copy propagation
Dead-code elimination
Code motion
Reduction in strength
Induction-variable elimination

Volgende week

- Practicum over opdracht 4
- Inleveren 14 december
- Vrijdag 8 december: laatste hoor-/werkcollege

Compilerconstructie

college 9

Code Optimization

Chapters for reading:

8.5–8.5.6, 8.7

9.intro, 9.1