Compilerconstructie

najaar 2013

`http://www.liacs.nl/home/rvvliet/coco/`

Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs(dot)nl

college 8, dinsdag 12 november 2013

Code Generation

Code Optimization

8.6 A Simple Code Generator

Use of registers

- Operands of operation must be in registers
- To hold values of temporary variables
- To hold (global) values that are used in several blocks
- To manage run-time stack

Assumption: subset of registers available for block

Machine instructions of form

- `LD` *reg*, *mem*
- `ST` *mem*, *reg*
- *OP* *reg*, *reg*, *reg*

Register and Address Descriptors

- Register descriptor keeps track of what is currently in register
  - Example:

    $$\texttt{LD}\ R, x \qquad \rightarrow \text{register } R \text{ contains } x$$

  - Initially, all registers are empty

- Address descriptor keeps track of locations where current value of a variable can be found
  - Example:

    $$\texttt{LD}\ R, x \qquad \rightarrow x \text{ is (also) in } R$$

  - Information stored in symbol table

The Code-Generation Algorithm

For each three-address instruction $x = y$ *op* z

1. Use *getReg*($x = y$ *op* z) to select registers R_x, R_y, R_z
2. If y is not in R_y, then issue instruction `LD` R_y, y',
   where y' is a memory location for y
   (according to address descriptor)
3. If z is not in R_z, . . .
4. Issue instruction *OP* R_x, R_y, R_z

Special case: $x = y$ . . .

At end of block: store all variables that are live-on-exit and not in their memory locations (according to address descriptor)

Managing Register / Address Descriptors

1. For the instruction `LD` R, x, . . .

2. For the instruction `ST` x, R, . . .

3. For an operation like `ADD` R_x, R_y, R_z, implementing $x = y + z$,

   (c) Remove R_x from addr. descr. of other variables

   (d) Remove x from reg. descr. of other registers

   (a) Change reg. descr. for R_x: only x

   (b) Change addr. descr. for x: only in R_x (not in x itself!)

4. For the copy statement $x = y$, . . .

Managing Register / Address Descriptors

Example: $d = (a - b) + (a - c) + (a - c)$ $a = \ldots$ old value of d

```
t = a - b
    LD  R1, a
    LD  R2, b
    SUB R2, R1, R2
u = a - c
    LD  R3, c
    SUB R1, R1, R3
v = t + u
    ADD R3, R2, R1
a = d
    LD  R2, d
d = v + u
    ADD R1, R3, R1

exit
    ST  a, R2
    ST  d, R1
```

R1	R2	R3

a	b	c	d	t	u	v
a	b	c	d			

Managing Register / Address Descriptors

Example: $d = (a - b) + (a - c) + (a - c)$ $a = \ldots$ old value of d

```
t = a - b
    LD  R1, a
    LD  R2, b
    SUB R2, R1, R2
u = a - c
    LD  R3, c
    SUB R1, R1, R3
v = t + u
    ADD R3, R2, R1
a = d
    LD  R2, d
d = v + u
    ADD R1, R3, R1

exit
    ST  a, R2
    ST  d, R1
```

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
a,R2	b	c	d,R1			R3

Function *getReg*

For each instruction $x = y$ *op* z

- To compute R_y
  1. If y is in register, $\longrightarrow R_y$
  2. Else, if empty register available, $\longrightarrow R_y$
  3. Else, select occupied register
     For each register R and variable v in R
     (a) If v is also somewhere else, then OK
     (b) If v is x, and x is not z, then OK
     (c) Else, if v is not used later, then OK
     (d) Else, $\texttt{ST}\ v, R$ is required

     Take R with smallest number of stores

Function *getReg*

For each instruction $x = y \ op \ z$

- To compute R_x, similar with few differences (which?)

For each instruction $x = y$, choose $R_x = R_y$

Exercise.
(Exercise 1 one from exercise class; cf. Exercise 8.6.1/8.6.4)

Consider the following C code:

```
x = a[i] + 1;
k = x;
b[i][j] = k + y;
```

Assume
that all array elements are integers taking four bytes each,
and that b is 100×100 array

a. Generate three-address code for this C code
b. Convert your three-address code into machine code, using the simple code-generation algorithm of this section, assuming three registers are available. Show the register and address descriptors after each step.

Addressing Modes of Target Machine

(from college 7)

Form	Address	Example
r	r	`LD R1, R2`
x	x	`LD R1, x`
$a(r)$	$a + \mathit{contents}(r)$	`LD R1, a(R2)`
$c(r)$	$c + \mathit{contents}(r)$	`LD R1, 100(R2)`
$*r$	$\mathit{contents}(r)$	`LD R1, *R2`
$*c(r)$	$\mathit{contents}(c + \mathit{contents}(r))$	`LD R1, *100(R2)`
$\#c$		`LD R1, #100`

8.8 Register Allocation and Assignment

So far, live variables in registers are stored at end of block

Use of registers

- Operands of operation must be in registers
- To hold values of temporary variables
- To hold (global) values that are used in several blocks
- To manage run-time stack

Usage counts

With x in register during loop L

- Save . . . for . . . use of x that is not preceded by assignment in same block

- Save . . . for each block, where x is assigned a value and x is live on exit

-

$$\text{Total savings} \approx \sum_{\text{blocks } B \in L} \ldots$$

Choose variables x with largest savings

Usage counts

With x in register during loop L

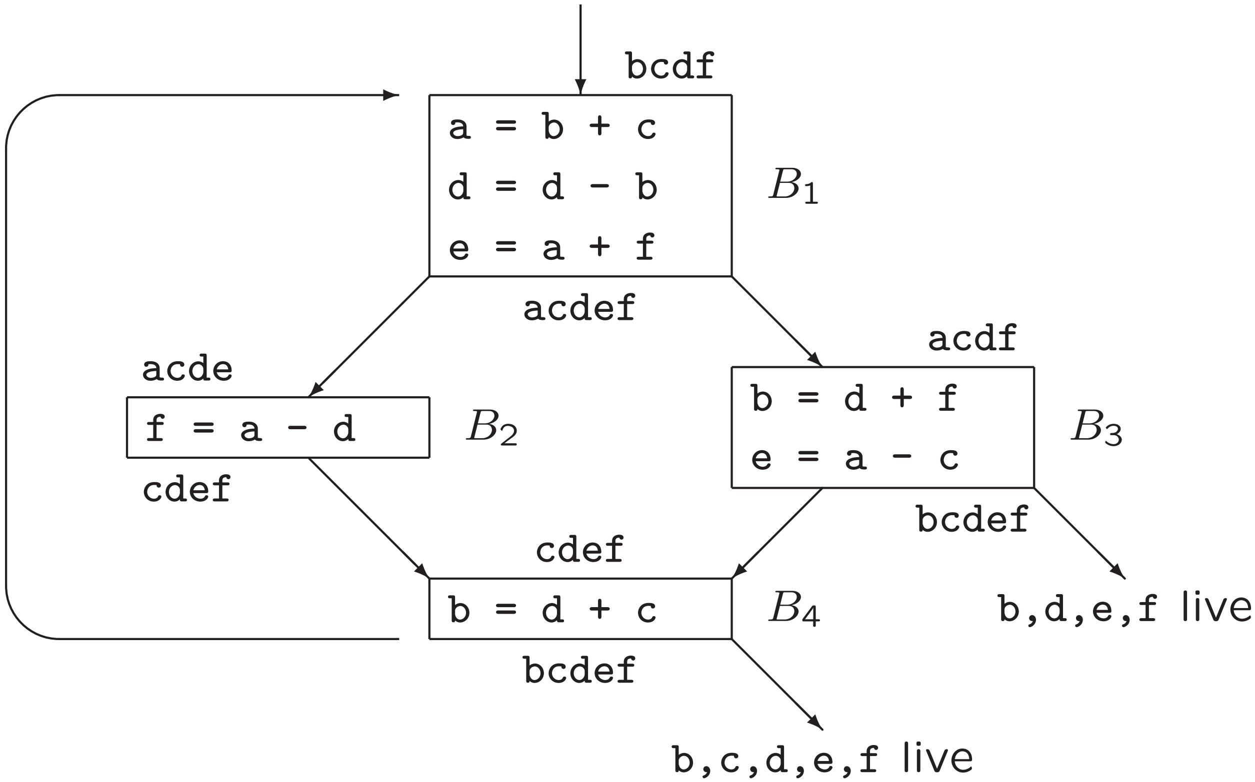
- Save 1 for each use of x that is not preceded by assignment in same block

- Save 2 for each block, where x is assigned a value and x is live on exit

- $$\text{Total savings} \approx \sum_{\text{blocks } B \in L} \textit{use}(x,B) + 2 * \textit{live}(x,B)$$

Choose variables x with largest savings

Usage counts (Example)



Savings for a are $1 + 1 + 1 * 2 = 4$

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

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
```

Local Common Subexpression Elimination

- 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
```

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. . .

Algebraic Transformations

(see 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 (peep-hole) 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

Redundant Instruction Elimination

Example:

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

Eliminating Unreachable Code

Example:

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

Eliminating Unreachable Code

Example:

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

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

Flow-of-Control Optimizations

Example 1:

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

Example 3:

```
     goto L1
     .  .  .
L1:  1f a < b goto L2
L3:
```

9.1 The Principal Sources of Optimization

Causes of redundancy

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

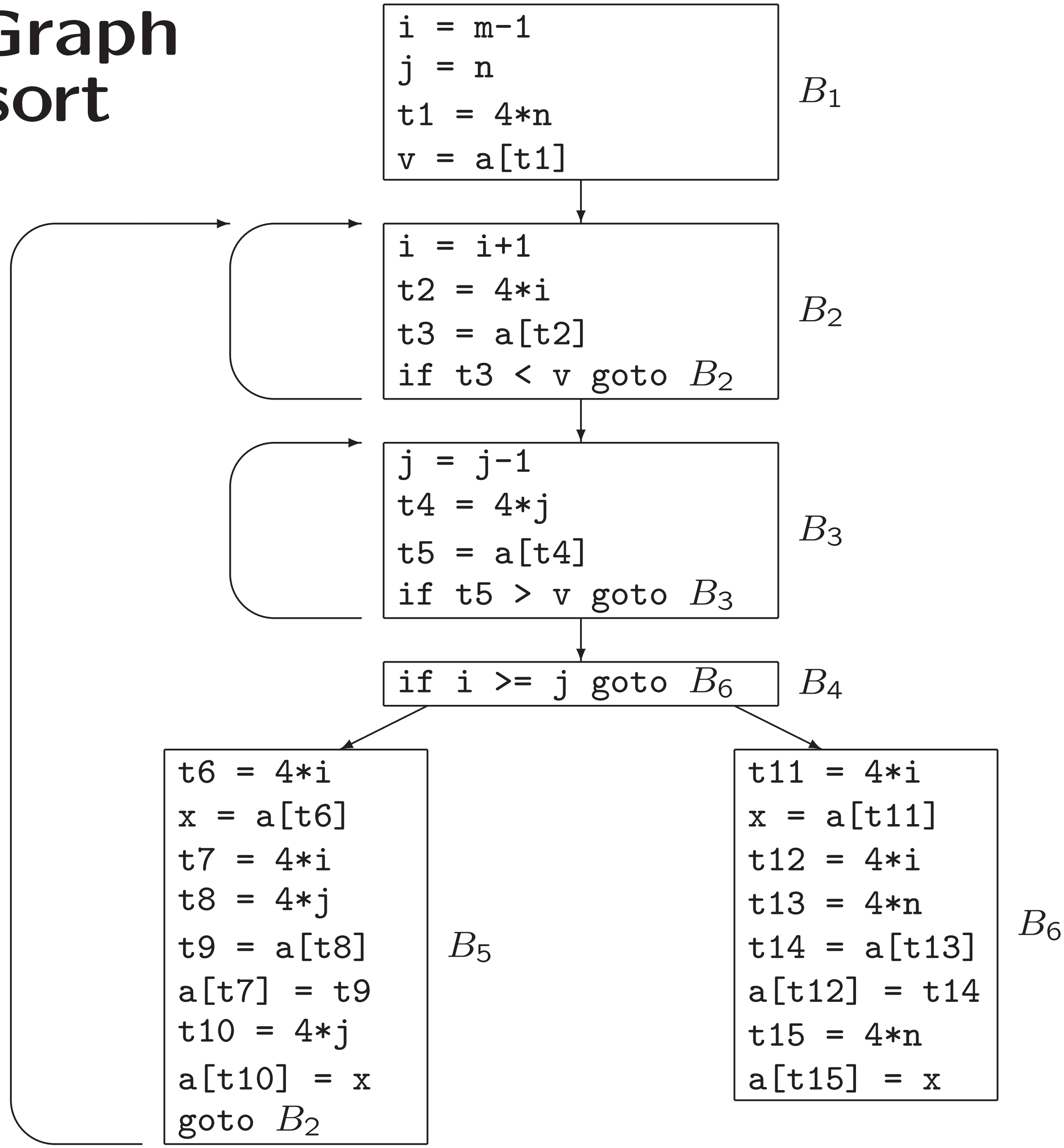
A Running Example: Quicksort

```
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);
}
```

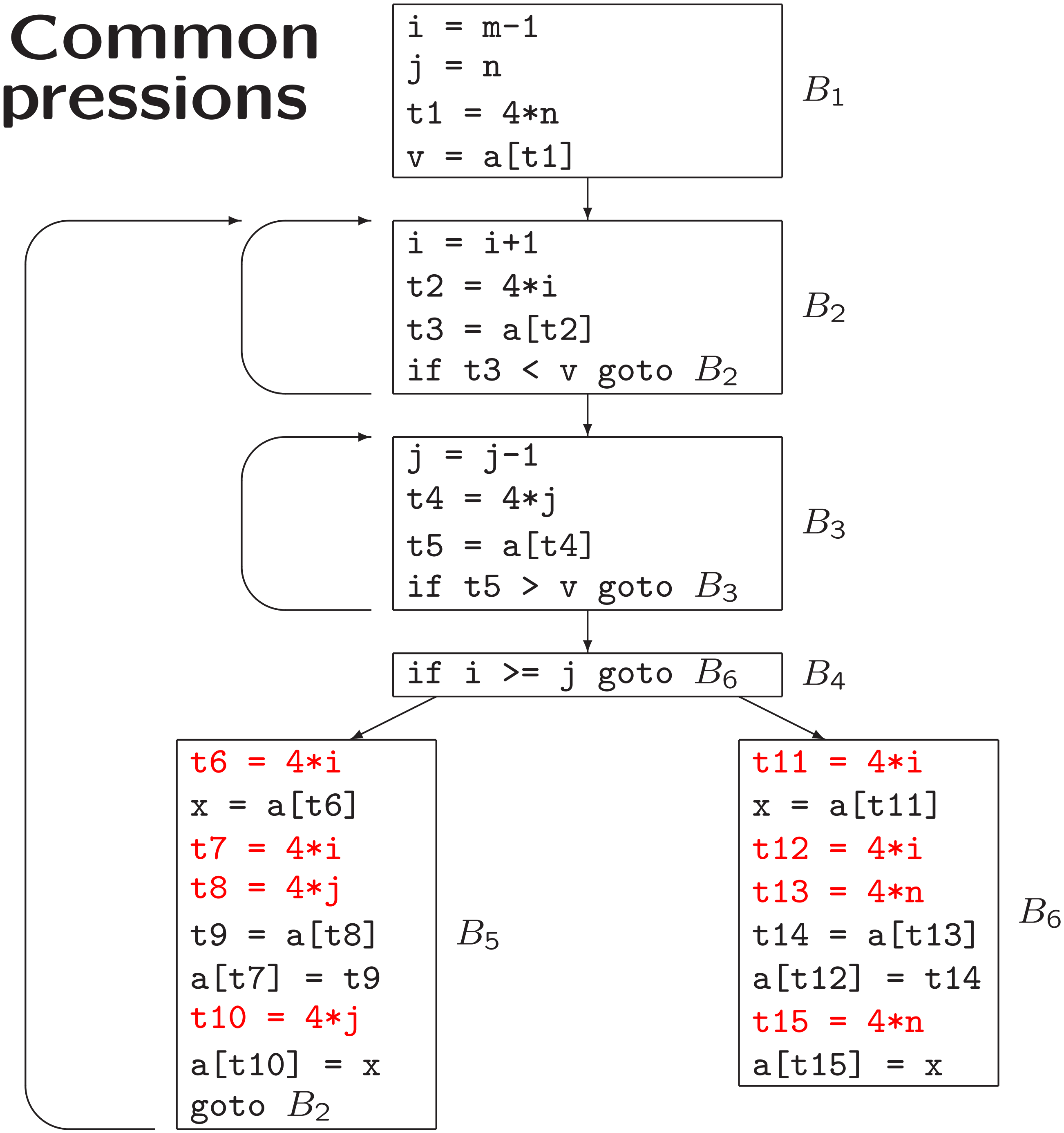
Three-Address Code Quicksort

```
→  (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
```

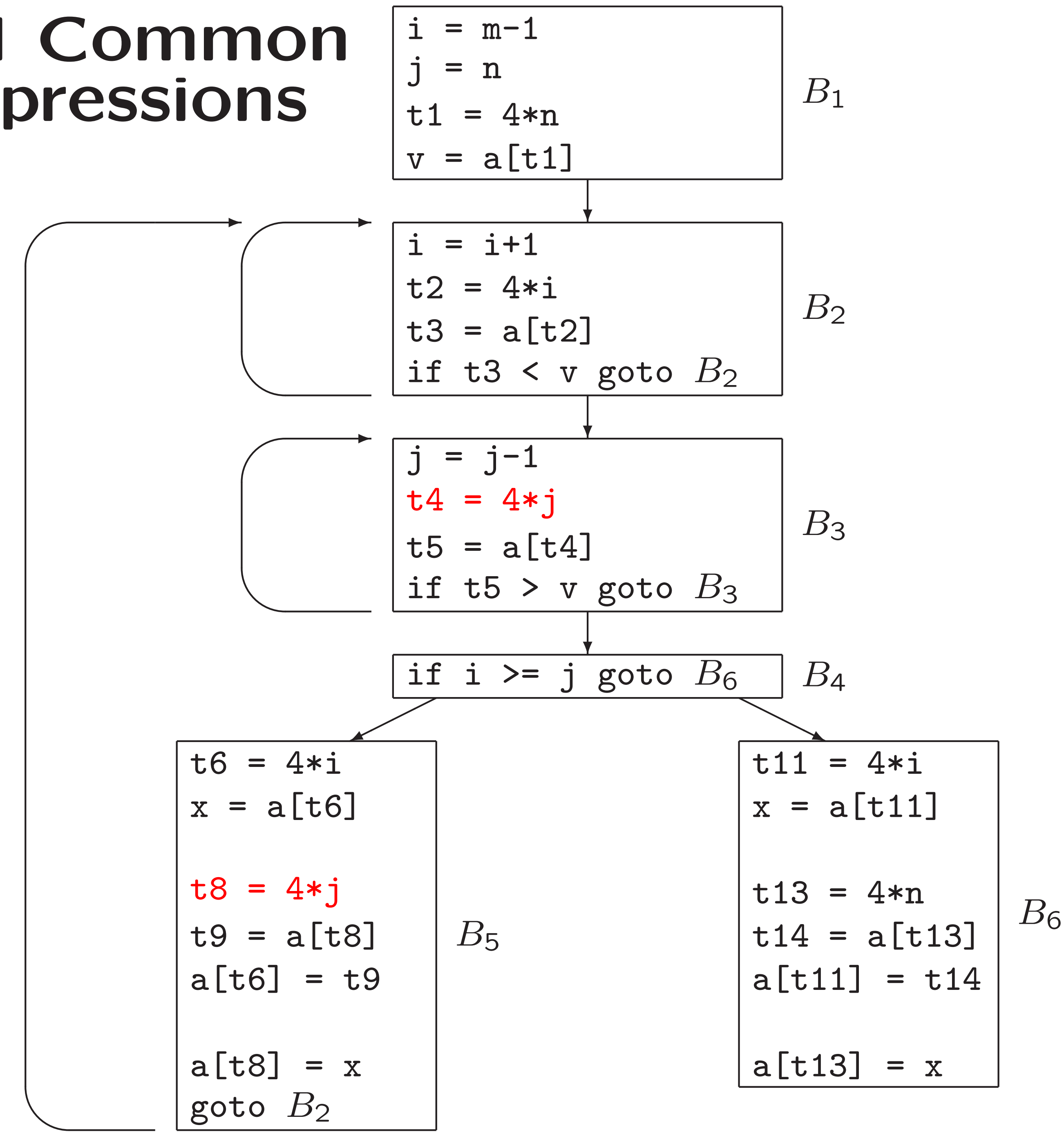
Flow Graph Quicksort



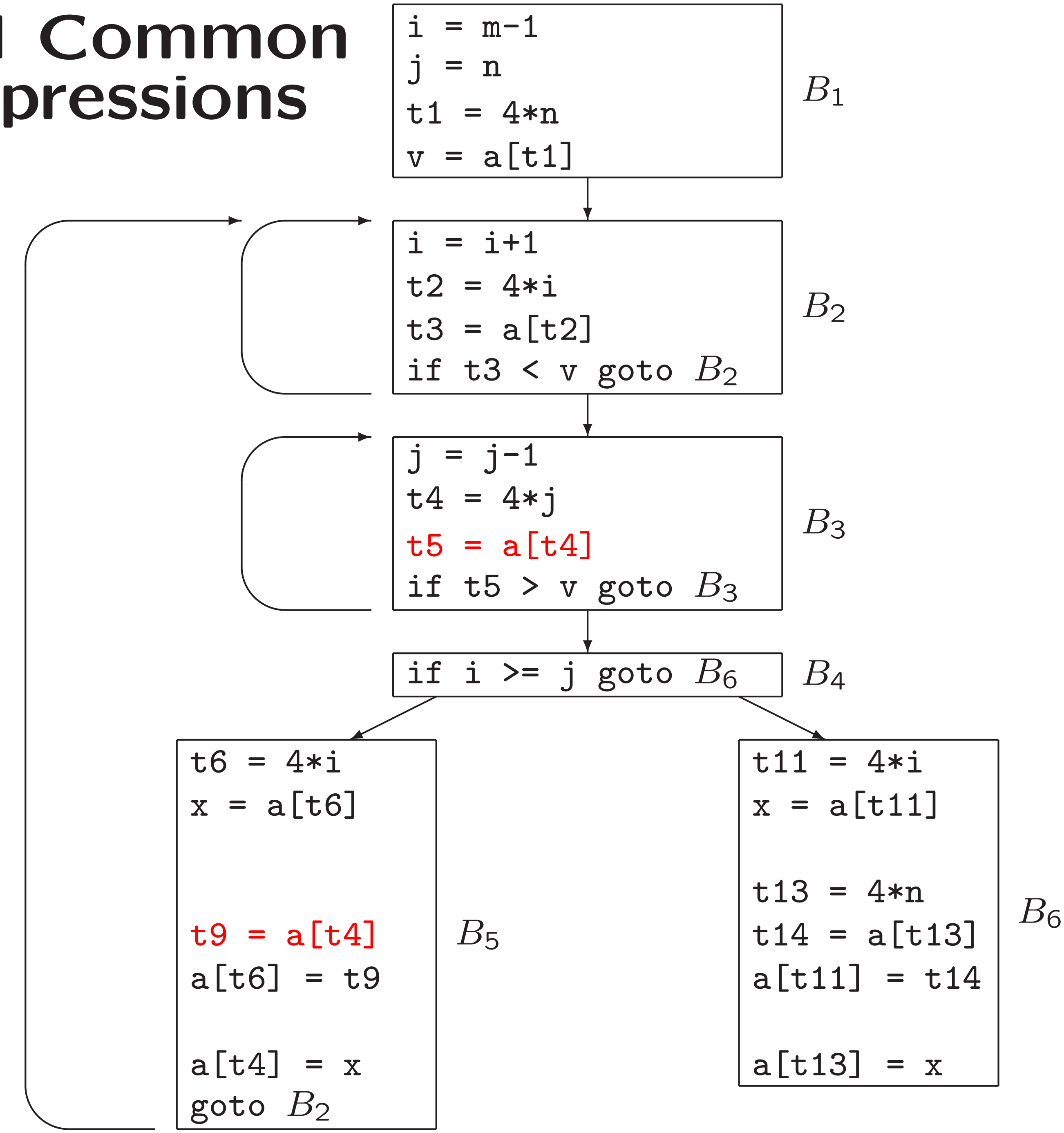
Local Common Subexpressions



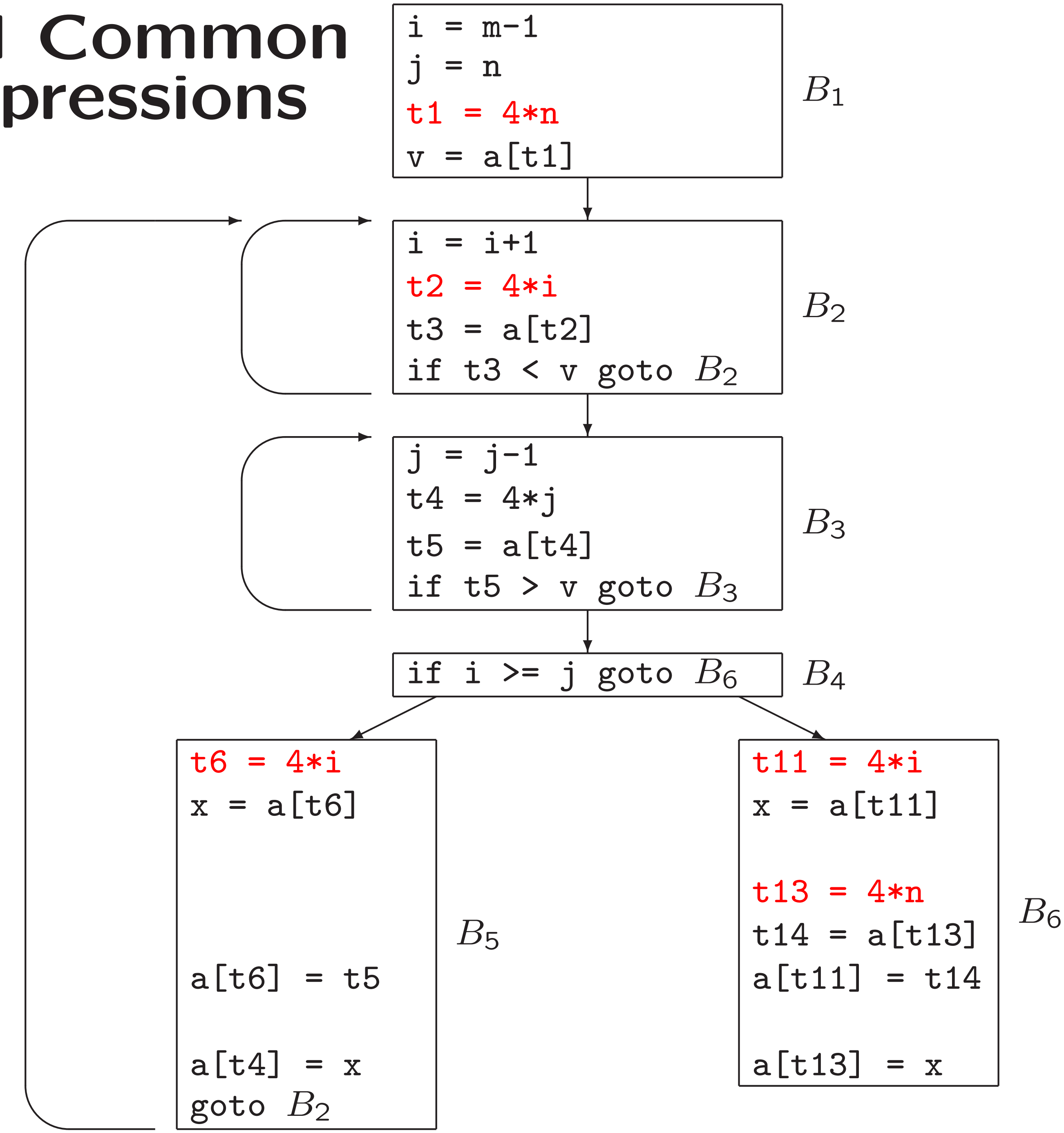
Global Common Subexpressions



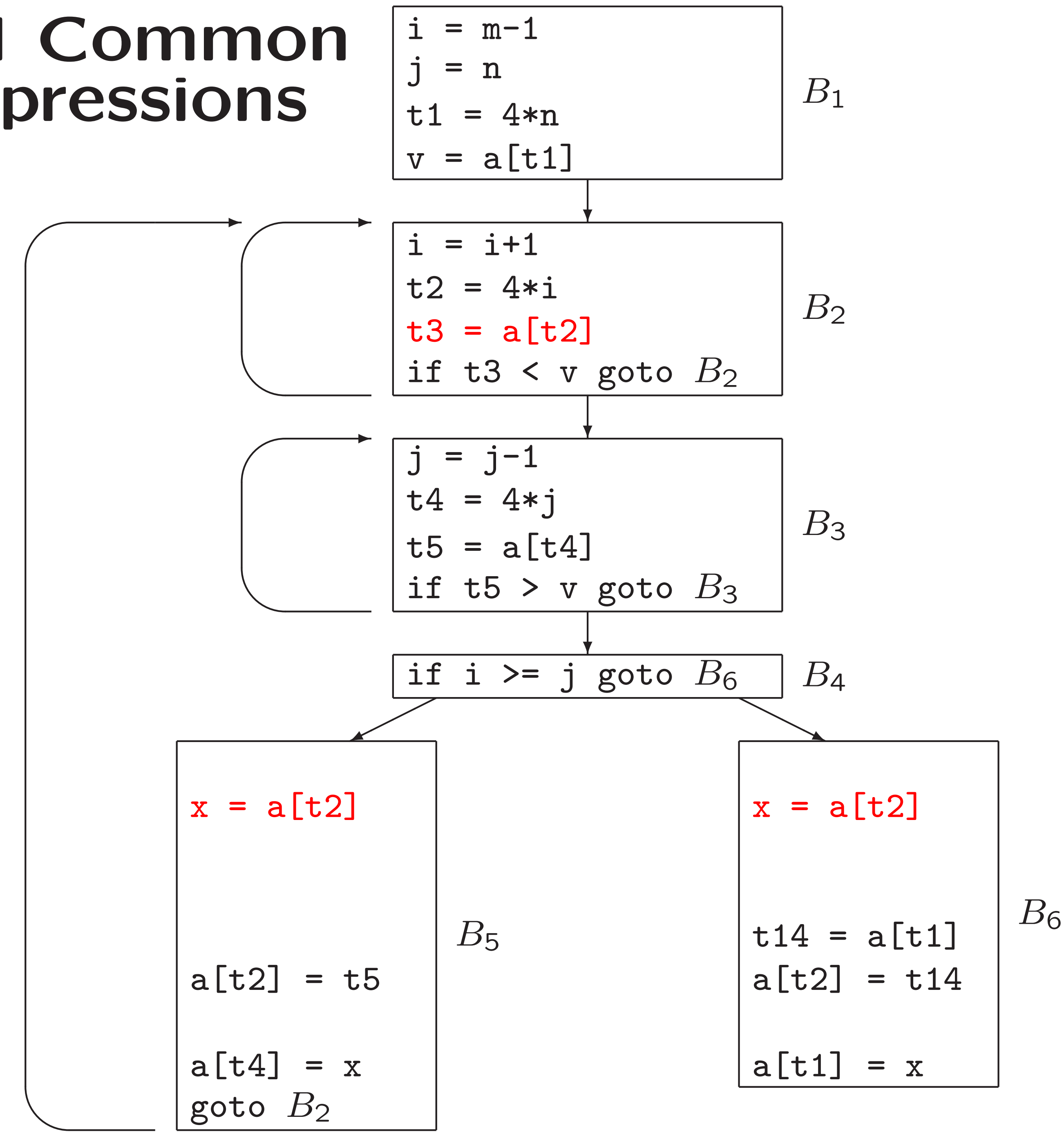
Global Common Subexpressions



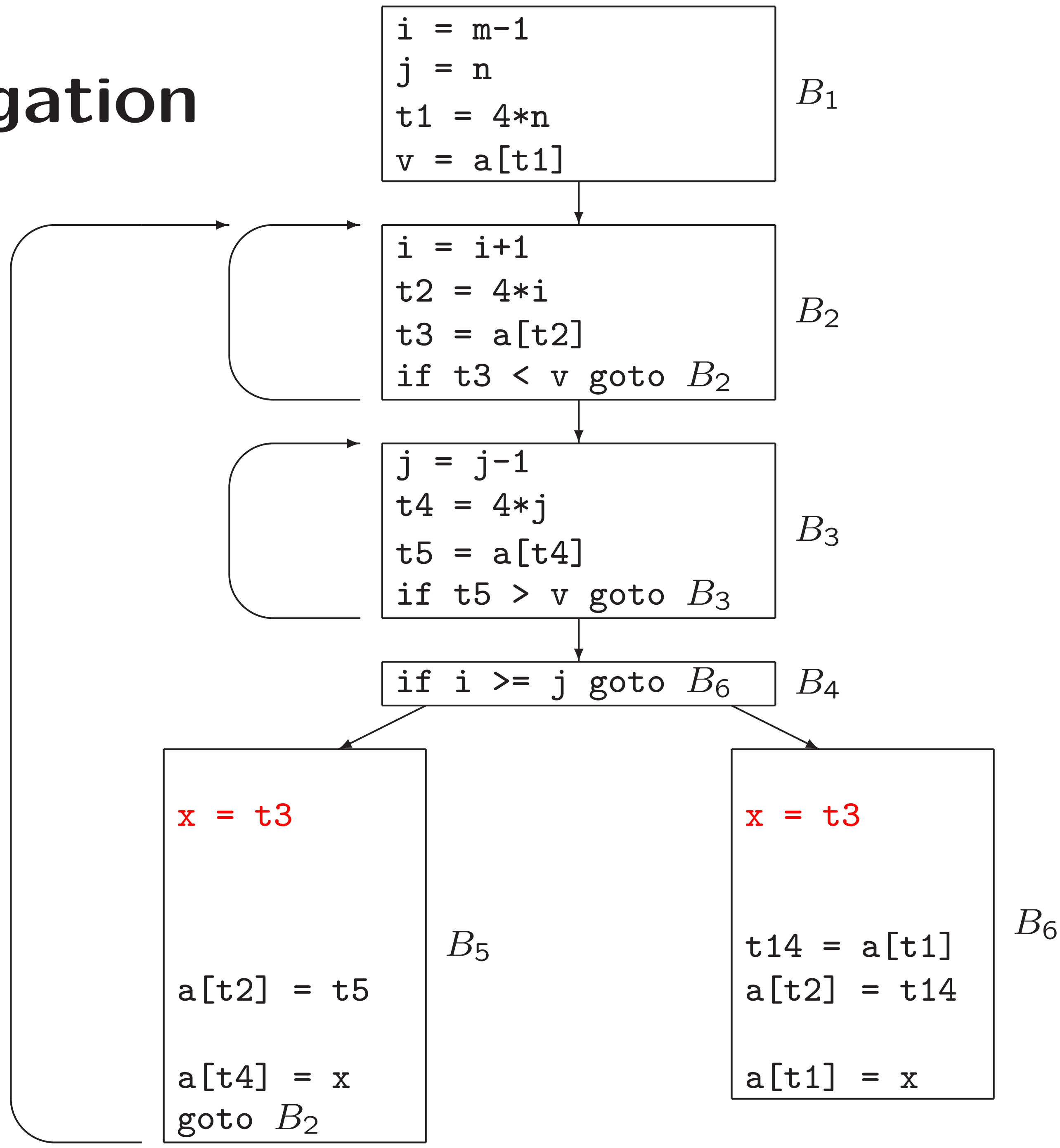
Global Common Subexpressions



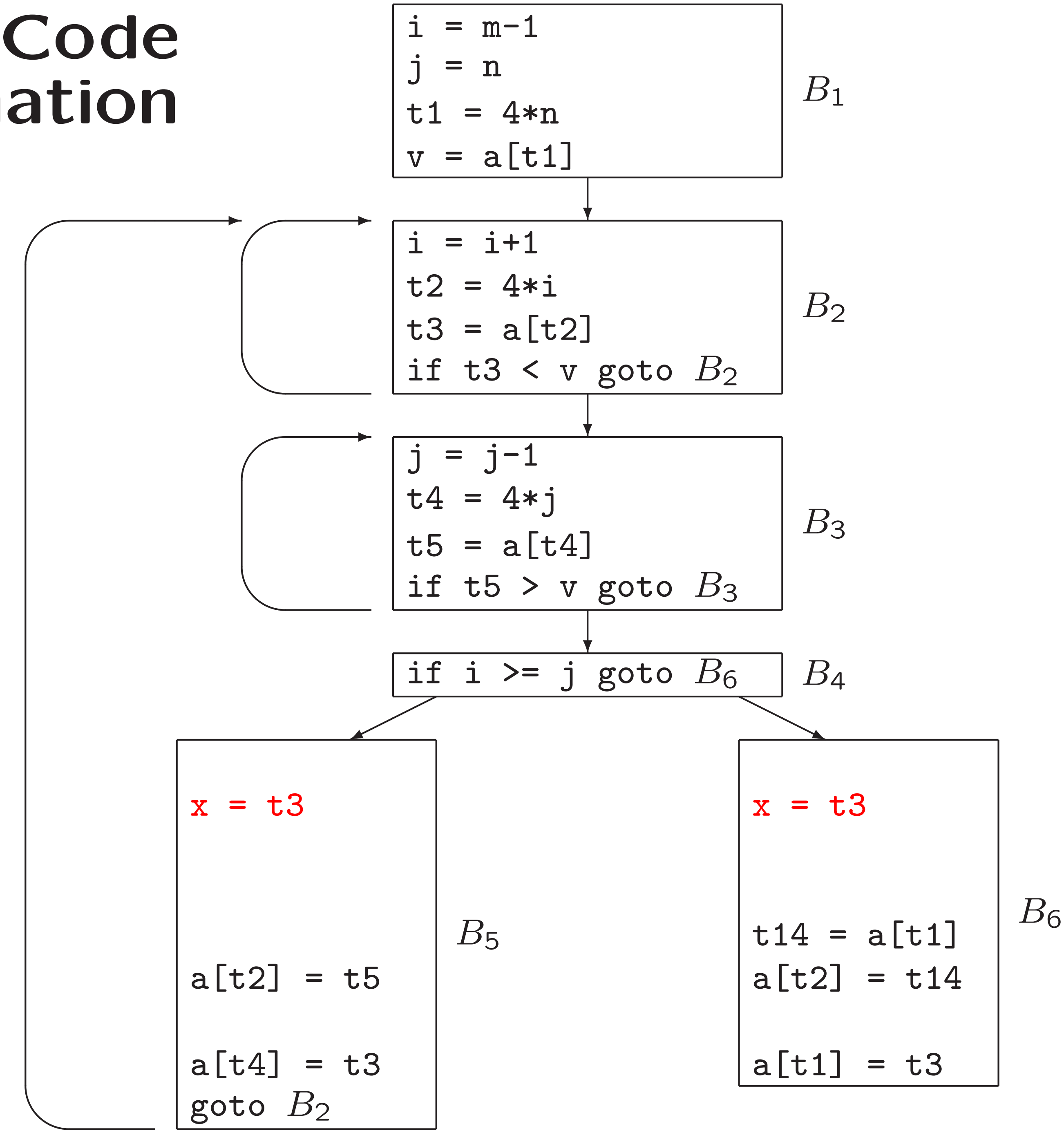
Global Common Subexpressions



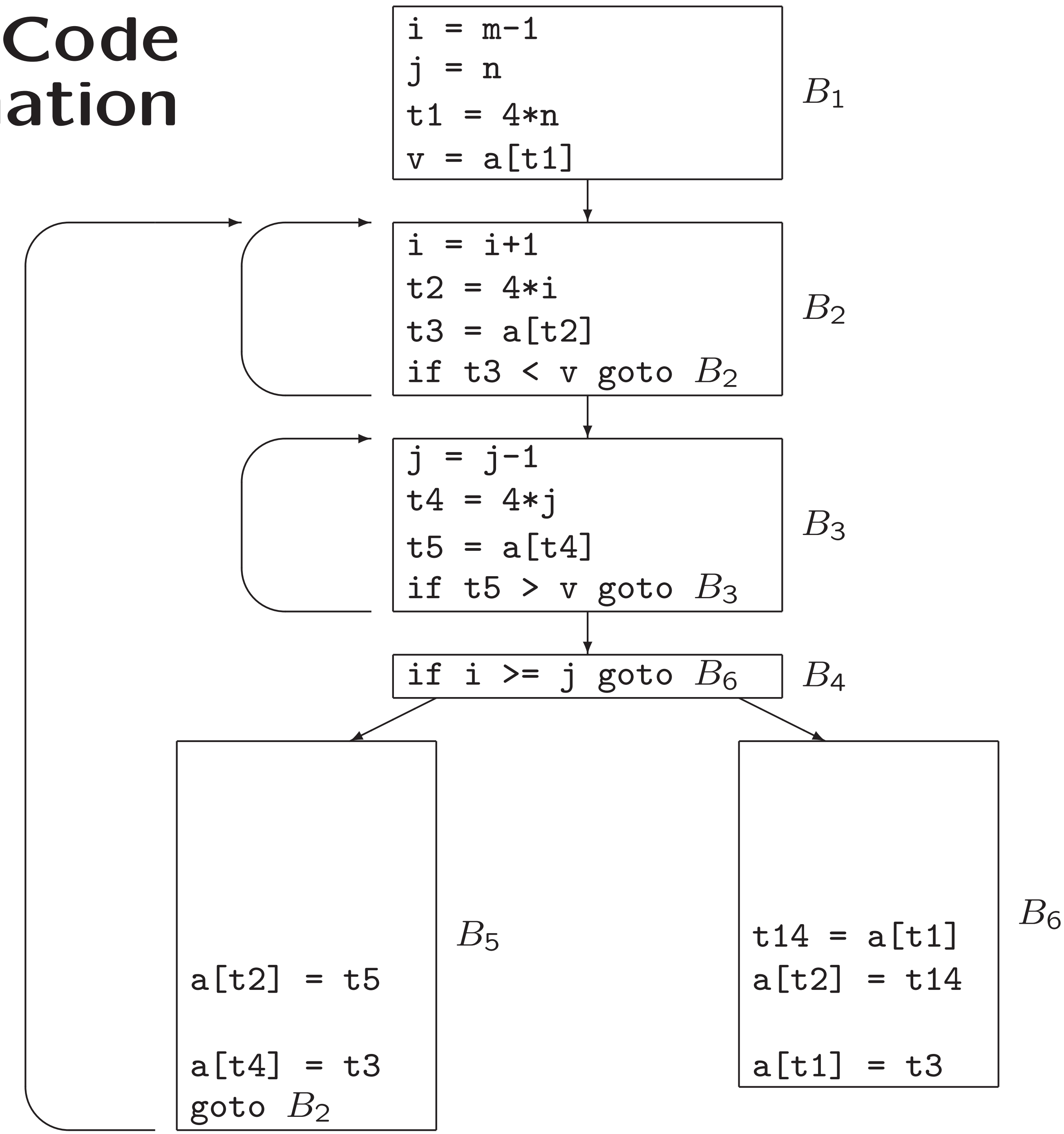
Copy Propagation



Dead-Code Elimination



Dead-Code Elimination



Code Motion

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

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

After code-motion

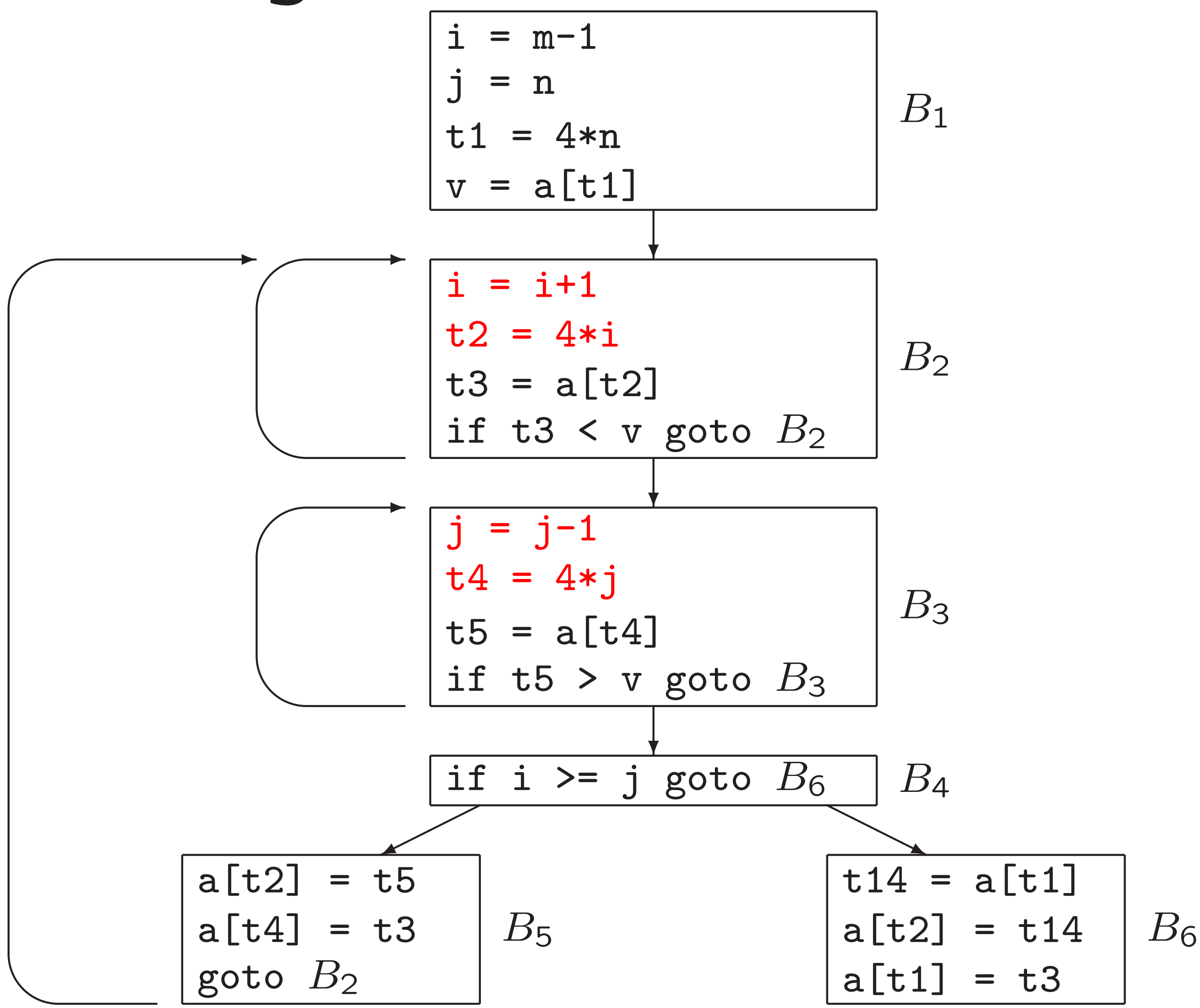
```
t = limit-2
while (i <= t)  /* statement does not change limit or t */
```

Induction Variables and Reduction in Strength

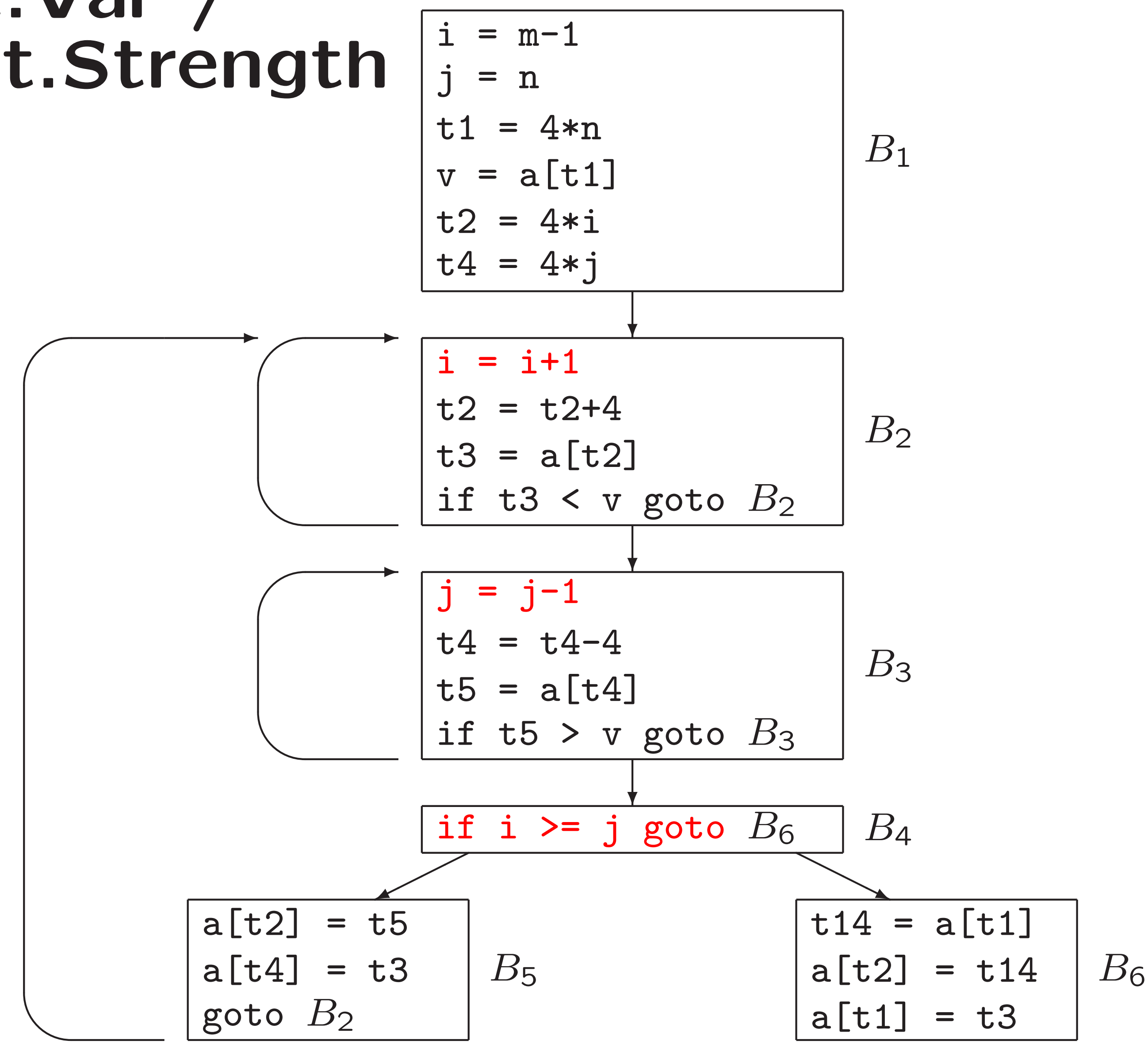
- Induction variable: each assignment to x of form $x = x + c$

- Reduction in strength: replace expensive operation by cheaper one

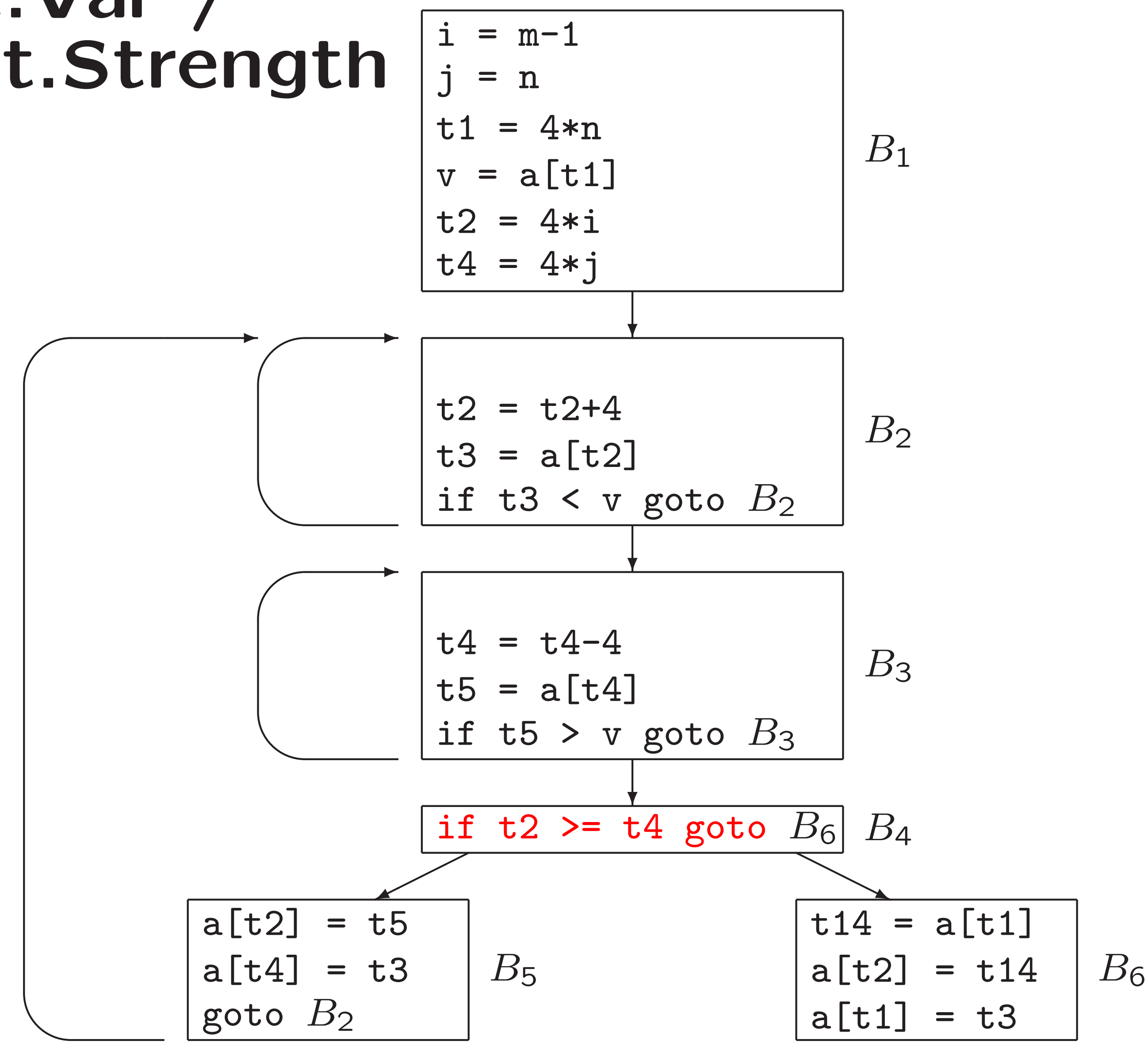
Induct.Var / Reduct.Strength



Induct.Var / Reduct.Strength



Induct.Var / Reduct.Strength



En verder. . .

- Maandag 18 november: inleveren opdracht 3
- Dinsdag 19 november: practicum over opdracht 4
- Eerst naar 403, daarna naar 302/304
- Inleveren 9 december
- Dinsdag 26 november: hoor-/werkcollege in 403
- Dinsdag 3 december: practicum over opdracht 4

Compiler constructie

college 8

Code Generation

Code Optimization

Chapters for reading:

8.5–8.5.4, 8.6–8.7, 8.8–8.8.2

9.intro, 9.1