BCSE307L – COMPILER DESIGN

TEXT BOOK:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Second Edition, Pearson Education Limited, 2014.

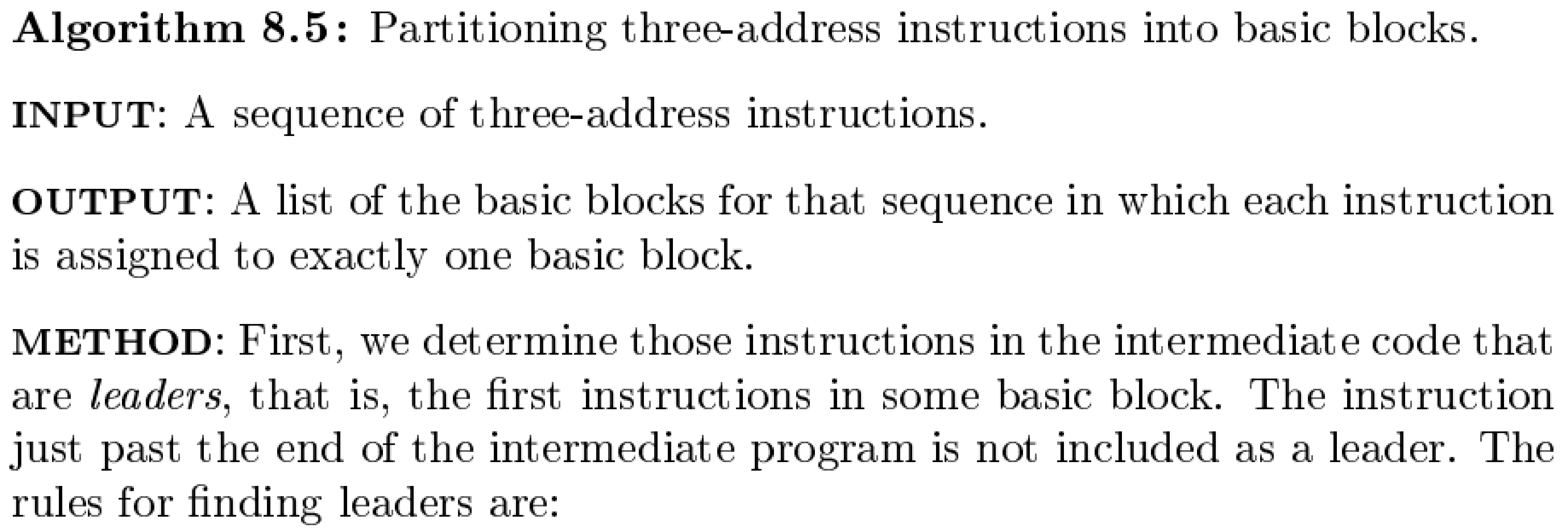
Module:5	CODE OPTIMIZATION	6 hours
Loop optimizations- Principal Sources of Optimization -Introduction to Data Flow Analysis - Basic Blocks - Optimization of Basic Blocks - Peephole Optimization- The DAG Representation of Basic Blocks -Loops in Flow Graphs - Machine Independent Optimization- Implementation of a naïve code generator for a virtual Machine- Security checking of virtual machine code.		

Basic Blocks and Flow Graphs

- Basic blocks
- Next-Use Information
- Flow Graphs
- Representation of Flow Graphs
- Loops

Basic Blocks and Flow Graphs

- Partition a sequence of three address instructions into basic blocks



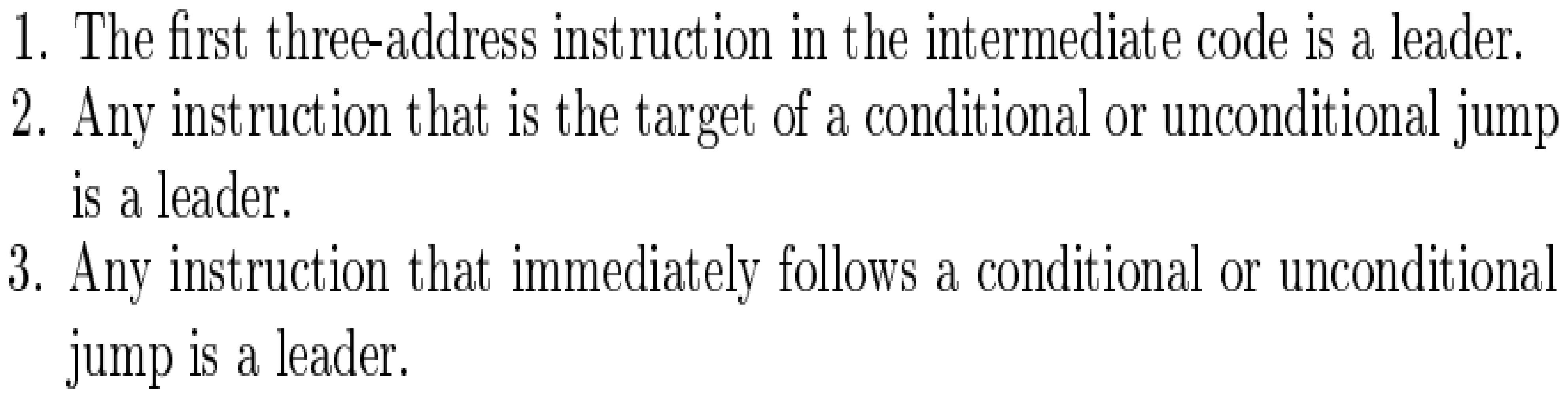
Algorithm 8.5: Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

Basic Blocks and Flow Graphs



1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Basic Blocks and Flow Graphs

Example 8.6: The intermediate code in Fig. 8.7 turns a 10×10 matrix **a** into an identity matrix. Although it is not important where this code comes from, it might be the translation of the pseudocode in Fig. 8.8. In generating the intermediate code, we have assumed that the real-valued array elements take 8 bytes each, and that the matrix **a** is stored in row-major form.

```
for $i$ from 1 to 10 do
    for $j$ from 1 to 10 do
        $a[i, j] = 0.0$;
for $i$ from 1 to 10 do
    $a[i, i] = 1.0$;
```

Figure 8.8: Source code for Fig. 8.7

Basic Blocks and Flow Graphs

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Figure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

Basic Blocks and Flow Graphs

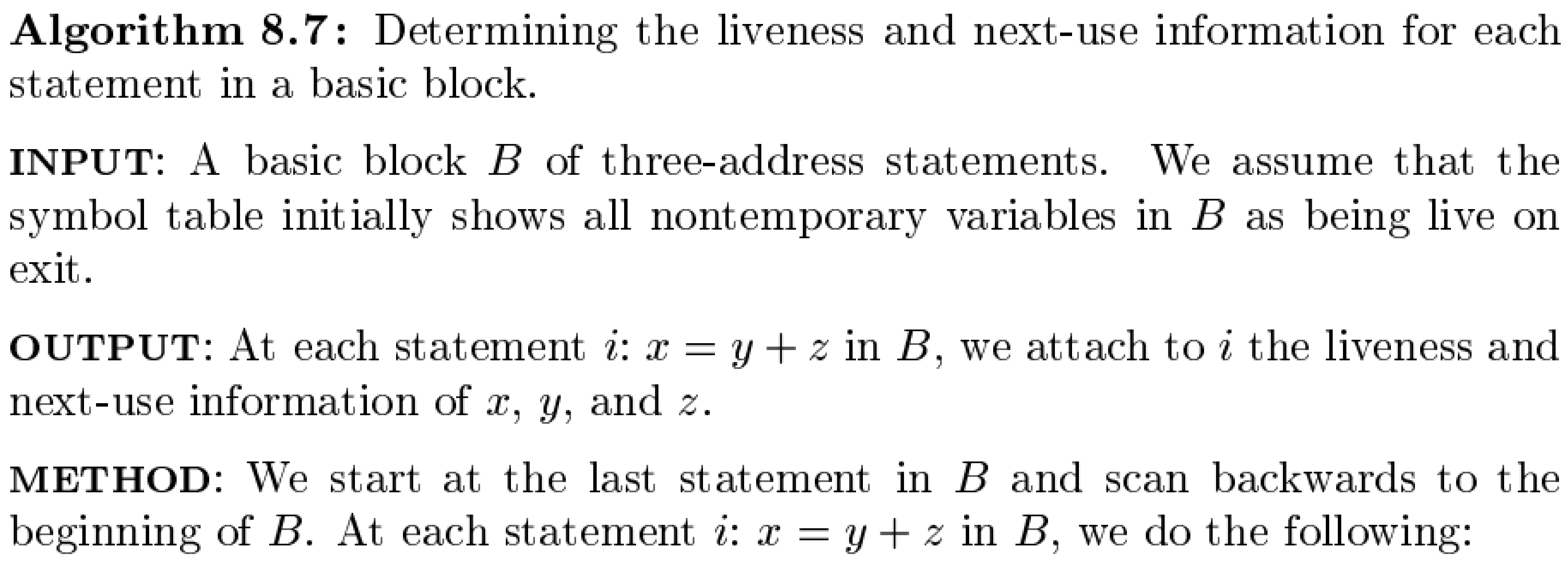
We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13. The basic block of each leader contains all the instructions from itself until just before the next leader. Thus, the basic block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12, and instruction 13's block is 13 through 17. □

Next-Use Information

The *use* of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j *uses* the value of x computed at statement i. We further say that x is *live* at statement i.

We wish to determine for each three-address statement $x = y + z$ what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.

Next-Use Information



Algorithm 8.7: Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

OUTPUT: At each statement i: $x = y + z$ in B, we attach to i the liveness and next-use information of x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: $x = y + z$ in B, we do the following:

Next-Use Information

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
2. In the symbol table, set x to "not live" and "no next use."
3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

Flow Graphs

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:

Flow Graphs

- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

We say that B is a *predecessor* of C, and C is a *successor* of B.

Flow Graphs

Example 8.8: The set of basic blocks constructed in Example 8.6 yields the flow graph of Fig. 8.9. The entry points to basic block B_1, since B_1 contains the first instruction of the program. The only successor of B_1 is B_2, because B_1 does not end in an unconditional jump, and the leader of B_2 immediately follows the end of B_1.

Flow Graphs

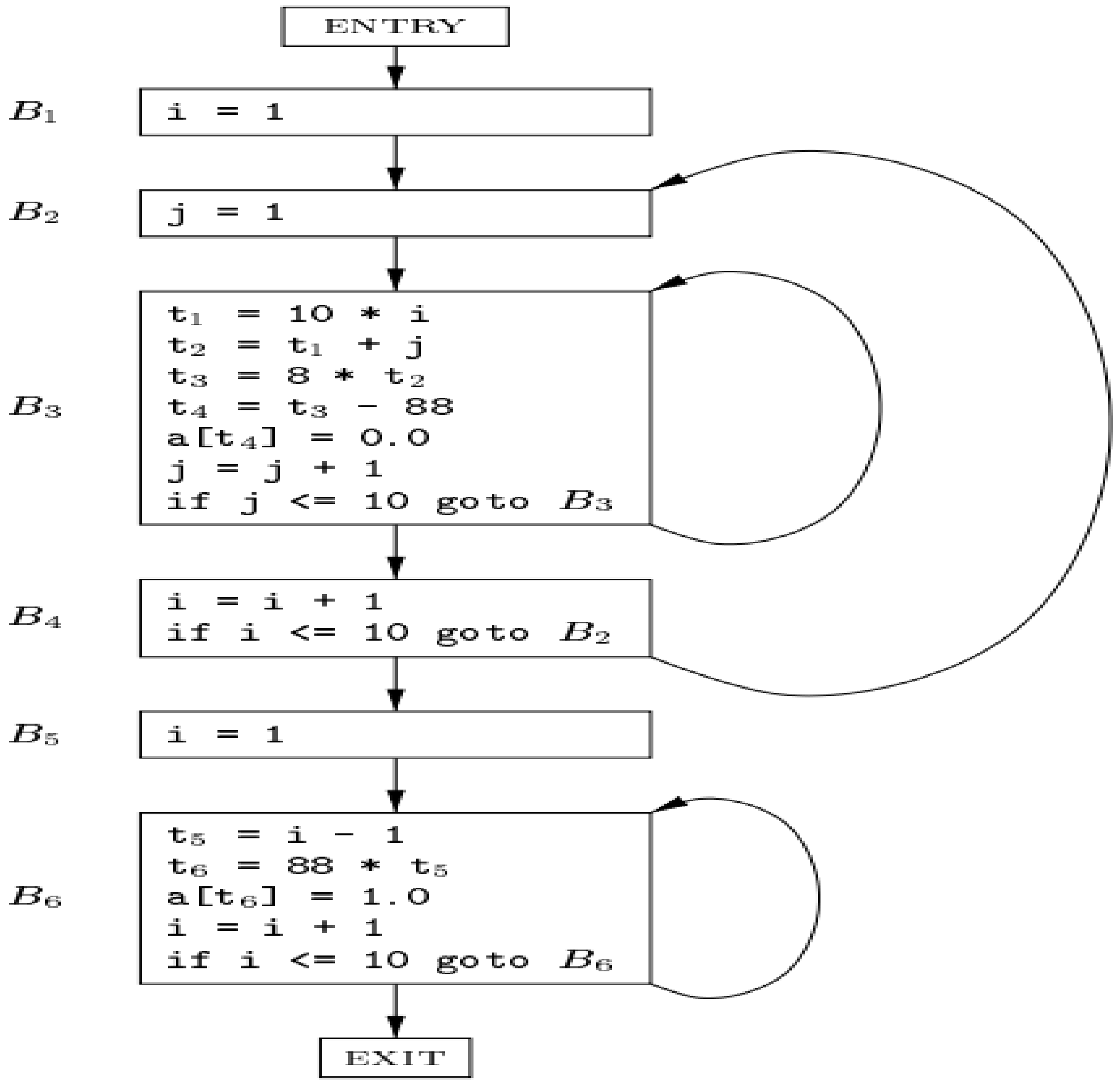
Block B_3 has two successors. One is itself, because the leader of B_3, instruction 3, is the target of the conditional jump at the end of B_3, instruction 9. The other successor is B_4, because control can fall through the conditional jump at the end of B_3 and next enter the leader of B_4.

Only B_6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B_6. $\square$

Representation of Flow Graphs

First, note from Fig. 8.9 that in the flow graph, it is normal to replace the jumps to instruction numbers or labels by jumps to basic blocks.

Representation of Flow Graphs



```
B1: i = 1

B2: j = 1

B3: t1 = 10 * i
    t2 = t1 + j
    t3 = 8 * t2
    t4 = t3 - 88
    a[t4] = 0.0
    j = j + 1
    if j <= 10 goto B3

B4: i = i + 1
    if i <= 10 goto B2

B5: i = 1

B6: t5 = i - 1
    t6 = 88 * t5
    a[t6] = 1.0
    i = i + 1
    if i <= 10 goto B6
```

Figure 8.9: Flow graph from Fig. 8.7

Loops

Many code transformations depend upon the identification of "loops" in a flow graph. We say that a set of nodes L in a flow graph is a *loop* if L contains a node e called the *loop entry*, such that:

1. e is not ENTRY, the entry of the entire flow graph.

2. No node in L besides e has a predecessor outside L. That is, every path from ENTRY to any node in L goes through e.

3. Every node in L has a nonempty path, completely within L, to e.

- The principal Sources of Optimization
- Optimization of Basic Blocks
- Peephole Optimization

Code optimization

- Code Improvement / code Optimization
    - Elimination of unnecessary instructions
    - Replacement of one sequence of instructions by a faster sequence of instructions
- Code optimization
    - Local code optimization – within a basic block
    - Global code optimization – across basic blocks
        - Data-flow analyses – algorithms to gather information about a program

The principal Sources of Optimization

- Causes of Redundancy
- A Running Example: Quicksort
- Semantics-Preserving Transformations
- Global Common Subexpressions
- Copy Propagation
- Dead Code Elimination
- Code Motion
- Induction variables and Reduction in Strength

The principal Sources of Optimization

A compiler optimization must preserve the semantics of the original program.

A compiler knows only how to apply relatively low-level semantic transformations, using general facts such as algebraic identities like $i + 0 = i$ or program semantics such as the fact that performing the same operation on the same values yields the same result.

Causes of Redundancy

There are many redundant operations in a typical program. Sometimes the redundancy is available at the source level.

As a program is compiled, each of these high-level data-structure accesses expands into a number of low-level arithmetic operations, such as the computation of the location of the (i, j)th element of a matrix A. Accesses to the same

A Running Example: Quicksort

In the following, we shall use a fragment of a sorting program called *quicksort* to illustrate several important code-improving transformations. The C program in Fig. 9.1 is derived from Sedgewick,[1] who discussed the hand-optimization of such a program. We shall not discuss all the subtle algorithmic aspects of this program here, for example, the fact that $a[0]$ must contain the smallest of the sorted elements, and $a[max]$ the largest.

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;
    /* fragment begins here */
    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] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Figure 9.1: C code for quicksort

A Running Example: Quicksort

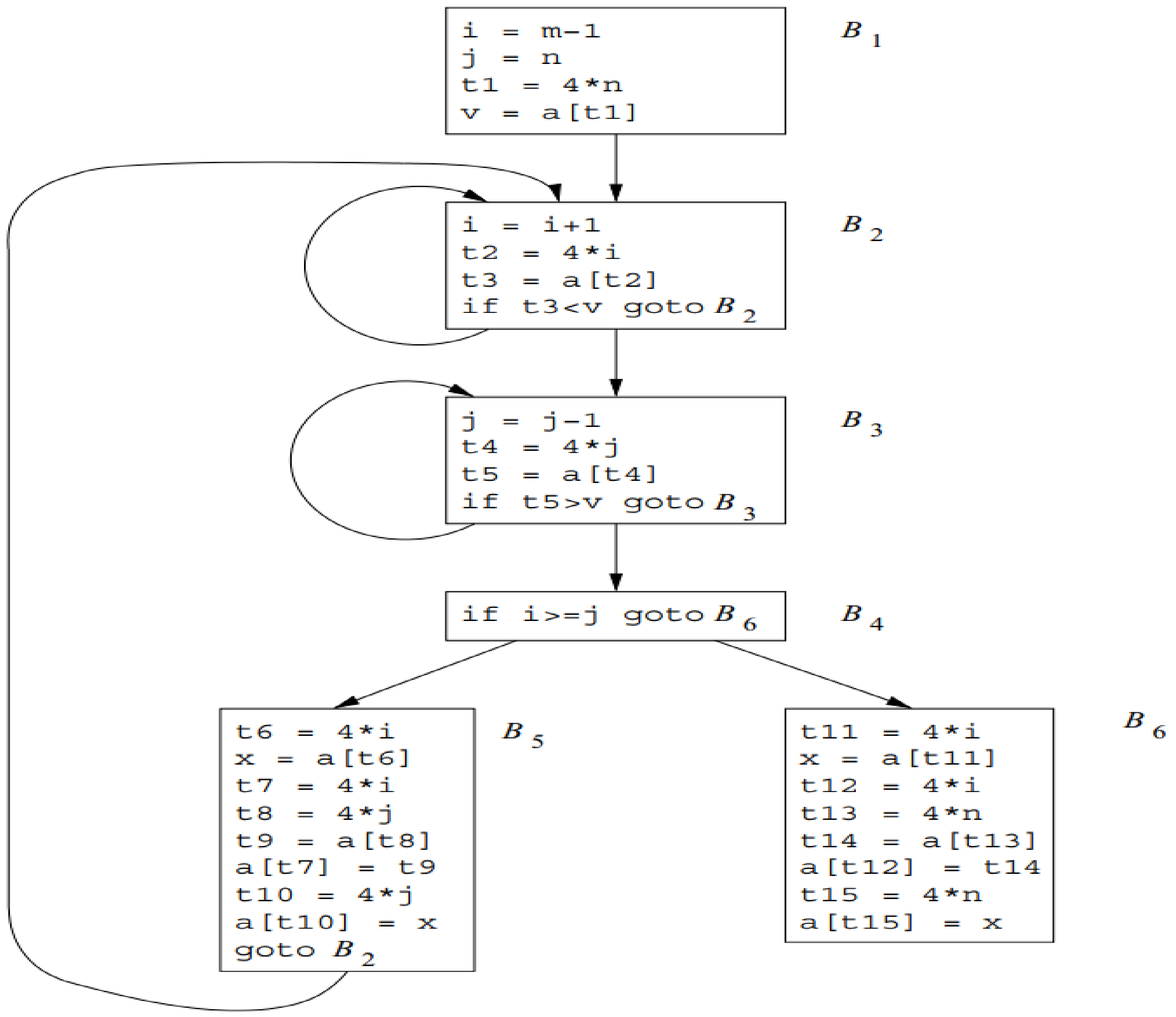
```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= 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] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Figure 9.1: C code for 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
```

Figure 9.2: Three-address code for fragment in Fig. 9.1

A Running Example: Quicksort



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

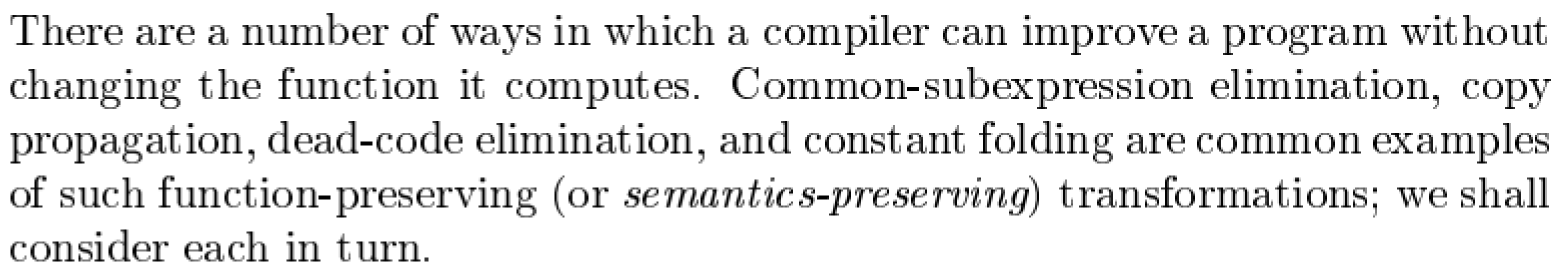
```
if i>=j goto B6
```

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

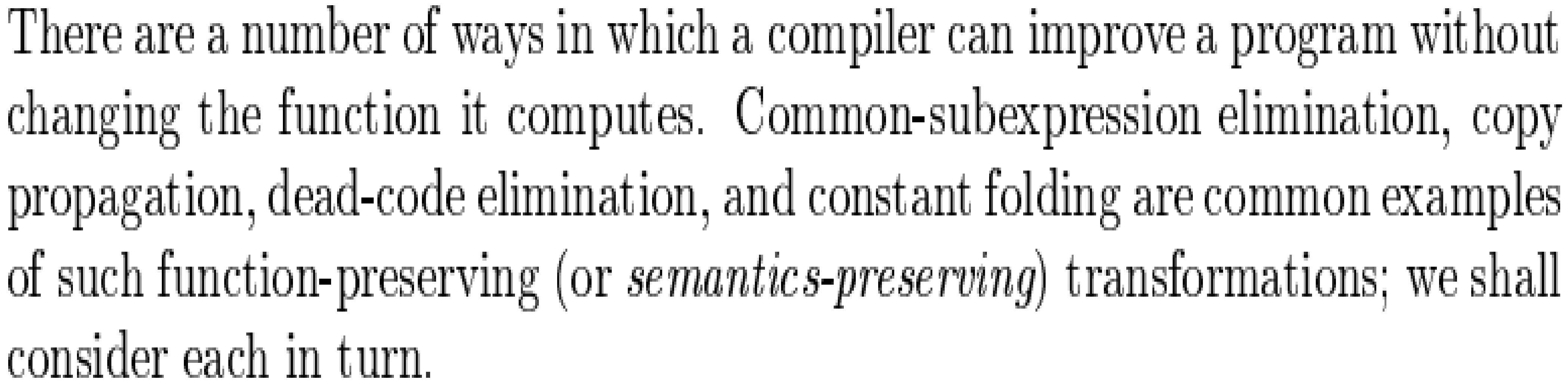
```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

Figure 9.3: Flow graph for the quicksort fragment

Semantics-Preserving Transformations

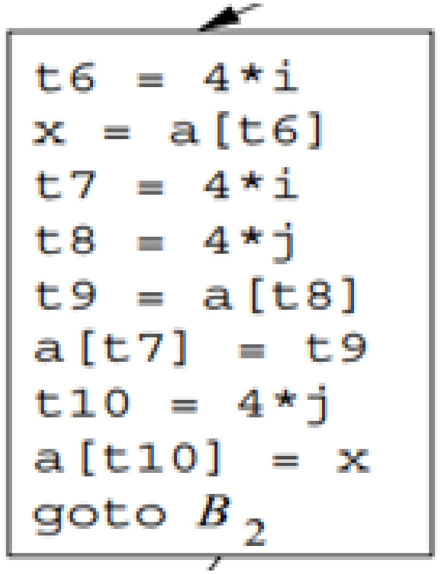

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common-subexpression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving (or *semantics-preserving*) transformations; we shall consider each in turn.

Semantics-Preserving Transformations



There are a number of ways in which a compiler can improve a program without changing the function it computes. Common-subexpression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving (or *semantics-preserving*) transformations; we shall consider each in turn.

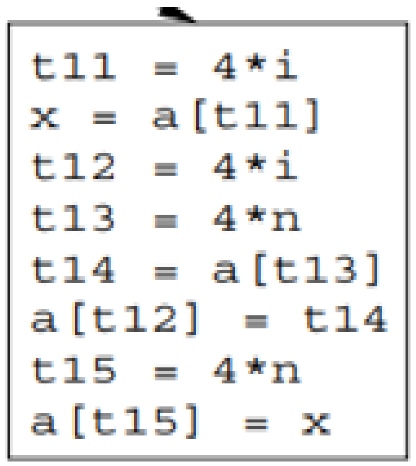
Local Common Subexpressions



```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

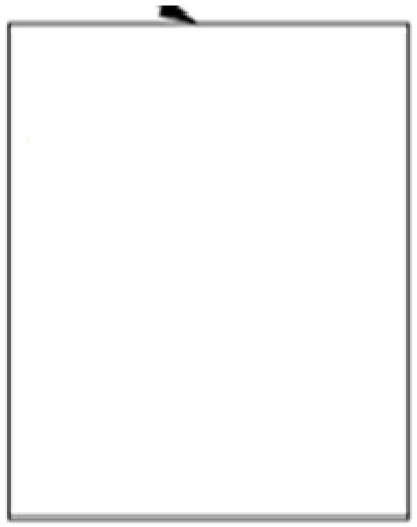
B_5

B_5



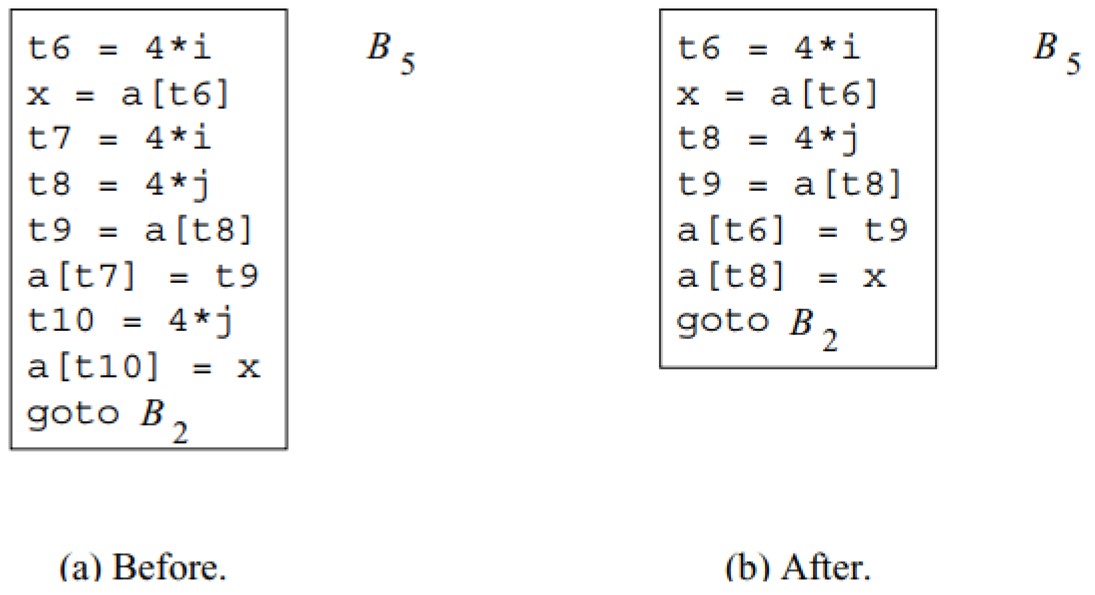
```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

B_6



B_6

Local Common Subexpressions



```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

B_5

(a) Before.

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

B_5

(b) After.

Figure 9.4: Local common-subexpression elimination

Local Common Subexpressions

```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

B_6

(a) Before.

```
t11 = 4 * i

x = a[t11]

t13 = 4 * n

t14 = a[t13]

a[t11] = t14

a[13] = x
```

B_6

(b) After.

Local common-subexpression elimination

Global Common Subexpressions

An occurrence of an expression E is called a *common subexpression* if E was previously computed and the values of the variables in E have not changed since the previous computation. We avoid recomputing E if we can use its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.[2]

Example 9.1: The assignments to `t7` and `t10` in Fig. 9.4(a) compute the common subexpressions $4 * i$ and $4 * j$, respectively. These steps have been eliminated in Fig. 9.4(b), which uses `t6` instead of `t7` and `t8` instead of `t10`.

Global Common Subexpressions

Example 9.2 : Figure 9.5 shows the result of eliminating both global and local common subexpressions from blocks B_5 and B_6 in the flow graph of Fig. 9.3. We first discuss the transformation of B_5 and then mention some subtleties involving arrays.

After local common subexpressions are eliminated, B_5 still evaluates $4*i$ and $4*j$, as shown in Fig. 9.4(b). Both are common subexpressions; in particular, the three statements

Global Common Subexpressions

B_5

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

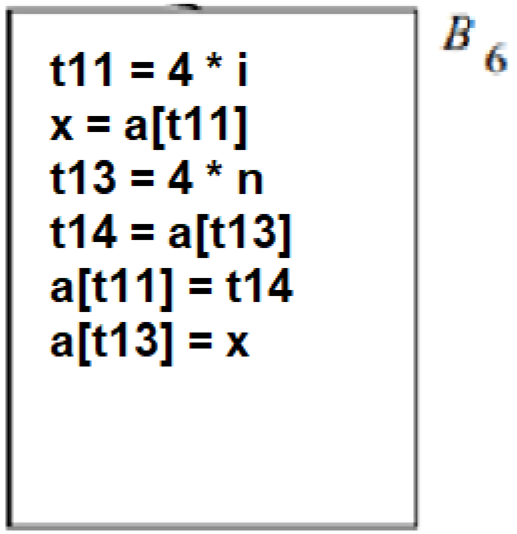
(a) Before.

B_5

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

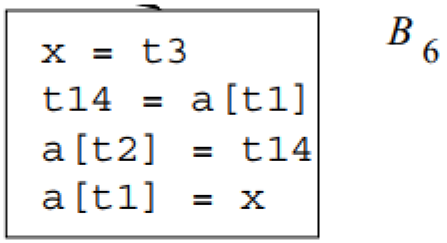
(b) After.

Global Common Subexpressions



```
t11 = 4 * i
x = a[t11]
t13 = 4 * n
t14 = a[t13]
a[t11] = t14
a[t13] = x
```

(a) Before.



```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

(b) After.

Global Common Subexpressions

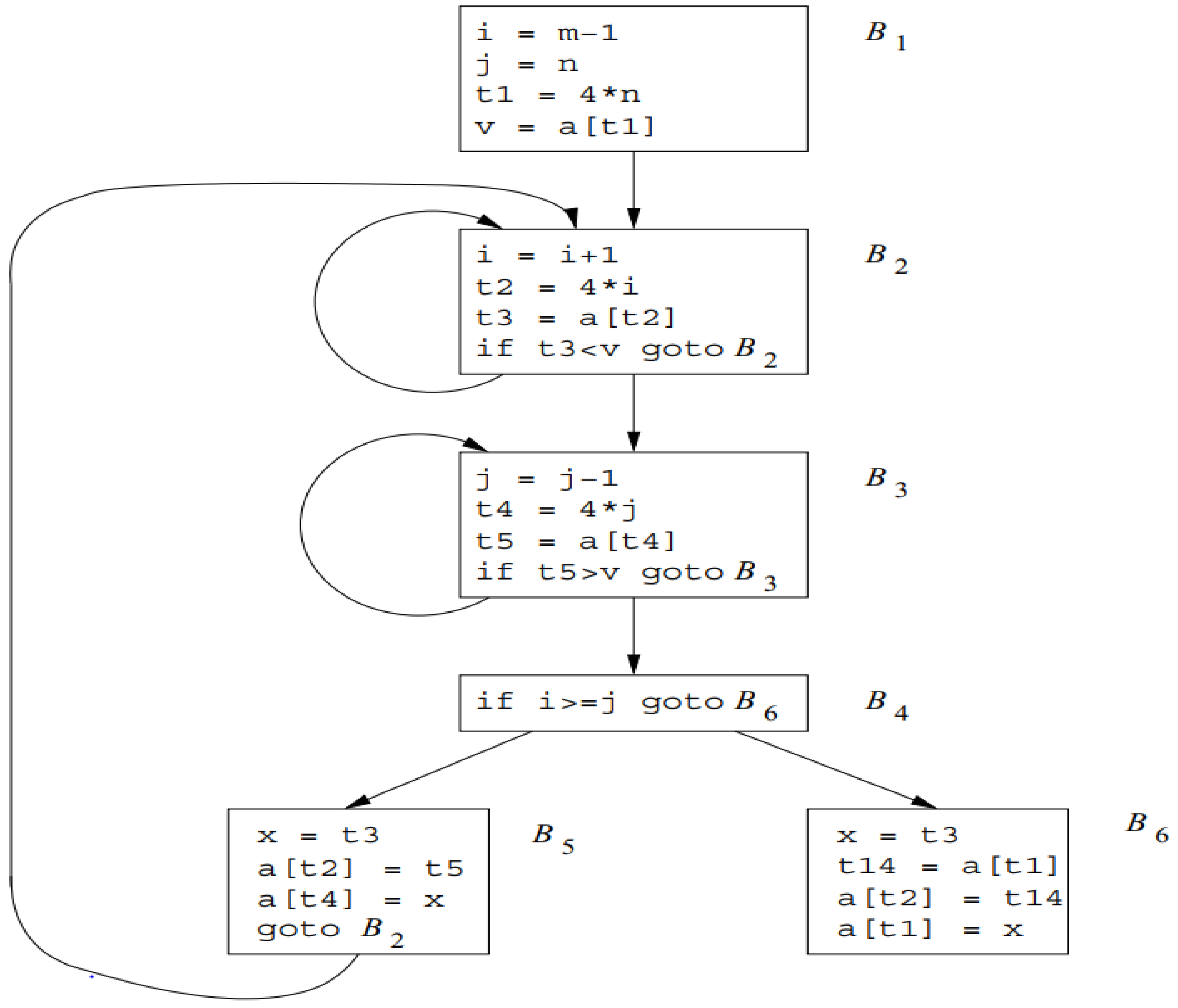


Figure 9.5: B_5 and B_6 after common-subexpression elimination

Copy Propagation

Block B_5 in Fig. 9.5 can be further improved by eliminating x, using two new transformations. One concerns assignments of the form `u = v` called *copy statements*, or *copies* for short. Had we gone into more detail in Example 9.2, copies would have arisen much sooner, because the normal algorithm for eliminating common subexpressions introduces them, as do several other algorithms.

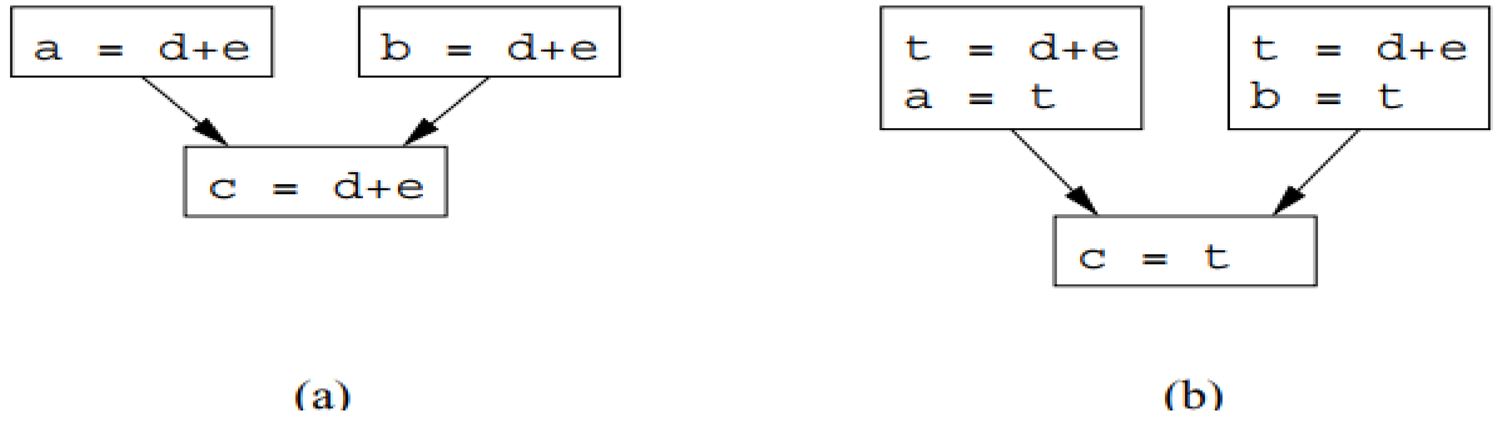
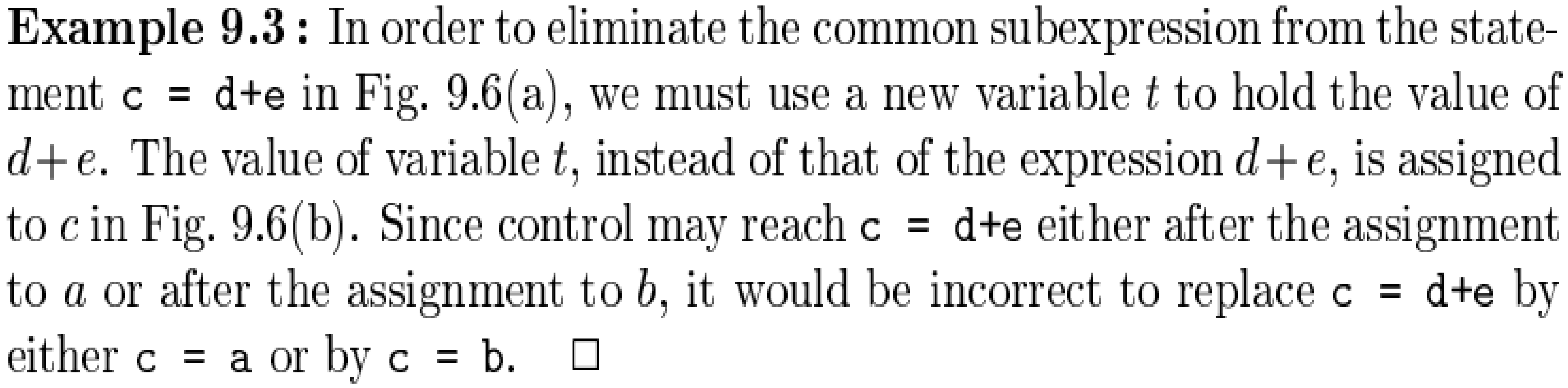


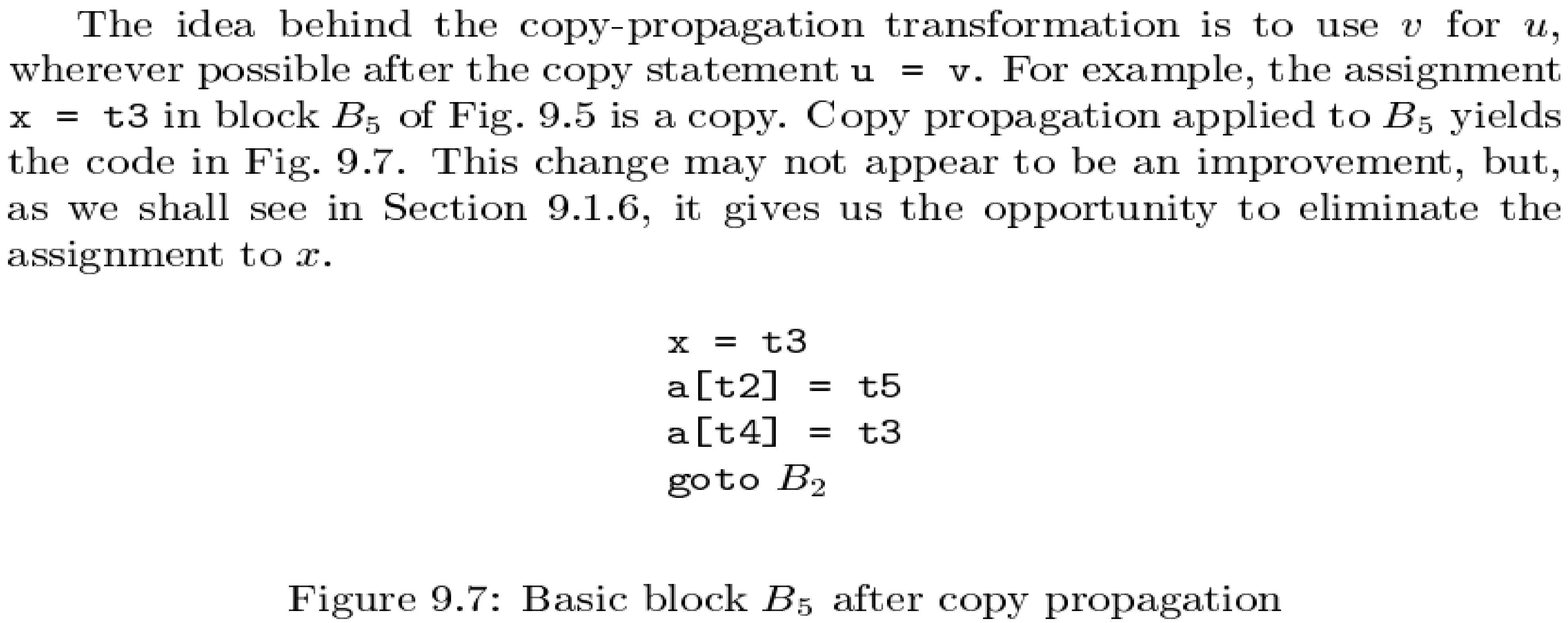
Figure 9.6: Copies introduced during common subexpression elimination

Copy Propagation



Example 9.3 : In order to eliminate the common subexpression from the statement `c = d+e` in Fig. 9.6(a), we must use a new variable t to hold the value of $d+e$. The value of variable t, instead of that of the expression $d+e$, is assigned to c in Fig. 9.6(b). Since control may reach `c = d+e` either after the assignment to a or after the assignment to b, it would be incorrect to replace `c = d+e` by either `c = a` or by `c = b`. □

Copy Propagation

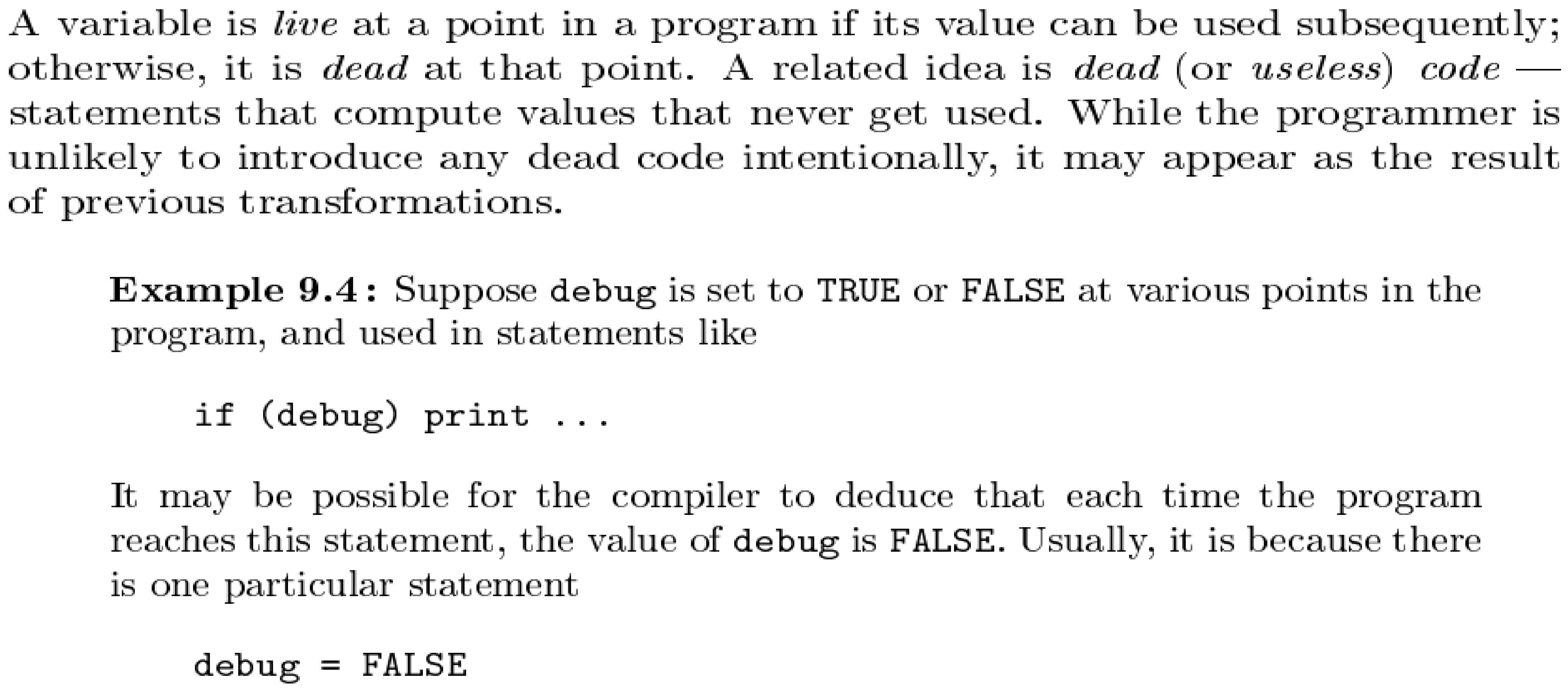


The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement `u = v`. For example, the assignment `x = t3` in block B_5 of Fig. 9.5 is a copy. Copy propagation applied to B_5 yields the code in Fig. 9.7. This change may not appear to be an improvement, but, as we shall see in Section 9.1.6, it gives us the opportunity to eliminate the assignment to x.

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Figure 9.7: Basic block B_5 after copy propagation

Dead Code Elimination



A variable is *live* at a point in a program if its value can be used subsequently; otherwise, it is *dead* at that point. A related idea is *dead* (or *useless*) *code* — statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Example 9.4: Suppose `debug` is set to `TRUE` or `FALSE` at various points in the program, and used in statements like

```
if (debug) print ...
```

It may be possible for the compiler to deduce that each time the program reaches this statement, the value of `debug` is `FALSE`. Usually, it is because there is one particular statement

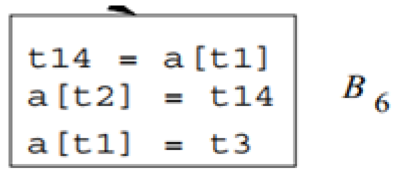
```
debug = FALSE
```

Dead Code Elimination

One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to x and transforms the code in Fig 9.7 into

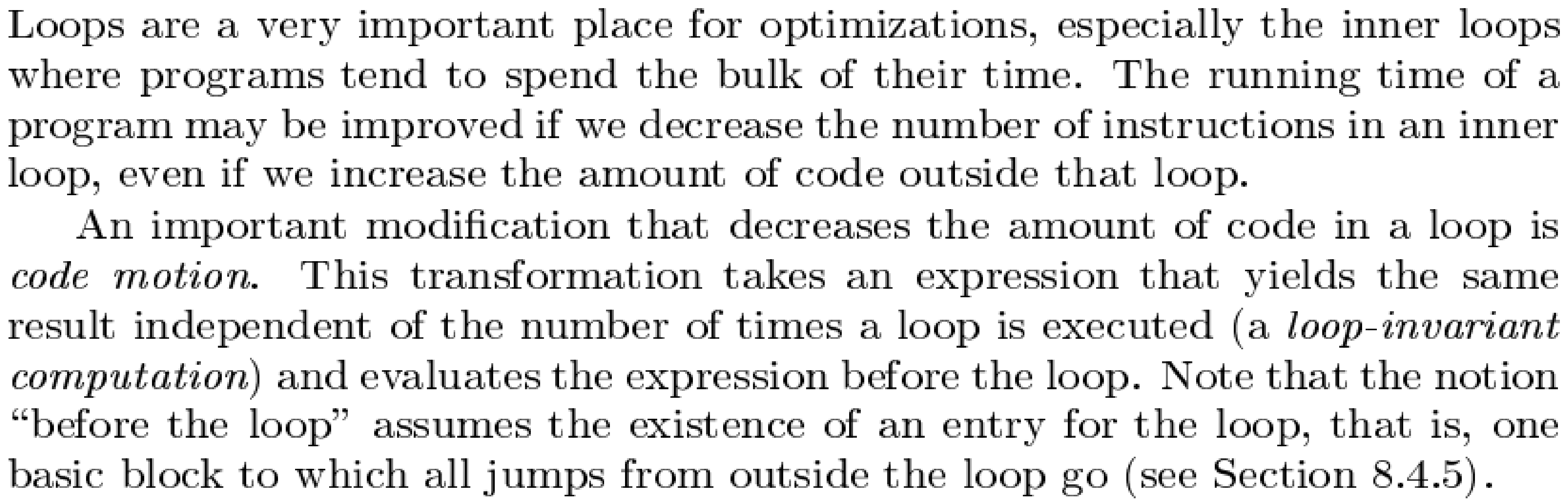
```
a[t2] = t5
a[t4] = t3
goto B2
```

This code is a further improvement of block B_5 in Fig. 9.5.



```
t14 = a[t1]
a[t2] = t14
a[t1] = t3
```

Code Motion



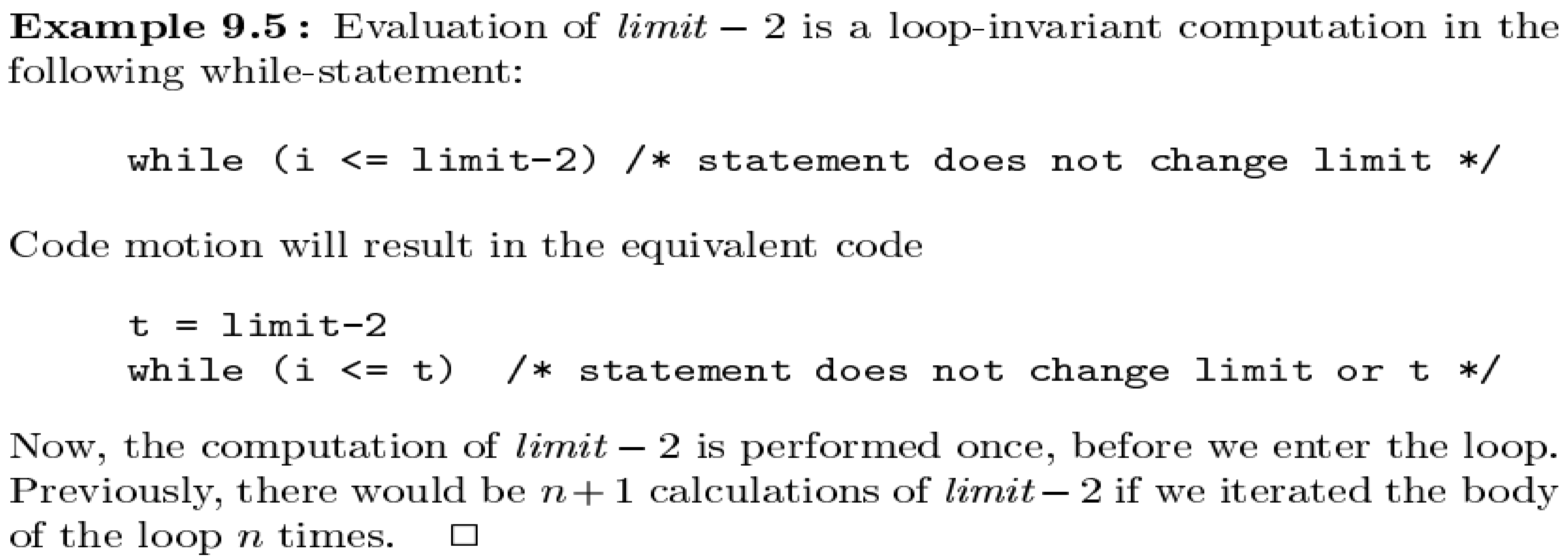
Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

An important modification that decreases the amount of code in a loop is *code motion*. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a *loop-invariant computation*) and evaluates the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop, that is, one basic block to which all jumps from outside the loop go (see Section 8.4.5).

Induction variables and Reduction in Strength

Another important optimization is to find induction variables in loops and optimize their computation. A variable x is said to be an "induction variable" if there is a positive or negative constant c such that each time x is assigned, its value increases by c. For instance, i and $t2$ are induction variables in the loop containing B_2 of Fig. 9.5. Induction variables can be computed with a single increment (addition or subtraction) per loop iteration. The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as *strength reduction*. But induction variables not only allow us sometimes to perform a strength reduction; often it is possible to eliminate all but one of a group of induction variables whose values remain in lock step as we go around the loop.

Code Motion



Example 9.5 : Evaluation of $limit - 2$ is a loop-invariant computation in the following while-statement:

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

Code motion will result in the equivalent code

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

Now, the computation of $limit - 2$ is performed once, before we enter the loop. Previously, there would be $n+1$ calculations of $limit - 2$ if we iterated the body of the loop n times. □

Induction variables and Reduction in Strength

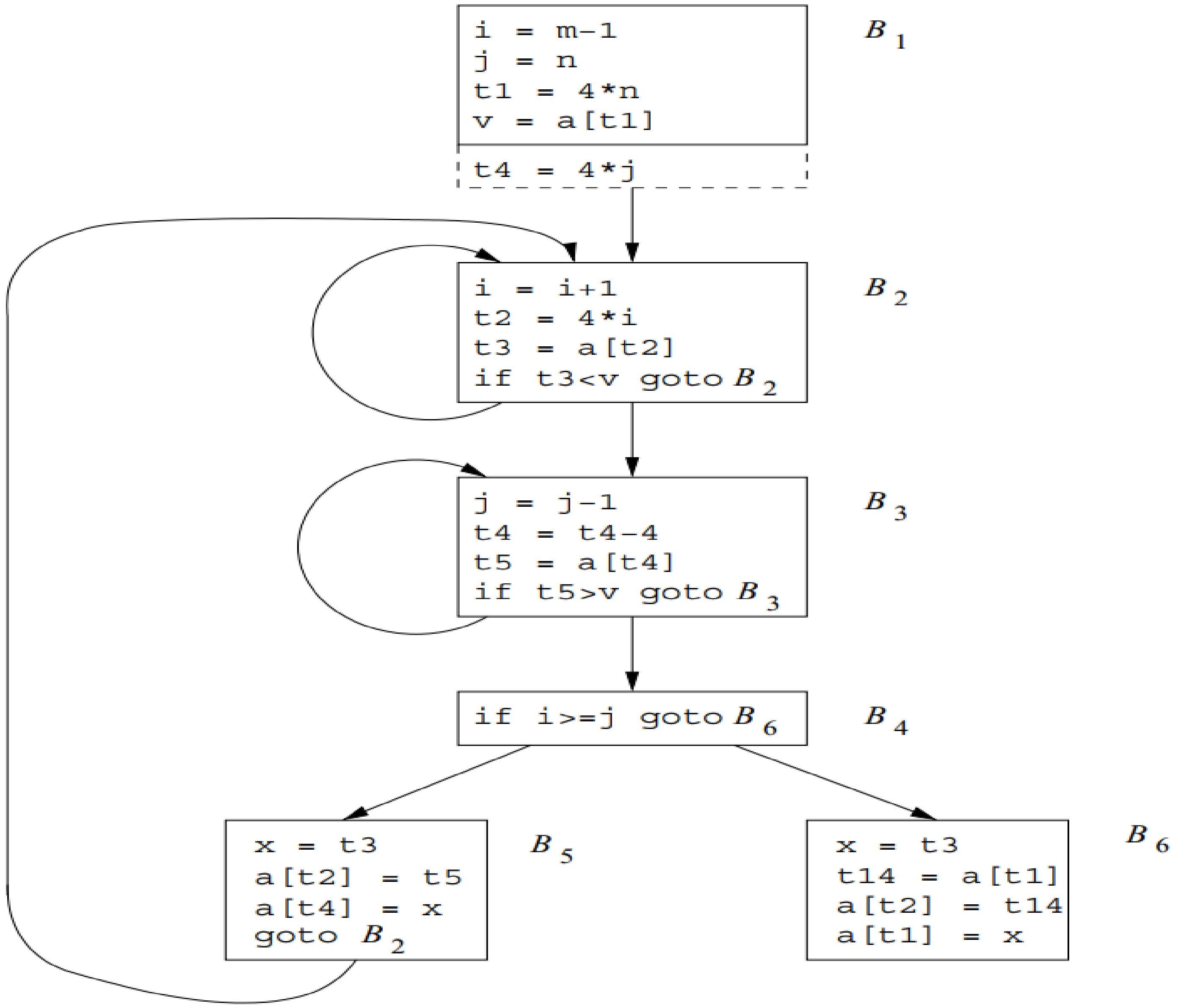
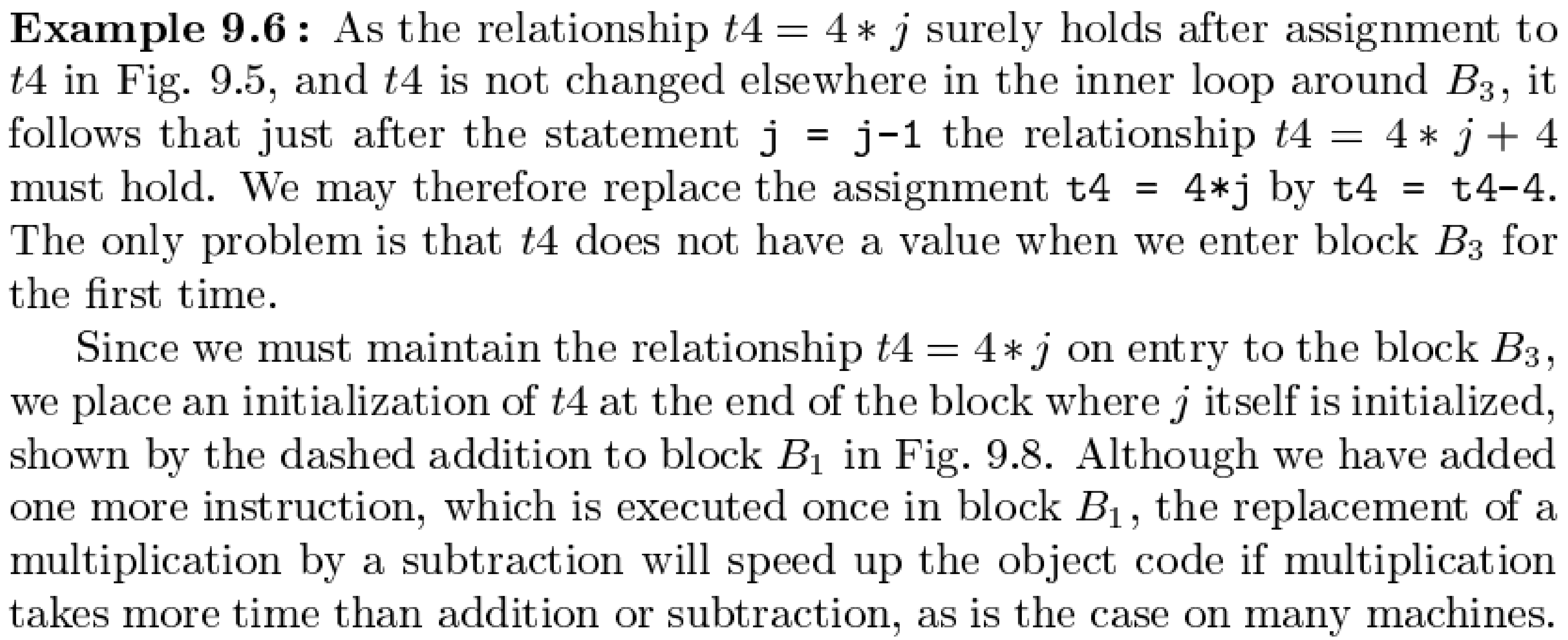


Figure 9.8: Strength reduction applied to $4 * j$ in block B_3

Induction variables and Reduction in Strength



Example 9.6 : As the relationship $t4 = 4 * j$ surely holds after assignment to $t4$ in Fig. 9.5, and $t4$ is not changed elsewhere in the inner loop around B_3, it follows that just after the statement `j = j-1` the relationship $t4 = 4 * j + 4$ must hold. We may therefore replace the assignment `t4 = 4*j` by `t4 = t4-4`. The only problem is that $t4$ does not have a value when we enter block B_3 for the first time.

Since we must maintain the relationship $t4 = 4 * j$ on entry to the block B_3, we place an initialization of $t4$ at the end of the block where j itself is initialized, shown by the dashed addition to block B_1 in Fig. 9.8. Although we have added one more instruction, which is executed once in block B_1, the replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Induction variables and Reduction in Strength

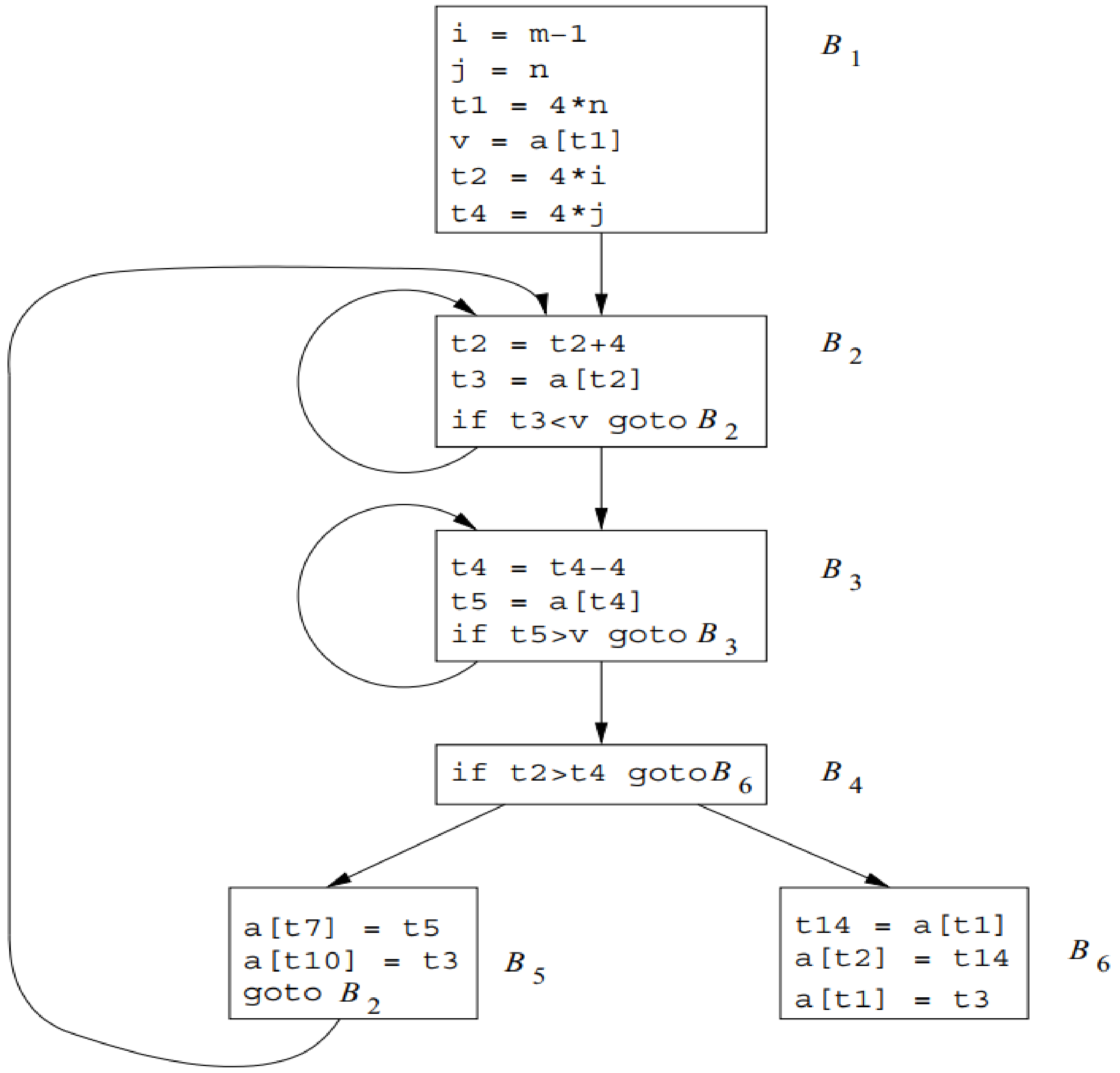


Figure 9.9: Flow graph after induction-variable elimination