

Code Optimization

Jordi Cortadella

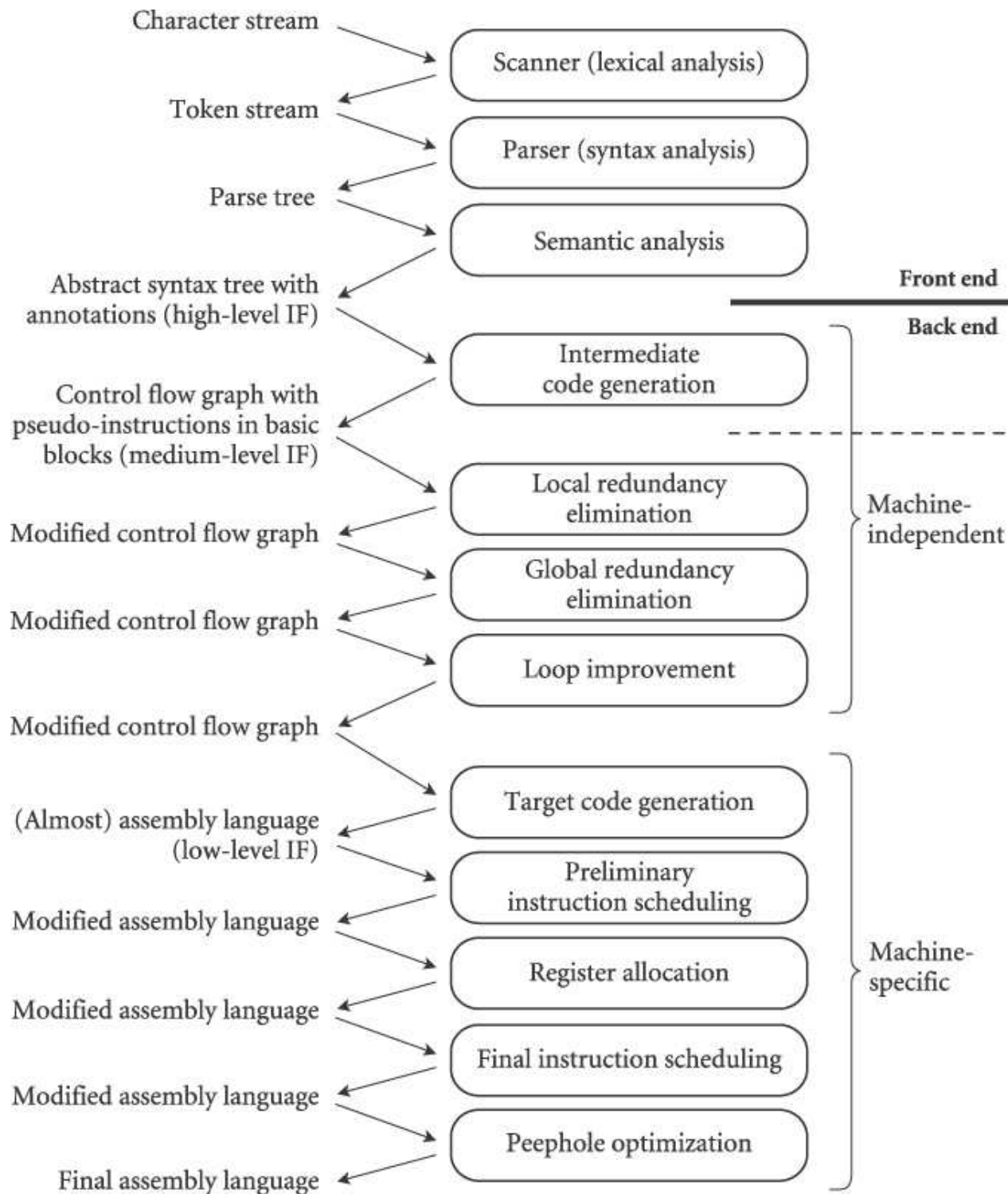
(material mostly taken from Aho, Lam, Sethi, Ullman's book)

Code optimization

A naive translation from high-level languages produces many inefficiencies.

Code optimization:

- Elimination of unnecessary instructions or replacement of a sequence of instructions by a faster sequence.
- Phases:
 - Machine-independent optimizations
 - Machine-dependent optimizations

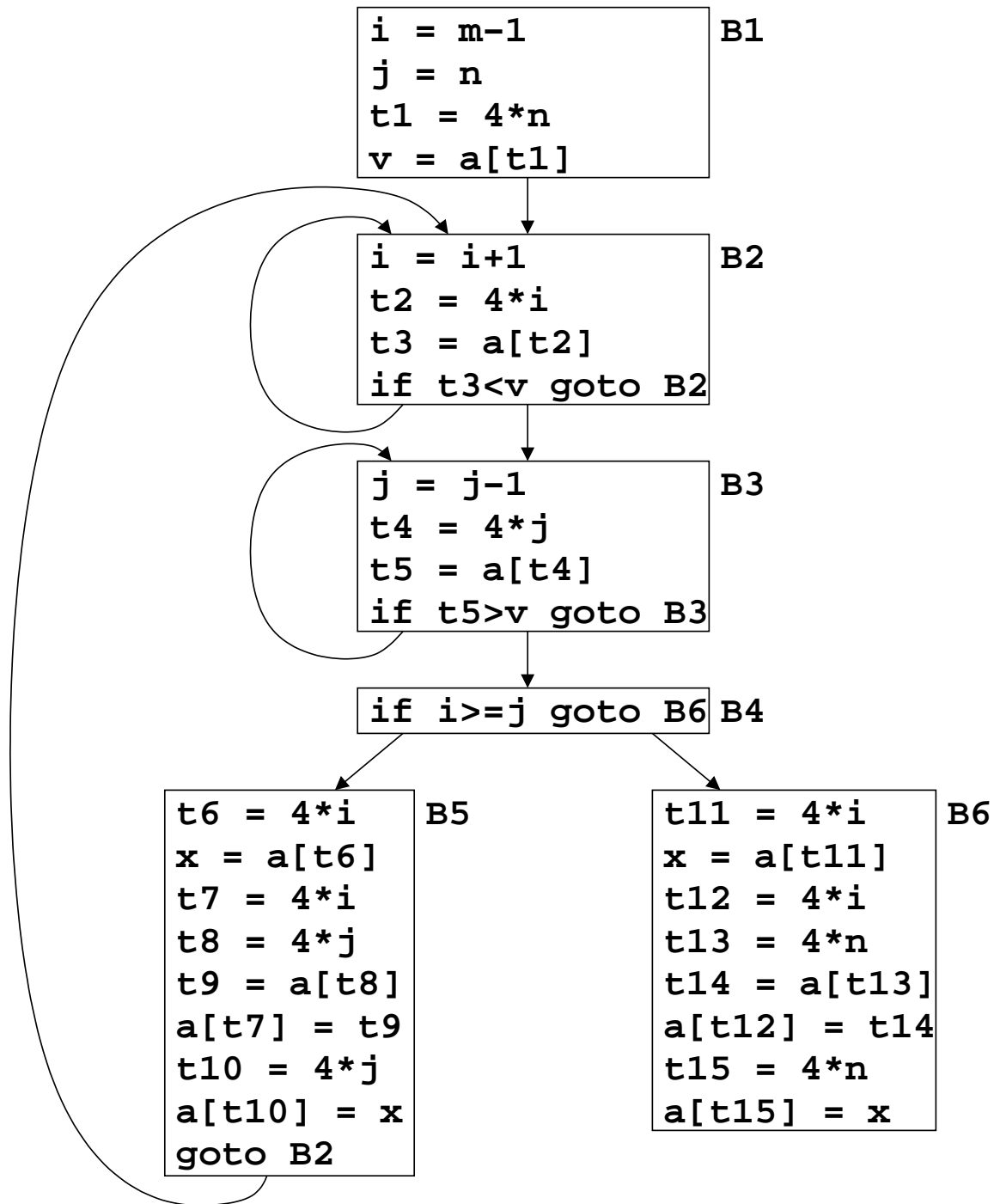


Overview (by example)

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

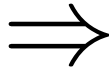
3-address code

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

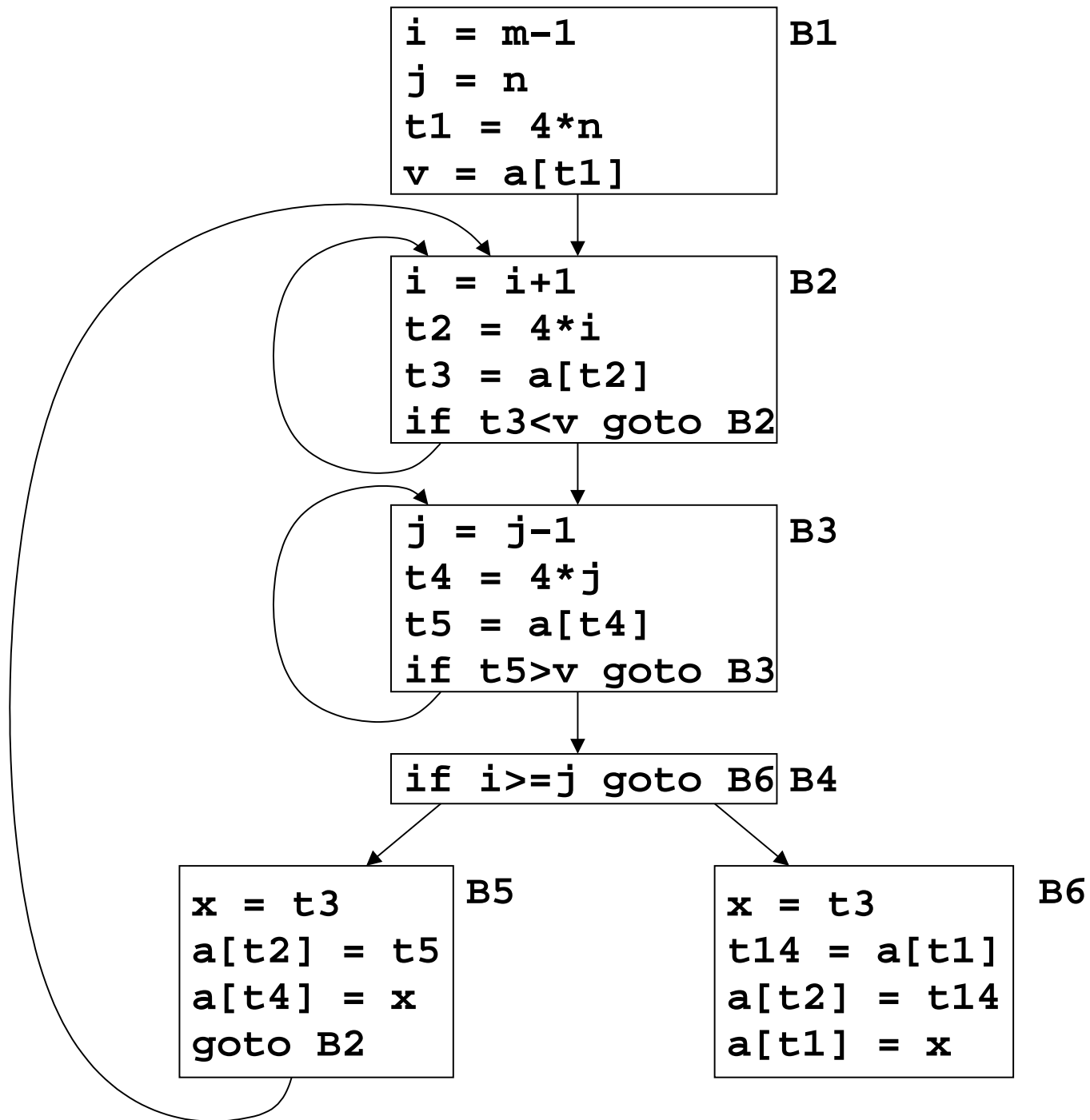


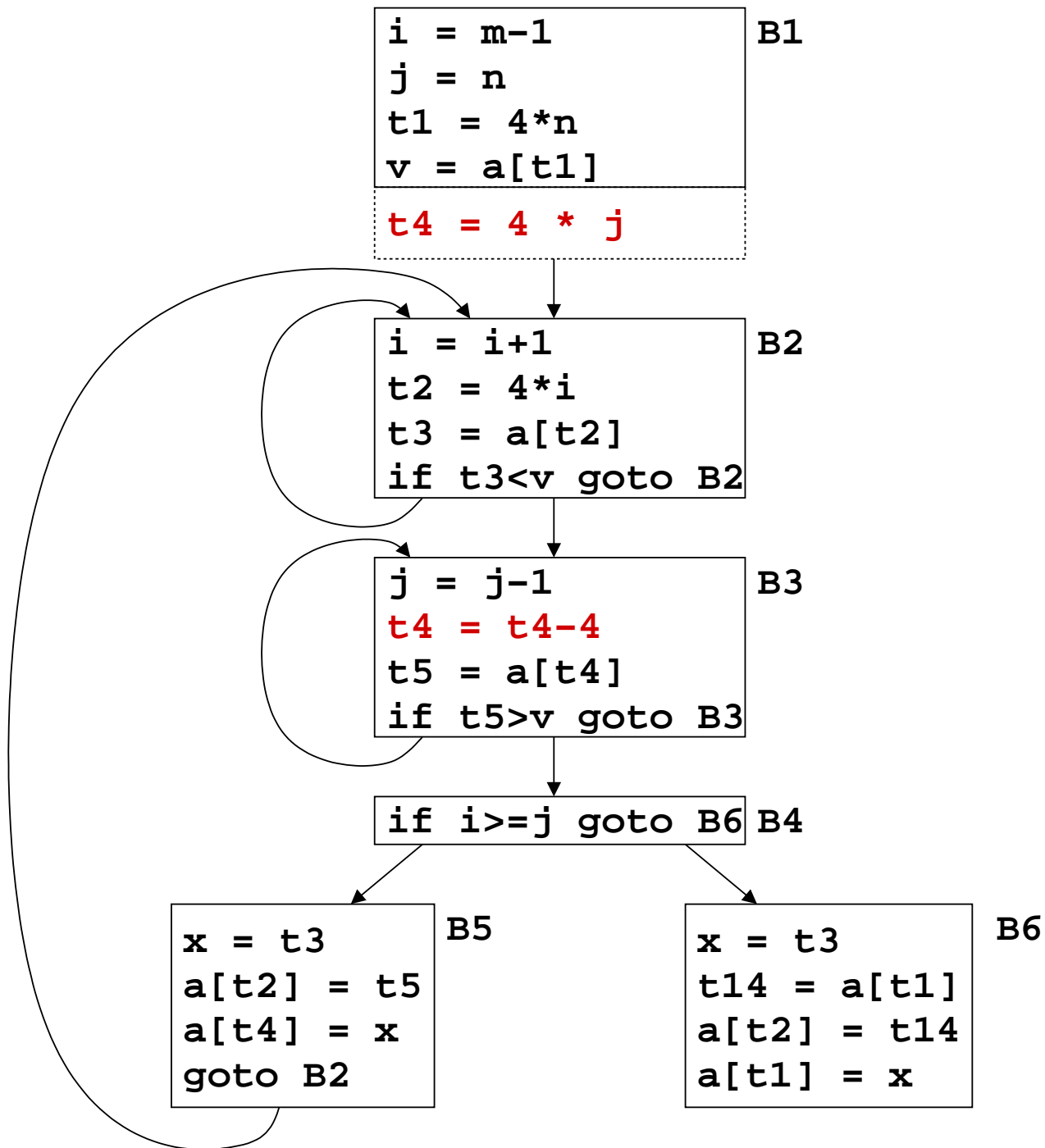
Local common-subexpression elimination

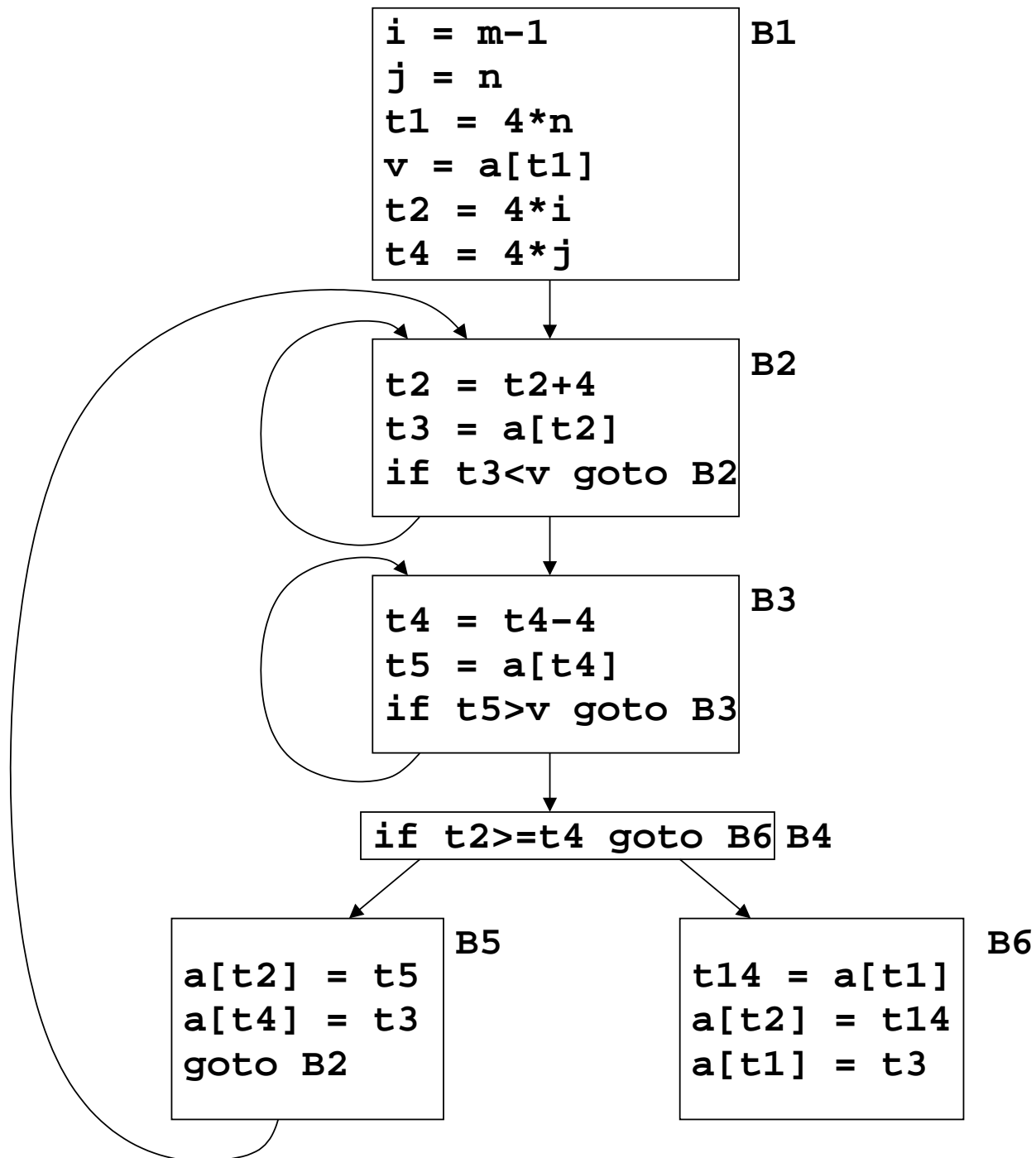
```
t6 = 4*i  
x = a[t6]  
t7 = 4*i  
t8 = 4*j  
t9 = a[t8]  
a[t7] = t9  
t10 = 4*j  
a[t10] = x  
goto B2
```



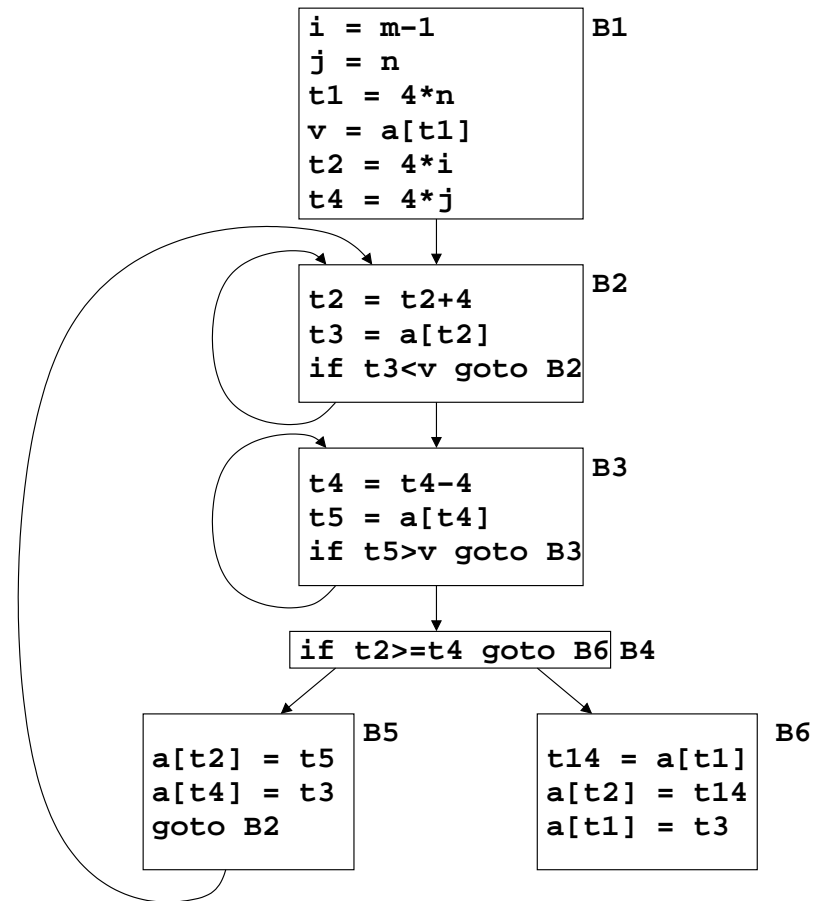
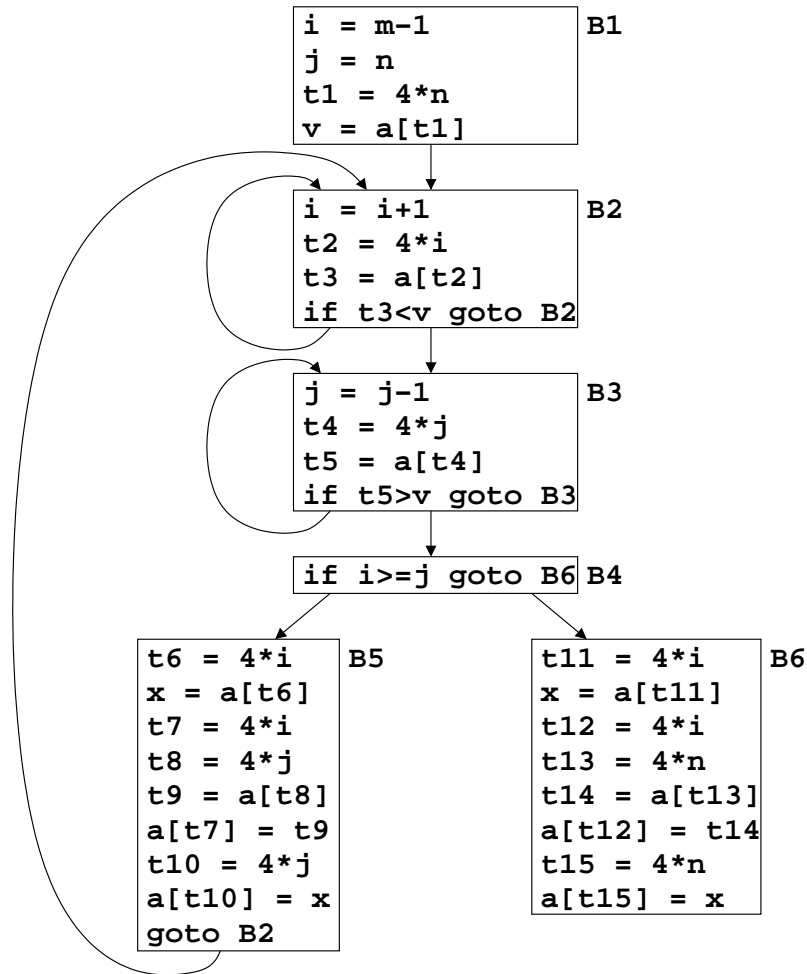
```
t6 = 4*i  
x = a[t6]  
t8 = 4*j  
t9 = a[t8]  
a[t6] = t9  
a[t8] = x  
goto B2
```







Impact of optimization



Local optimizations

Basic blocks

A **basic block** is a maximal sequence of consecutive instructions such that:

- The flow of control can only enter the basic block through the first instruction in the block.
- Inside the basic block, the flow of control can only exit the basic block through the last instruction in the block.

Leaders of basic blocks:

- The first instruction of the intermediate code.
- Targets of conditional or unconditional jumps.
- Instructions that follow a jump.

```
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;
```

- (1) is leader
(initial instruction)
- (2), (3) and (13) are leaders
(target of jumps)
- (10) and (12) are leaders
(follow a jump)

```
(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)
```

DAG representation of BBs

- One node for each initial value of the variables involved in the BB.
- One node for each statement. The children of the node are the nodes that represent the last definitions of the operands.
- Each node is labelled with the operator and the list of variables for which it is the last definition within the BB.
- The nodes whose variables are *live on exit* are declared as *output nodes*. Calculation of live variables is done by the global analysis.

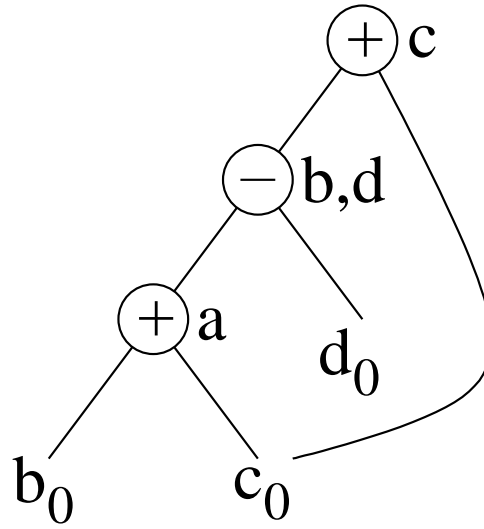
Local optimizations

a = b + c

b = a - d

c = b + c

d = a - d



a = b + c

d = a - d

c = d + c

(assumes **b** is not
live on exit)

Finding **local common subexpressions**:

- Before the creation of a new node, check that there is an existing node with the same children.

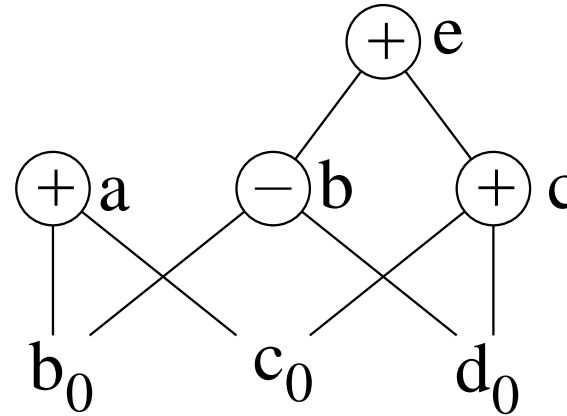
Local optimizations

$a = b + c$

$b = b - d$

$c = c + d$

$e = b + c$



Local optimization will miss the fact that " $a=b+c$ " and " $e=b+c$ " compute the same value.

Dead code elimination:

- Remove any root without any live variable attached.
- Repeat until no more nodes can be removed.

Example: if c and e are not live, the nodes with label e and c can be removed one after the other. The nodes with labels a and b will remain.

Algebraic identities

$$x + 0 = 0 + x = x$$

EXPENSIVE

CHEAPER

$$x \times 1 = 1 \times x = x$$

$$x^2 = x \times x$$

$$x - 0 = x$$

$$2 \times x = x + x$$

$$x/1 = x$$

$$x/2 = x \times 0.5$$

Constant folding: $2 * 3.14$ replaced by 6.28

Commutativity: $x * y = y * x$

Associativity:

$$a = b + c$$

$$a = b + c$$

$$e = c + d + b$$

$$e = a + d$$

Not all rearrangements of computations are permitted (check the language reference manual). For example, some rearrangements may produce undesired overflows.

Array references

$x = a[i]$

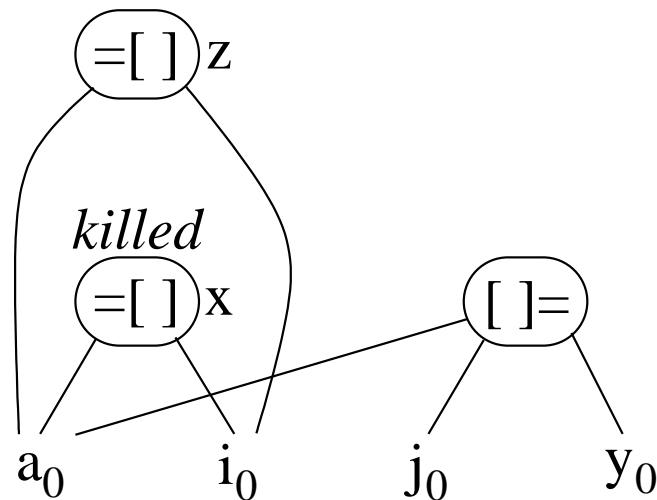
$a[j] = y$

$z = a[i]$

$a[i]$ is not a common subexpression,
since i and j might have the same value.

Use the $=[]$ and $[]=$ operators to represent assignments from and to arrays, respectively.

The creation of a $[]=$ node *kills* the currently constructed nodes whose value depends on the same array.



Pointers and procedure calls

$x = *p$

$*q = y$

$=*p$ must consider all possible variables associated to p and $*q =$ must kill all other nodes created in the DAG.

Pointer analysis is complex. Global pointer analysis may limit the set of variables a pointer could reference at a given point.

Procedure calls that can change global variables can be considered as assignments through pointers.

Analysis of pointers and procedure calls must always be conservative.

Flow-of-Control optimizations

Jumps to jumps, jumps to conditional jumps and conditional jumps to jumps can be optimized.

		goto L1
		...
goto L1	if a < b goto L1	goto L4
...	...	L1: if a < b goto L2
L1: goto L2	L1: goto L2	L3:
⇓	⇓	⇓
goto L2	if a < b goto L2	if a < b goto L2
...	...	goto L3
L1: goto L2	L1: goto L2	...
		goto L4
		L3:

Eliminating unreachable code

Unlabeled instructions following an unconditional branch can be removed. The following code

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

can be transformed into the following one by removing jumps over jumps:

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

Eliminating unreachable code

In case `debug` is set to 0, constant propagation would transform the code into

```
    if 0 != 1 goto L2
L1: print debugging information
L2: ...
```

Since the condition of the expression always evaluates to true, the conditional jump can be substituted by `goto L2`. Then, all the statements that print debugging information are unreachable and can be eliminated.

Data-Flow Analysis

Data-Flow Analysis

Set of techniques that derive information about the flow of data along program execution paths.

State: The values of all the variables in the program.

Program point: Location within the program.

- Within one BB, the program point after a statement is the same as the program point before the next statement.
- If there is an edge $B_1 \rightarrow B_2$, then the program point after the last statement of B_1 may be immediately followed by the program point before the first statement of B_2 .

Path: Sequence of program points.

Data-Flow Analysis Schema

With every program point, a **data-flow value** is associated that represents an abstraction of the set of all possible program states that can be observed at that point.

The domain of data-flow values will depend on the type of analysis to be performed (e.g. set of live variables).

For every statement s , the data-flow values before and after the statement are denoted by $IN[s]$ and $OUT[s]$, respectively.

Data-Flow Problem: Find a solution for all $IN[s]$ and $OUT[s]$ such that the constraints determined by the semantics of the statements and the flow of control are met.

Transfer functions

The relationship between $\text{IN}[s]$ and $\text{OUT}[s]$ is known as the **transfer function**.

Transfer functions:

- For **forward** propagation:

$$\text{OUT}[s] = f_s(\text{IN}[s])$$

- For **backward** propagation:

$$\text{IN}[s] = f_s(\text{OUT}[s])$$

Transfer functions

For a basic block B with statements s_1, \dots, s_n :

- $\text{IN}[B] = \text{IN}[s_1]$ and $\text{OUT}[B] = \text{OUT}[s_n]$.
- $f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$.

For forward-flow problems:

$$\begin{aligned}\text{OUT}[B] &= f_B(\text{IN}[B]) \\ \text{IN}[B] &= \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]\end{aligned}$$

For backward-flow problems:

$$\begin{aligned}\text{IN}[B] &= f_B(\text{OUT}[B]) \\ \text{OUT}[B] &= \bigcup_{S \text{ a successor of } B} \text{IN}[S]\end{aligned}$$

Reaching definitions

A statement $x = y \text{ op } z$ is a **definition** of x .

Any statement $x = y \text{ op } z$ **kills** any other definition of x .

A definition d reaches a point p if there is a path from the point immediately following d to p such that d is not *killed* along that path.

Reaching definitions are useful for:

- Knowing whether x has a constant value at point p .
- Knowing whether x is uninitialized at point p .

Transfer function for reaching definitions

Given the statement $d: \mathbf{u} = \mathbf{v} + \mathbf{w}.$

This statement *generates* a definition d of variable u and *kills* all the other definitions of variable u .

Transfer function:

$$f_d(x) = \mathit{gen}_d \cup (x - \mathit{kill}_d)$$

Composition of transfer functions:

$$\begin{aligned} f_2(f_1(x)) &= \mathit{gen}_2 \cup (\mathit{gen}_1 \cup (x - \mathit{kill}_1) - \mathit{kill}_2) \\ &= (\mathit{gen}_2 \cup (\mathit{gen}_1 - \mathit{kill}_2)) \cup (x - (\mathit{kill}_1 \cup \mathit{kill}_2)) \end{aligned}$$

Transfer function for reaching definitions

The transfer function for a basic block B is

$$f_B(x) = \textit{gen}_B \cup (x - \textit{kill}_B)$$

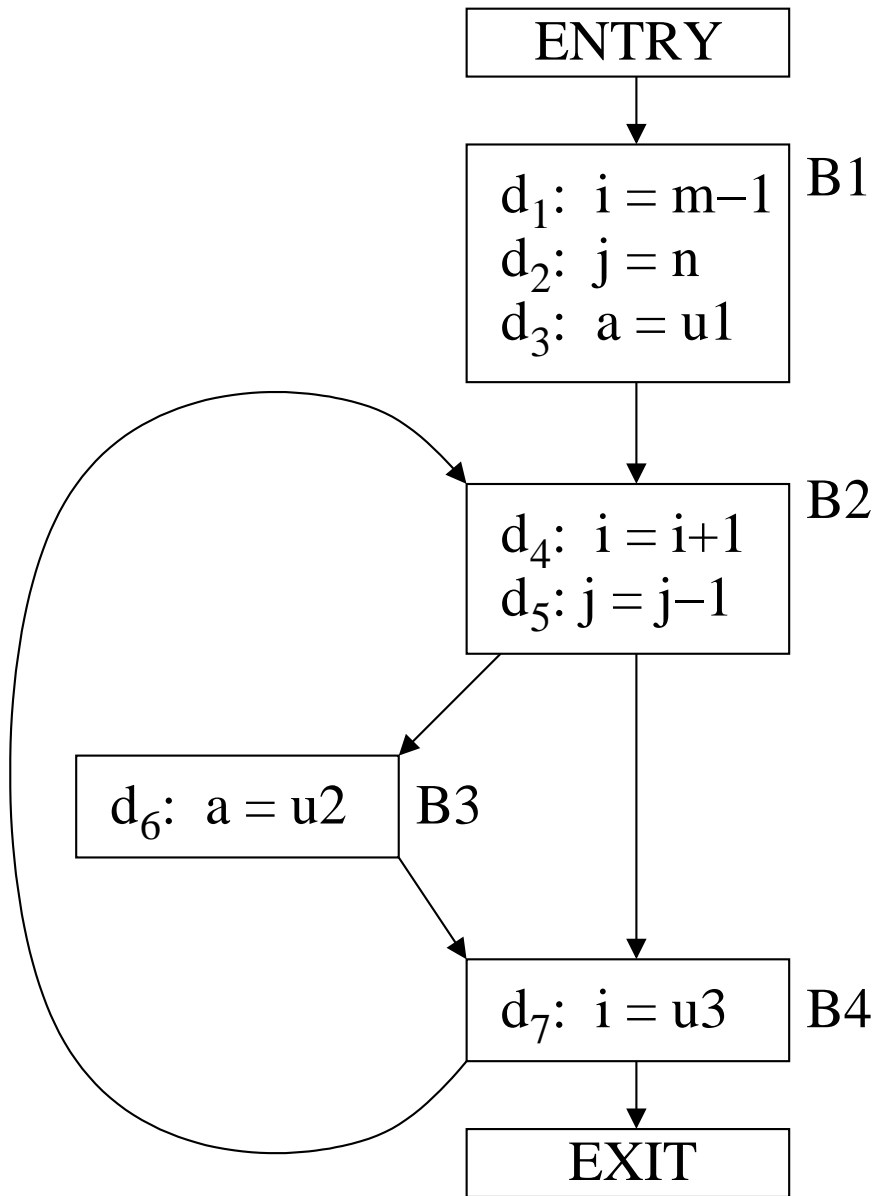
where

$$\textit{kill}_B = \textit{kill}_1 \cup \textit{kill}_2 \cup \dots \cup \textit{kill}_n$$

and

$$\begin{aligned} \textit{gen}_B &= \textit{gen}_n \cup \\ &\quad (\textit{gen}_{n-1} - \textit{kill}_n) \cup \\ &\quad (\textit{gen}_{n-2} - \textit{kill}_{n-1} - \textit{kill}_n) \cup \\ &\quad \dots \\ &\quad (\textit{gen}_1 - \textit{kill}_2 - \textit{kill}_3 - \dots - \textit{kill}_n) \end{aligned}$$

Gen and Kill



$$gen_{B1} = \{ d_1, d_2, d_3 \}$$

$$kill_{B1} = \{ d_4, d_5, d_6, d_7 \}$$

$$gen_{B2} = \{ d_4, d_5 \}$$

$$kill_{B2} = \{ d_1, d_2, d_7 \}$$

$$gen_{B3} = \{ d_6 \}$$

$$kill_{B3} = \{ d_3 \}$$

$$gen_{B4} = \{ d_7 \}$$

$$kill_{B4} = \{ d_1, d_4 \}$$

Computing reaching definitions

```
for (each basic block B) OUT[B] =  $\emptyset$ ;  
while (changes to any OUT occur)  
    for (each basic block B other than ENTRY)  
        IN[B] =  $\bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$   
        OUT[B] =  $\text{gen}_B \cup (\text{IN}[B] - \text{kill}_B)$ 
```

Block	OUT[B] ⁰	IN[B] ¹	OUT[B] ¹	IN[B] ²	OUT[B] ²
B ₁	000 0000	000 0000	111 0000	000 0000	111 0000
B ₂	000 0000	111 0000	001 1100	111 0111	001 1110
B ₃	000 0000	001 1100	000 1110	001 1110	000 1110
B ₄	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Live-Variable analysis

Problem statement:

- Given a variable x and a program point p , we want to know whether the value of x at p can be used in some path starting at p .

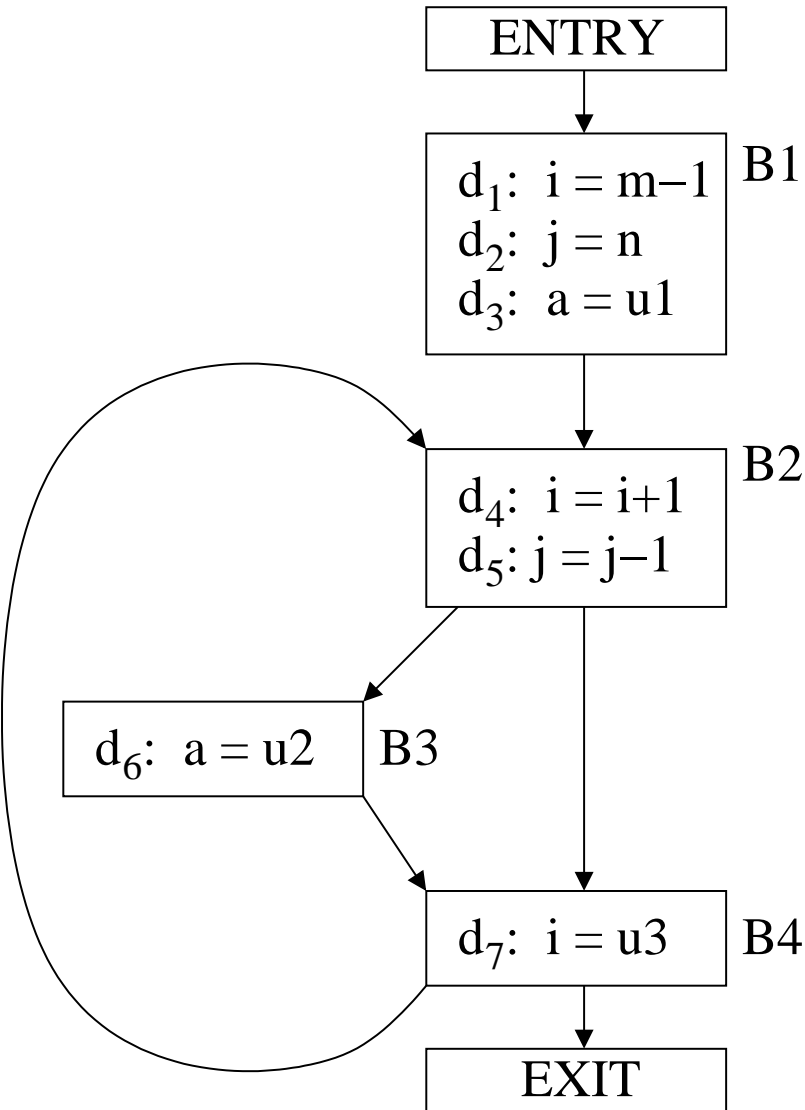
Live-variable analysis is essential for register allocation.

Live-Variable analysis

Data-flow equations:

- $\text{IN}[B]$ and $\text{OUT}[B]$ represent the set of variables live at the beginning and end of a block B , respectively.
- The transfer functions of a block can be calculated by composing the transfer functions of individual statements.
- def_B is the set of variables *defined* in B prior to any use of that variable in B .
- use_B is the set of variables whose values may be used in B prior to any definition of the variable.

Live-Variable analysis



Block	<i>use</i>	<i>def</i>
$B1$	$\{m, n, u1\}$	$\{i, j, a\}$
$B2$	$\{i, j\}$	$\{\}$
$B3$	$\{u2\}$	$\{a\}$
$B4$	$\{u3\}$	$\{i\}$

Live-Variable analysis

Data-flow equations:

$$\text{IN}[\text{EXIT}] = \emptyset$$

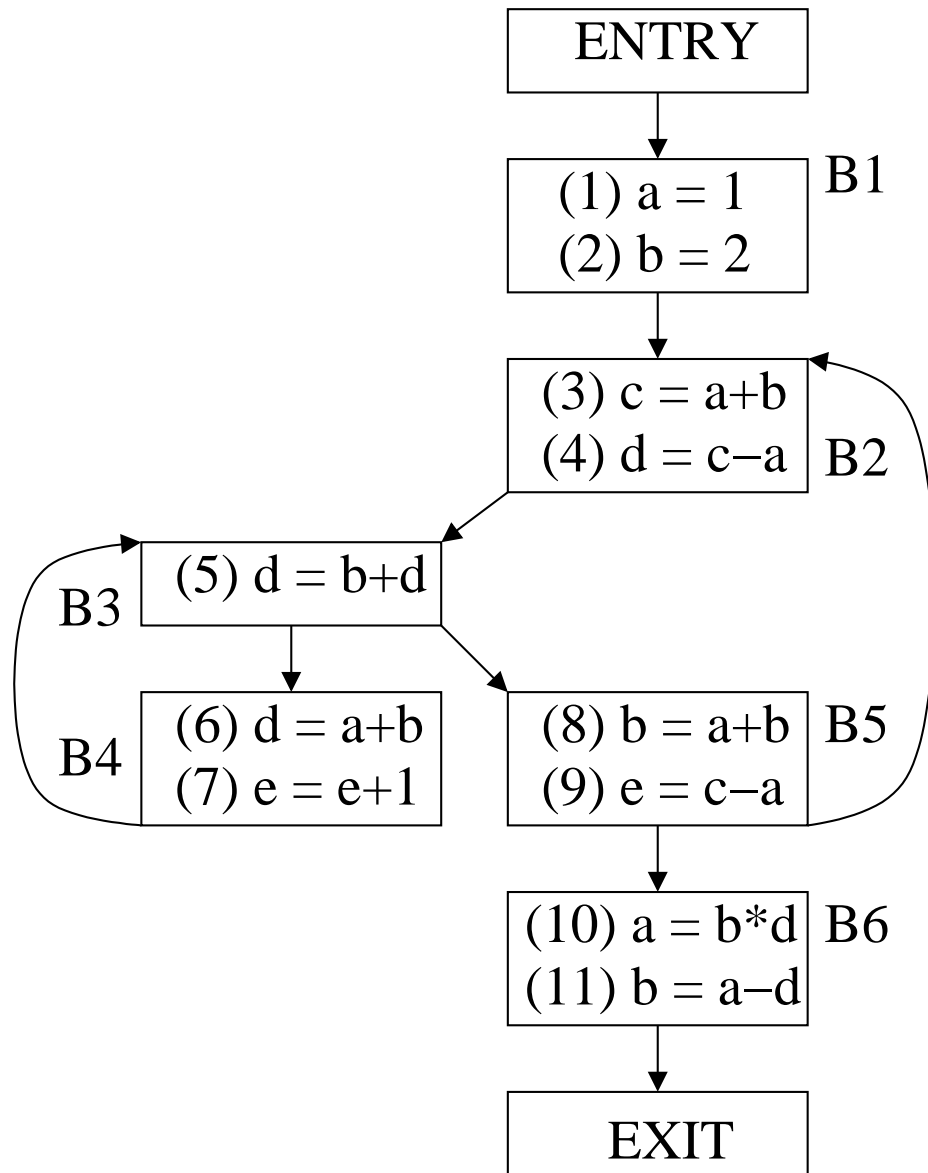
For all other basic blocks B :

$$\begin{aligned}\text{IN}[B] &= \text{use}_B \cup (\text{OUT}[B] - \text{def}_B) \\ \text{OUT}[B] &= \bigcup_{S \text{ a successor of } B} \text{IN}[S]\end{aligned}$$

Algorithm:

- Initialize $\text{IN}[B] = \emptyset$ for all B .
- Iterate the data-flow equations until no changes occur.

Live-Variable analysis

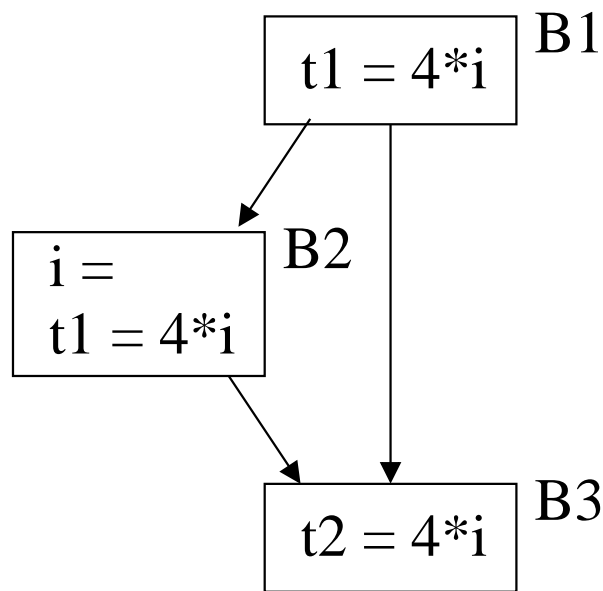
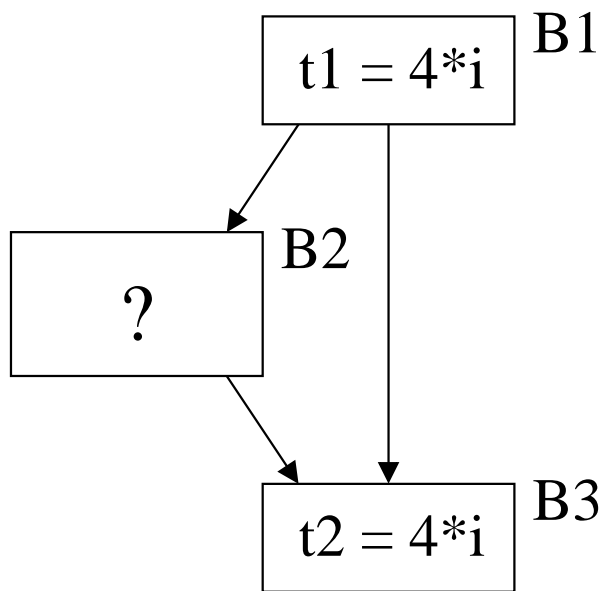


Exercise:

Find the variables alive at each point of the program.

Available expressions

An expression $x + y$ is *available* at a point p if every path from the entry node to p evaluates $x + y$, and after the last such evaluation prior to reaching p , there are no subsequent assignments to x or y .



Is $4 * i$ available at the entry of B3 ?

Available expressions

Data-flow schema:

- A block *kills* expression $x + y$ if it assigns x or y and does not subsequently recompute $x + y$.
- A block *generates* expression $x + y$ if it evaluates $x + y$ and does not subsequently define x or y .

Generated expressions in a block (from beginning to end):

- Let S the set of expressions available at point p and let q be the point after p , with statement $\mathbf{x=y+z}$.
- Add to S the expression $y + z$.
- Delete from S any expression involving variable x .

Available expressions

Statement	Available expressions
	\emptyset
$a = b + c$	$\{b+c\}$
$d = a + b$	$\{b+c, a+b\}$
$a = e - b$	$\{b+c, e-b\}$
$b = c * d$	$\{c*d\}$

Available expressions

Data-flow equations:

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

For all other basic blocks B :

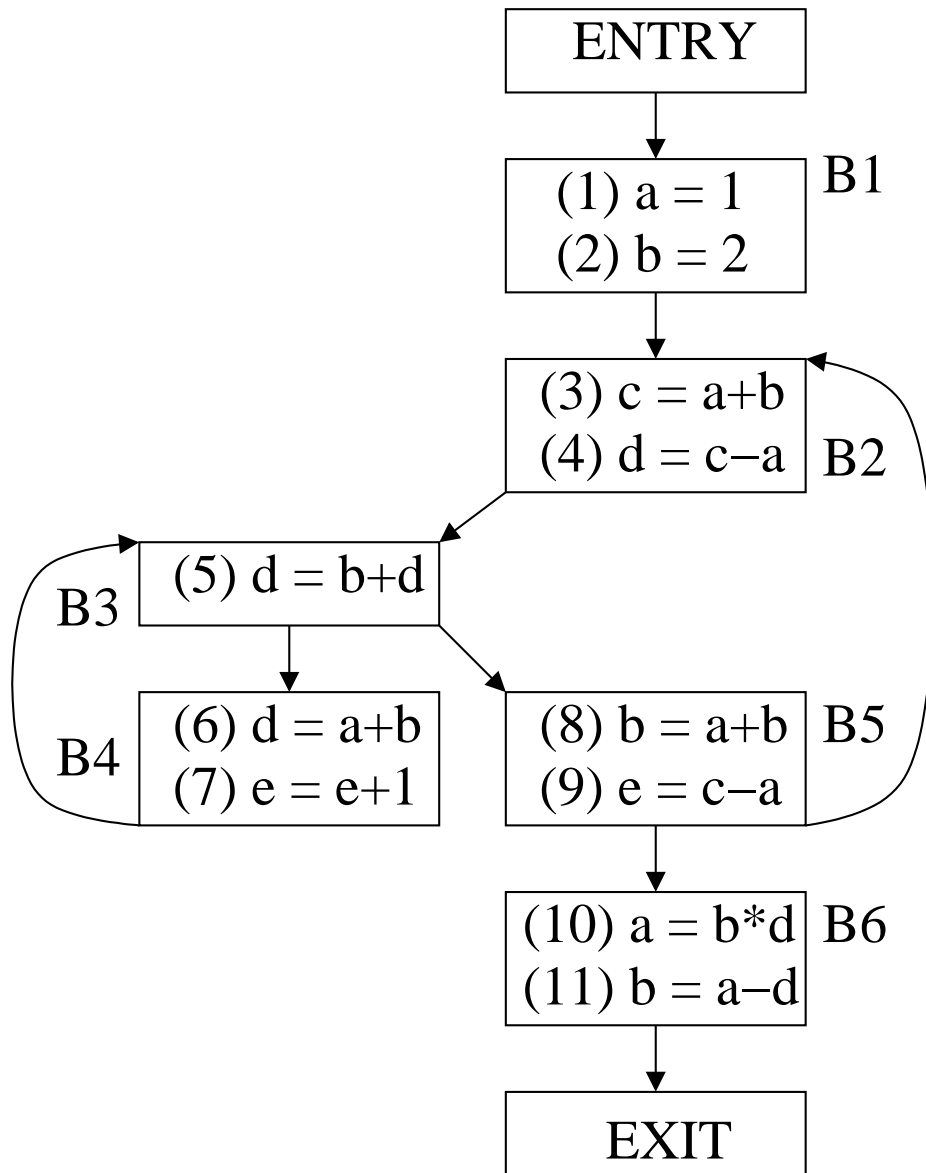
$$\text{OUT}[B] = e_gen_B \cup (\text{IN}[B] - e_kill_B)$$

$$\text{IN}[B] = \bigcap_{P \text{ a predecessor of } B} \text{OUT}[P]$$

Algorithm:

- Initialize $\text{OUT}[\text{ENTRY}] = \emptyset$ and $\text{OUT}[B] = \mathbb{U}$ for all other blocks (\mathbb{U} = "all expressions").
- Iterate the data-flow equations until no changes occur.

Available expressions



Exercise:

Compute the available expressions at each point of the program.

Use-def chains

In a similar way as live variable are calculated, we can also calculate *use-def chains*.

Given an instruction that *defines* variable x (e.g. $x = \dots$), it is possible to calculate all the instructions that can *use* this definition.

Method: Find forward paths from the instruction that are not *killed* by other definitions of x .

Elimination of common subexpressions

Let S be the instruction $\mathbf{x=y+z}$ such that $y + z$ is available at the beginning of the block and y and z are not defined before S .

1. Find (backward) all the evaluations of $y + z$. Stop the search each time $y + z$ is found.
2. Create a new variable \mathbf{u} .
3. Substitute each found instruction $\mathbf{w=y+z}$ by

$$\mathbf{u = y+z}$$

$$\mathbf{w = u}$$

4. Substitute S by $\mathbf{x=u}$.

Elimination of common subexpressions introduces *copies*.

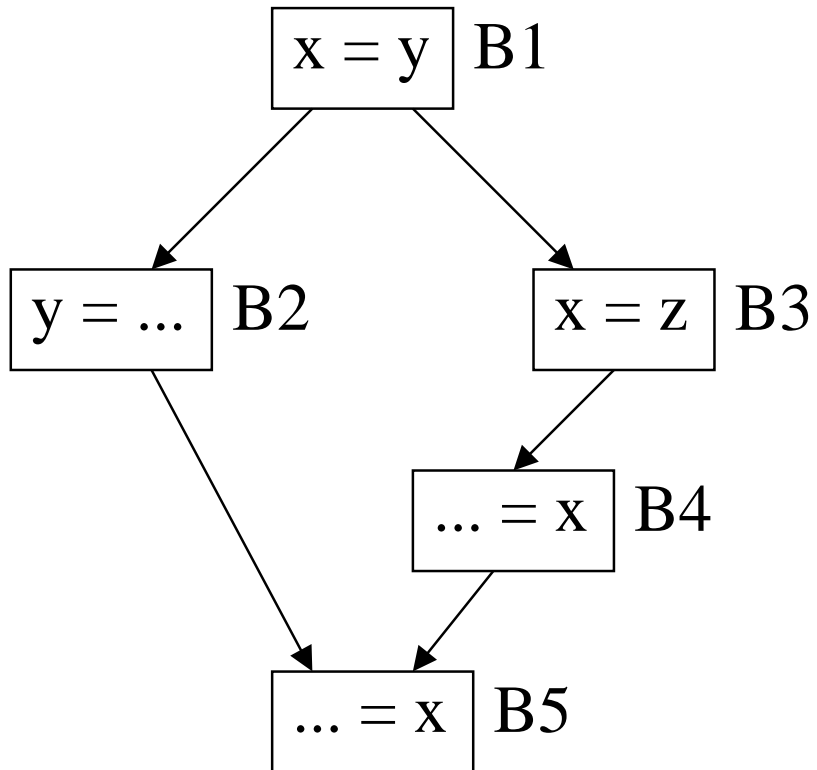
Copy propagation

Goal: Eliminate copy instructions (e.g. $\mathbf{x=y}$).

Given an instruction $\mathbf{s: x=y}$ (definition):

- Calculate all *uses* of the definition.
- Every use of \mathbf{x} can be substituted by \mathbf{y} at the instruction \mathbf{u} if:
 1. \mathbf{s} is the only definition of \mathbf{x} that reaches \mathbf{u} , and
 2. For all paths from \mathbf{s} to \mathbf{u} there is no assignment to \mathbf{y} .

Copy propagation



Only the use in B4 can be substituted by z .

Dead-code elimination

Programmers do not introduce dead code intentionally, but it may appear as a result of some transformations.

Example:

```
debug = FALSE;  
...  
if (debug) { ... }
```

Dead code often appears after copy propagation:

<code>x = t3</code>		<code>x = t3</code>
<code>a[t2] = t5</code>		<code>a[t2] = t5</code>
<code>a[t4] = x</code>	\Rightarrow	<code>a[t4] = t3</code>
<code>goto B2</code>		<code>goto B2</code>

A variable is *live* at a point if its value can be used subsequently. The statement `x = t3` can be eliminated if `x` is *dead* at that point.

Optimization on loops

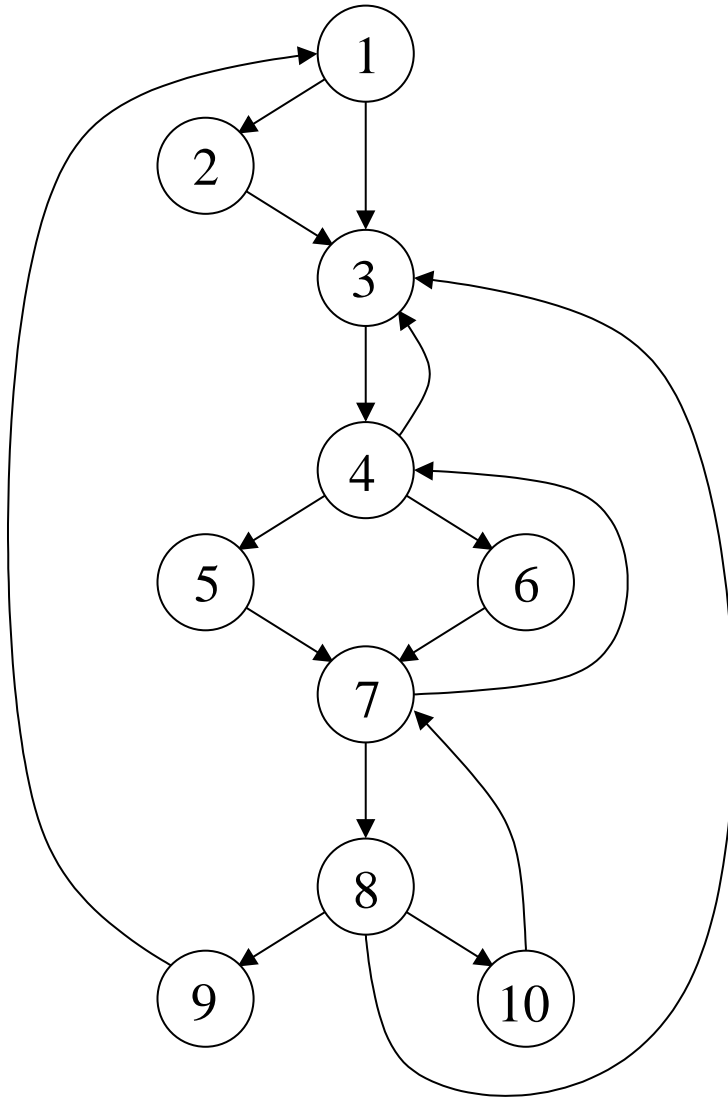
Optimization on loops

Programs spend most of their time executing instructions in loops. Loop optimization is important.

Outline:

- What is a loop?
- Invariant computations.
- Induction Variables and Reduction in Strength.

Dominators



A node d of a flow graph *dominates* node n ($d \text{ dom } n$) if every path from the entry node to n goes through d .

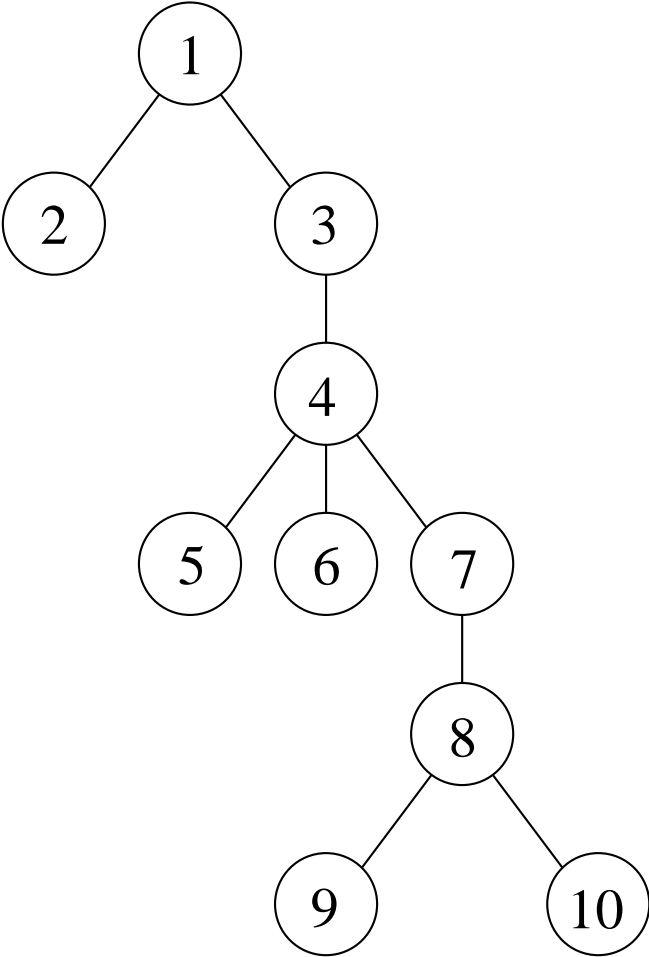
1 dominates all nodes.

2 dominates only itself.

3 dominates all but 1 and 2.

7 dominates 7, 8, 9 and 10.

Dominator tree



Algorithm to compute dominators

Data-flow equations:

- $OUT[n] = IN[n] \cup \{n\}$
- $IN[n] = \bigcap_{m \text{ a predecessor of } n} OUT[m]$

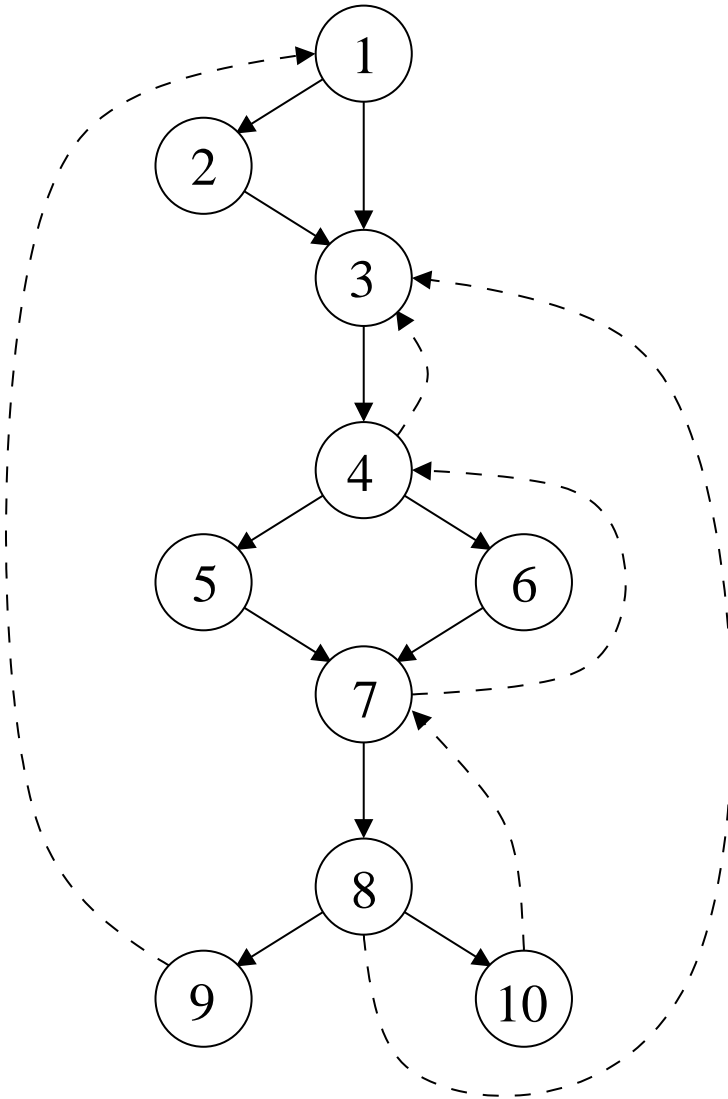
Algorithm:

- Initialize $OUT[ENTRY] = \{ENTRY\}$ and $OUT[B] = \{\text{all nodes}\}$ for all other blocks
- Iterate the data-flow equations forward until no changes occur.

Result:

- $OUT[n]$ is the set of nodes that dominate n .

Back edges



Back edge:

- An edge $a \rightarrow b$ such that $b \text{ dom } a$.

Natural loops

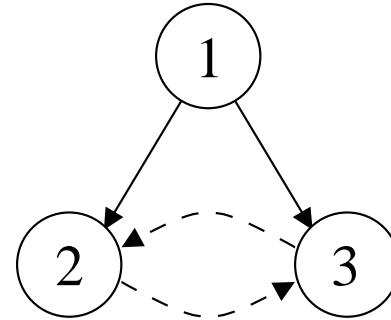
A *natural loop* is a set of nodes with two properties:

- It has a single-entry node, called the *header*. The header dominates all nodes in the loop.
- There is a back edge that enters the header.

Given a back edge $n \rightarrow d$, the *natural loop of the edge* is the set of nodes that can reach n without going through d . Node d is the header of the loop.

Reducible graphs

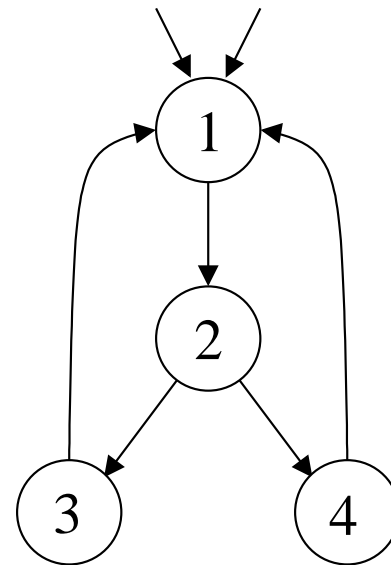
A graph is *reducible* if it is acyclic after removing its back edges.



Nonreducible flow graph

Property of reducible graphs:

- All back edges are associated to a natural loop.
- If two loops have different headers, they are either disjoint or one is nested within the other.



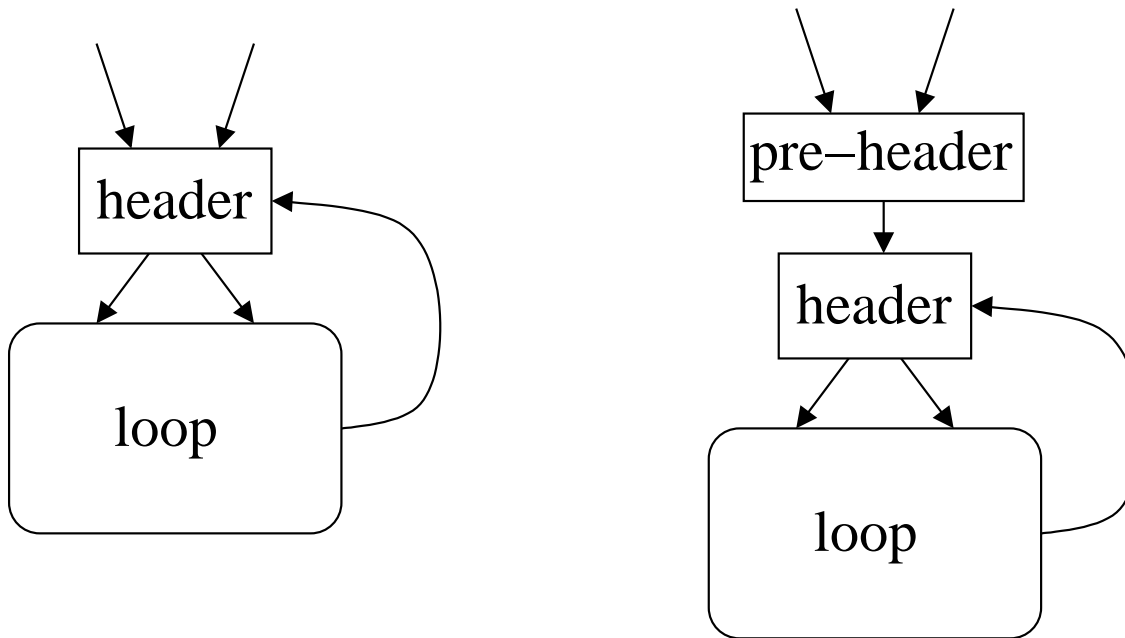
Two loops with the same header

Innermost loops: loops that contain no other loops.

Pre-header

Strategy:

- Create a pre-header before the header of the loop (see figure)
- Move computations from the loop to the header



Invariant computations

- Let us assume that the statement $x = y + z$ is inside a loop and all the definitions of y and z are outside the loop. Then $y + z$ is invariant inside the loop.
- Assume that $v = x + w$ is another statement in the loop and all the definitions of w are outside the loop. Then, $x + w$ is also invariant.

Algorithm to calculate invariant computations:

1. Mark as invariant all those statements with constant operands or with all definitions outside the loop.
2. Repeat 3 until no changes
3. Mark as invariant those statements with constant operands, with all definitions outside the loop or with only one definition that reaches the statement and this definition being invariant.

Moving invariants to the pre-header

Assume that the statement $\mathbf{x}=\mathbf{y}+\mathbf{z}$ is invariant. The statement can be moved to the pre-header if:

- the block that contains the statement dominates all the exits of the loop and
- there is no other definition of \mathbf{x} inside the loop and
- no other use of \mathbf{x} in the loop has another definition different from the statement.

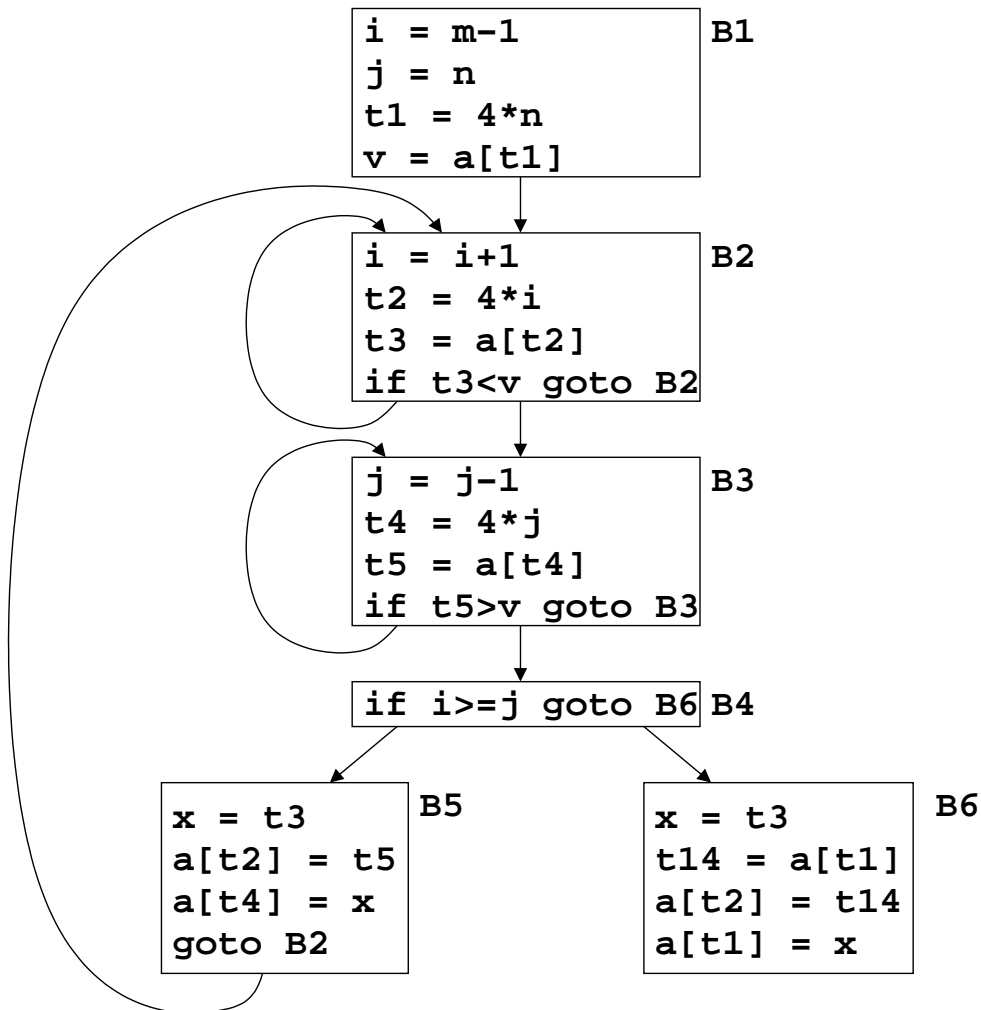
If some of the above conditions does not hold, it is still possible:

- Move the computation $\mathbf{t}=\mathbf{y}+\mathbf{z}$ to the pre-header
- Keep the assignment $\mathbf{x}=\mathbf{t}$ in the loop

Induction variables

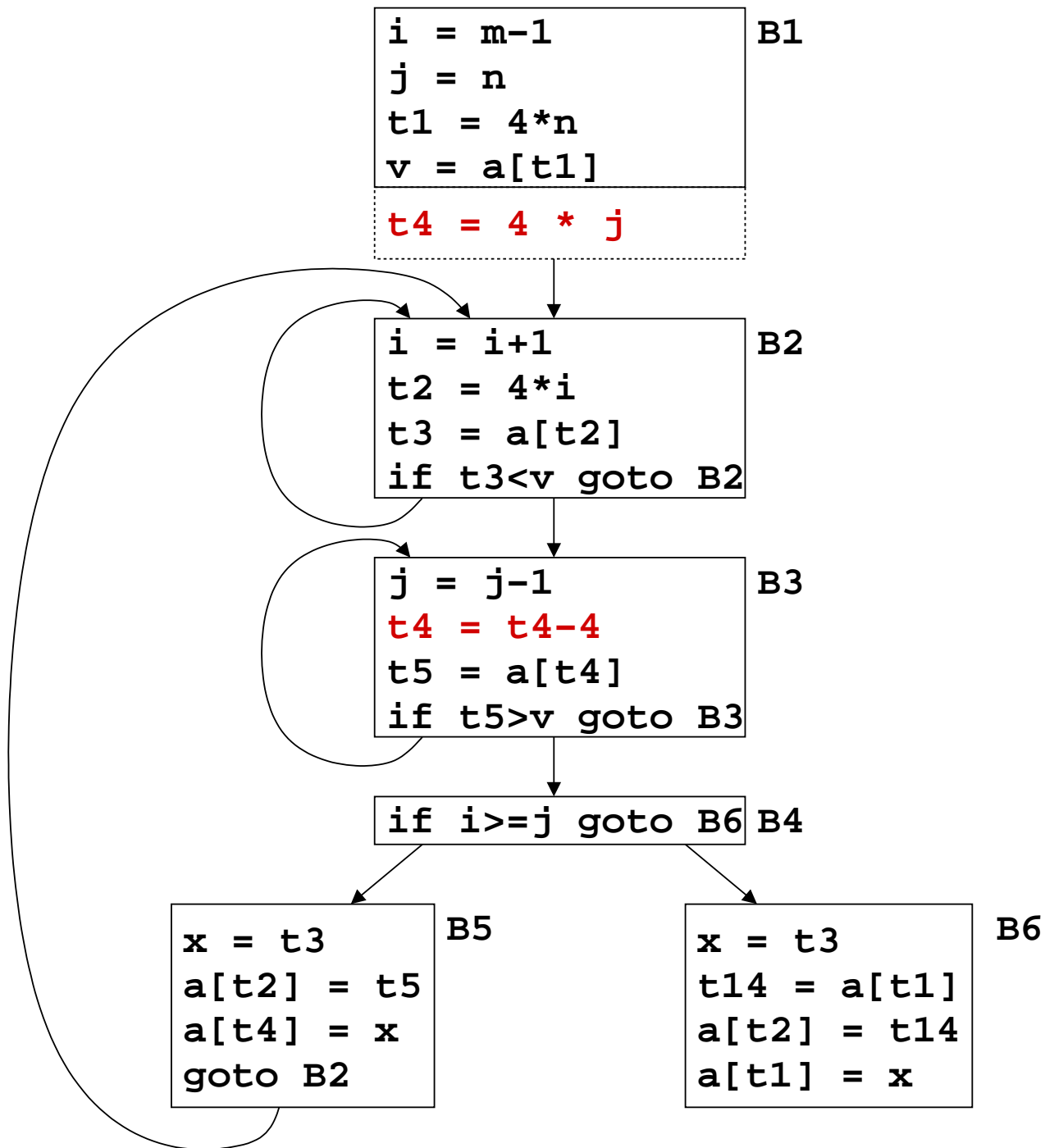
A variable x is said to be an *induction variable* if there is a positive or negative constant c such that each time x is assigned, its value increases by c .

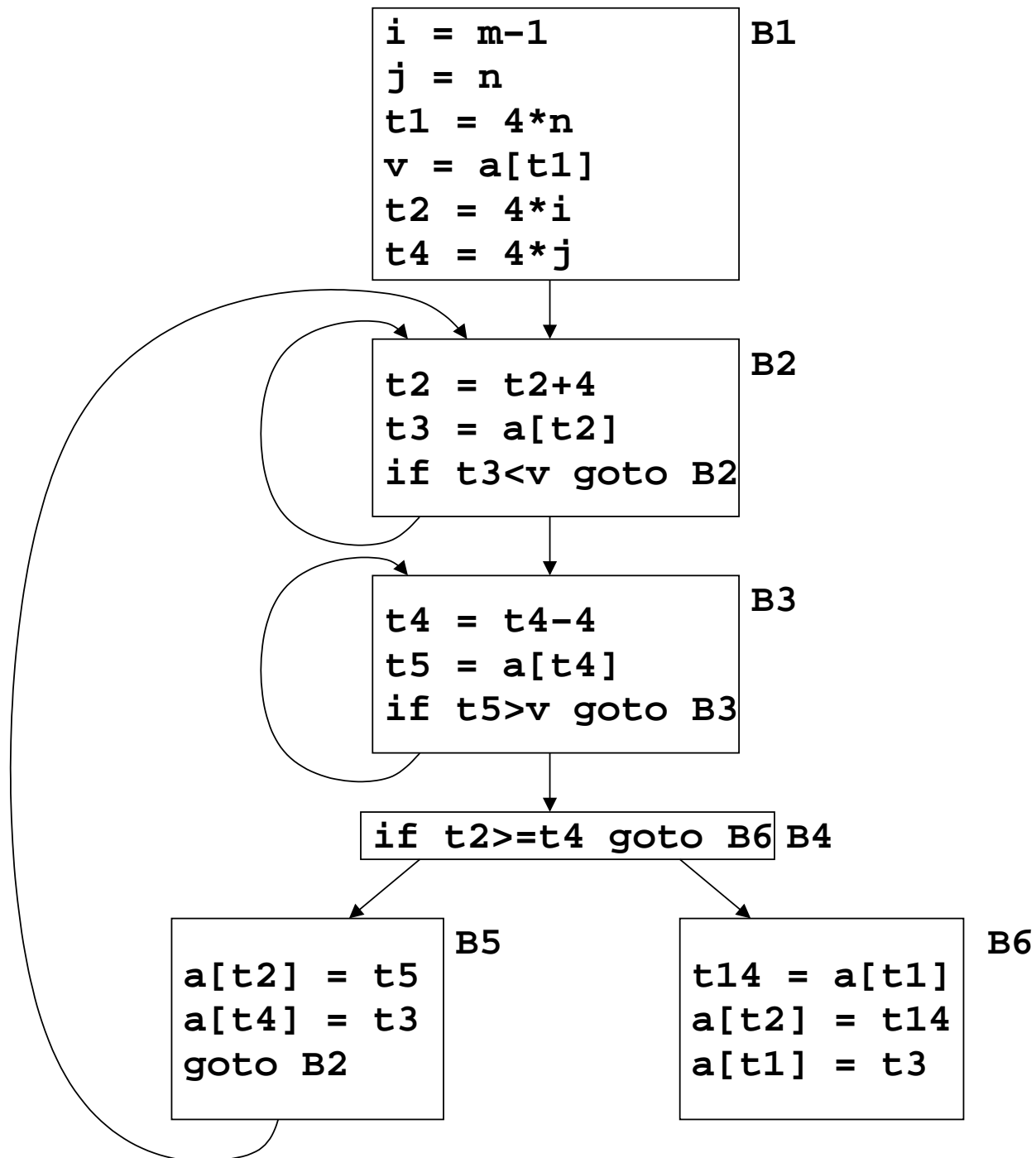
i and j are induction variables, but $t2$ and $t4$ are also induction variables.



Induction variables

- Find variables with only one assignment inside the loop of the form $i = i \pm c_1$.
- Find variables with only one assignment inside the loop of the form $k = c_2 * i + d$, such that $i = i \pm c_1$ is the only definition reaching this assignment.
- Move $k = c_2 * i + d$ outside the loop (pre-header).
- Substitute by $k = k \pm (c_1 * c_2)$ inside the loop.





Exercise 1

Consider the following basic block:

$g = x + y$

$h = u - v$

$i = x + y$

$x = u - v$

$u = g + h$

$v = i + x$

$w = u + v$

- Build the DAG representation
- Generate code under the assumptions that the set of live variables is $\{g, h, u, w\}$ and $\{u, h, x\}$.

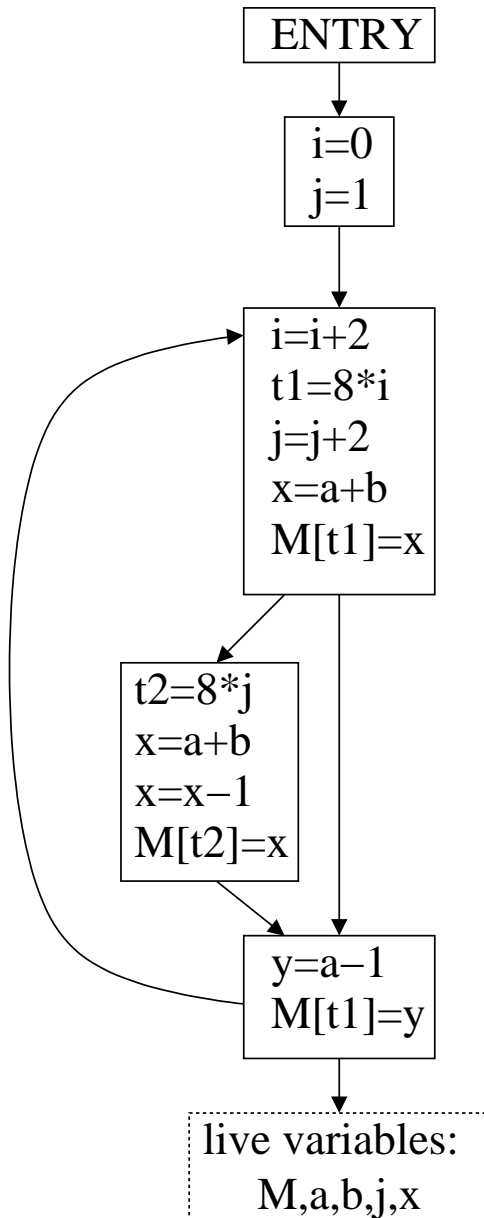
Exercise 2

Consider the following code:

- Calculate the basic blocks and draw the associated control-flow graph.
- Calculate the tree of dominators.
- Identify the back edges and loops of the control-flow graph.
- Perform loop optimizations (assume that all variables are alive at the exit of the code).

```
1:    m = 0
2:    v = 0
3:    t = x+4
4:    if v >= n goto 18
5:    r = v
6:    s = 0
7:    k = 0
8:    k = k+1
9:    z = k*4
10:   q = M[z]
11:   s = s+q
12:   if s <= m goto 14
13:   m = s
14:   r = r+1
15:   if r < n goto 8
16:   v = v+1
17:   goto 3
18:   return m
```

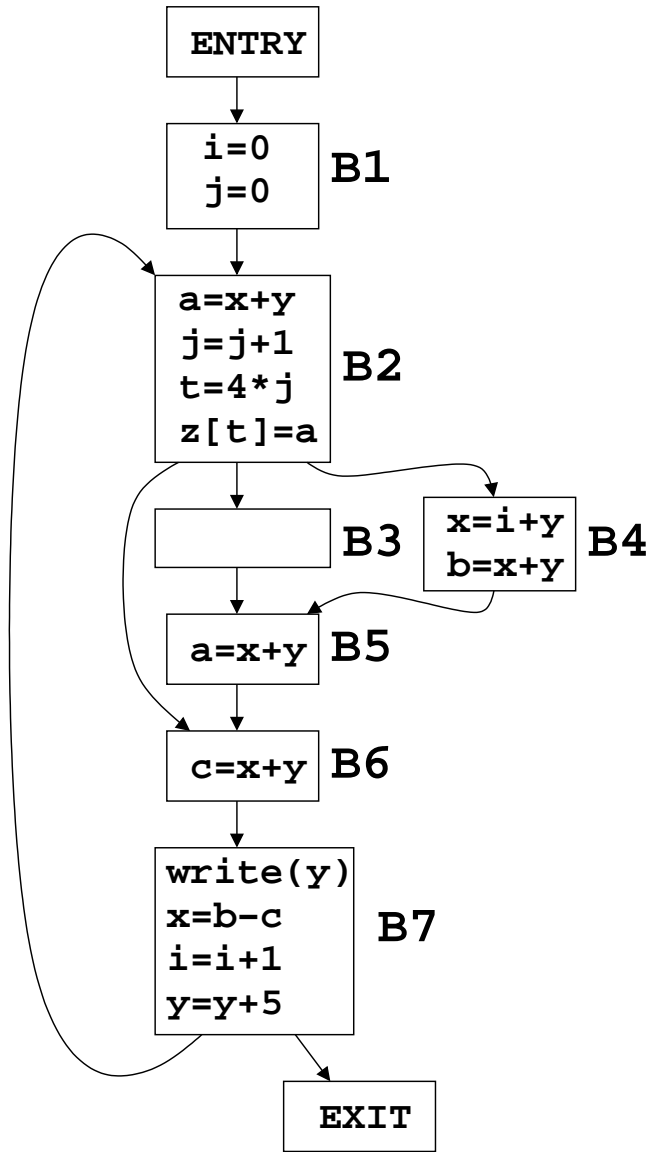
Exercise 3



Consider the code in the figure.

- Apply all possible optimizations considering the live variables specified at the exit of the code.
- Show the final code after the optimizations and explain the transformations produced by each optimization.

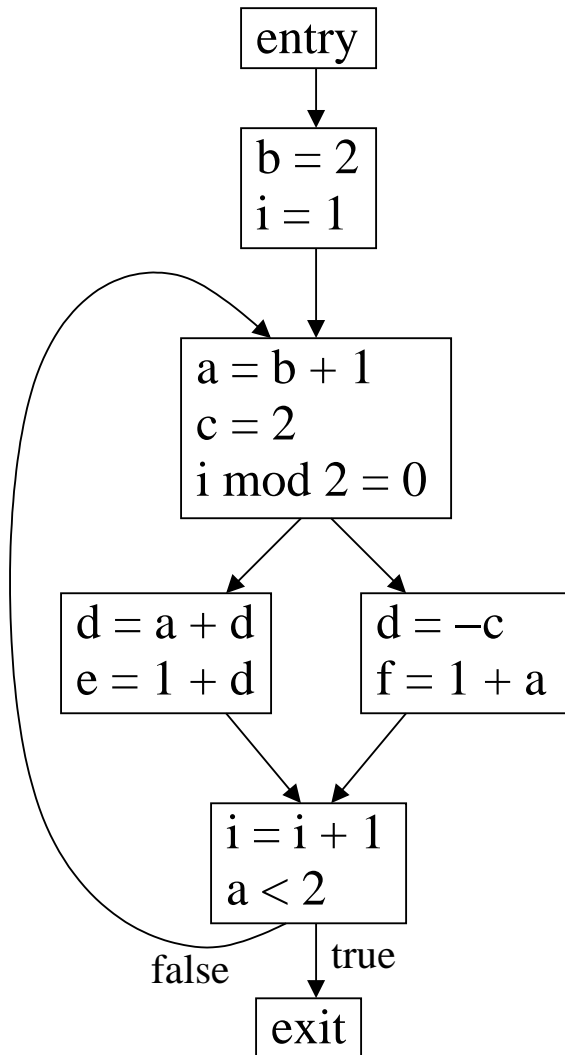
Exercise 4



Consider the code in the figure.

- Which variables are alive at the end of block B7?
- Which blocks dominate B7?
- Apply all possible optimizations you know. Assume array `z` is the only variable alive at the exit of the code.
- Assume that `x` and `y` are parameters of the function initialized before the function call. Is there any variable that could be potentially used without having been initialized before?

Exercise 5



Consider the code in the figure.

- Apply all possible optimizations.
- Discuss how much information can be extracted at compile time from the execution of the code.