Code Optimization

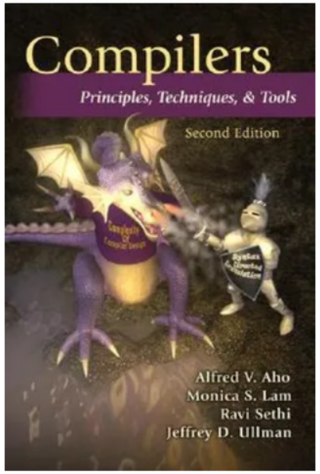
José Miguel Rivero   Jordi Cortadella

Dept. CS, UPC

2025

Credits

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

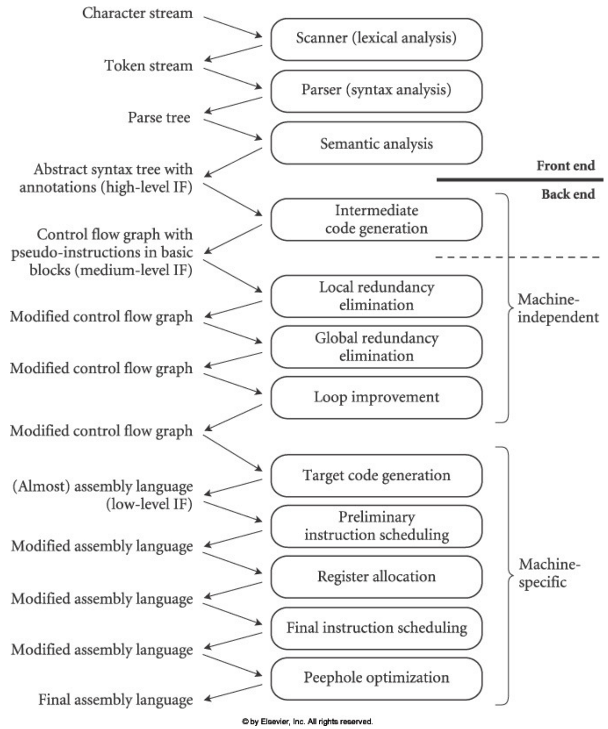


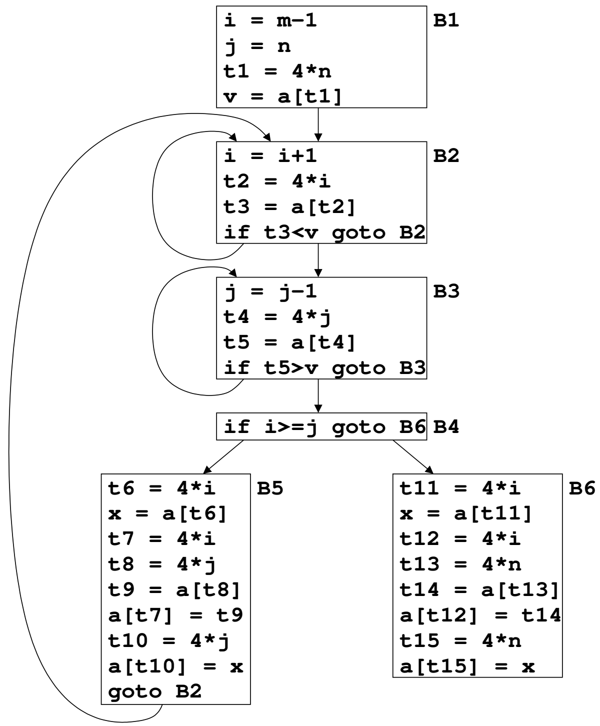
Table of contents

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>=j goto (23)  (28) a[t12] = t14
(14) t6 = 4*i           (29) t15 = 4*n
(15) x = a[t6]          (30) a[t15] = x
```

Local redundancy 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
```

B5

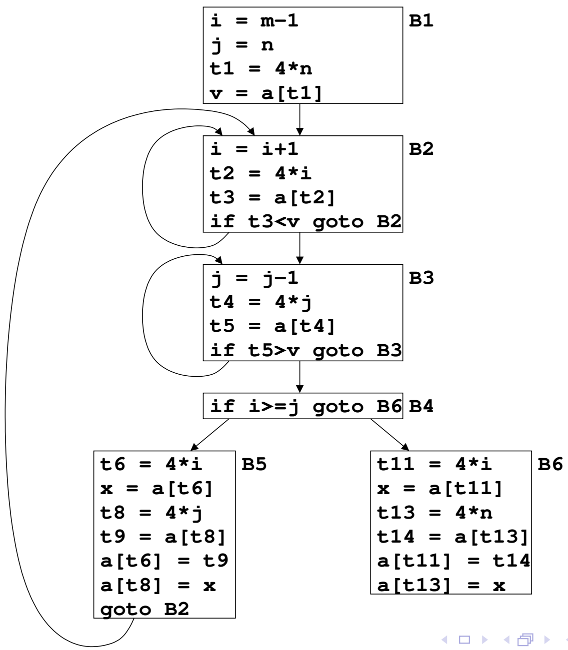
Local redundancy elimination

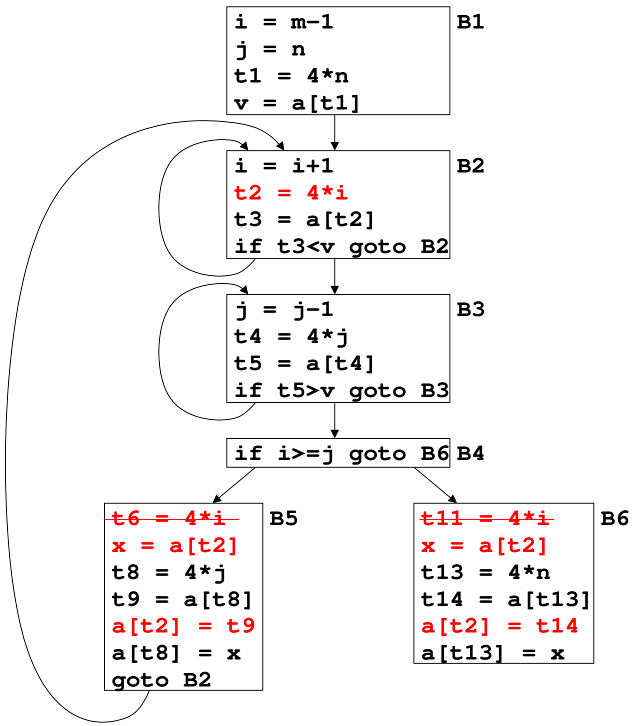
```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

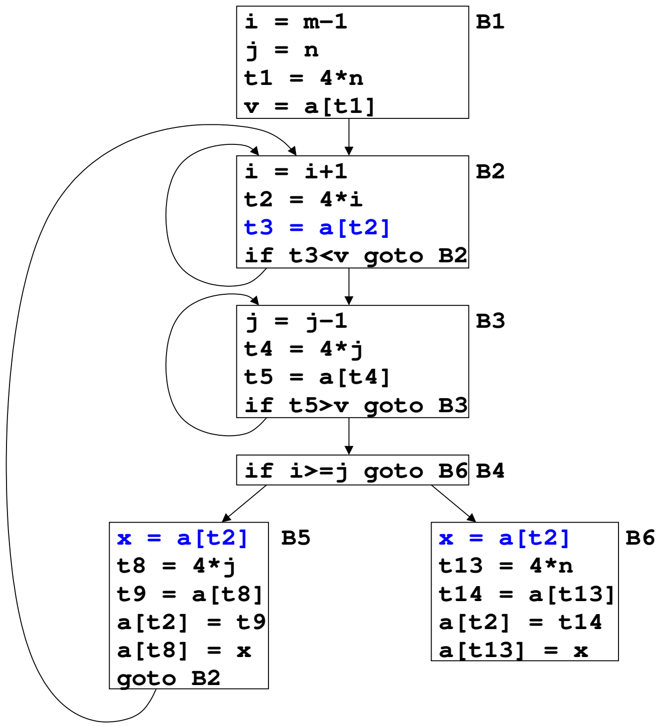
$\Rightarrow$

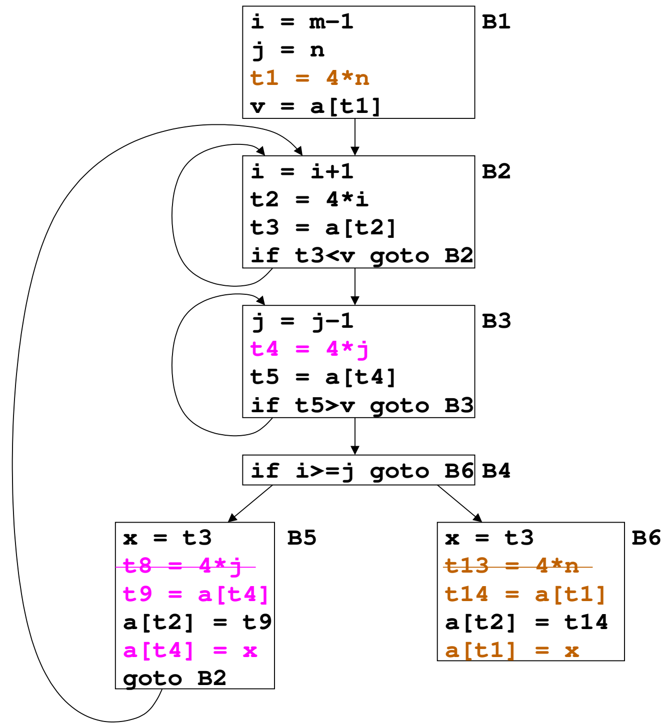
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

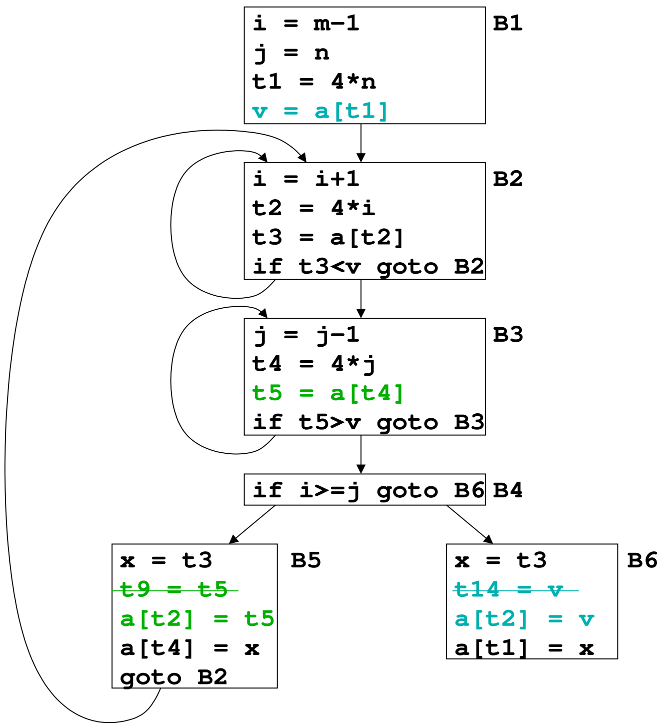

Global redundancy elimination

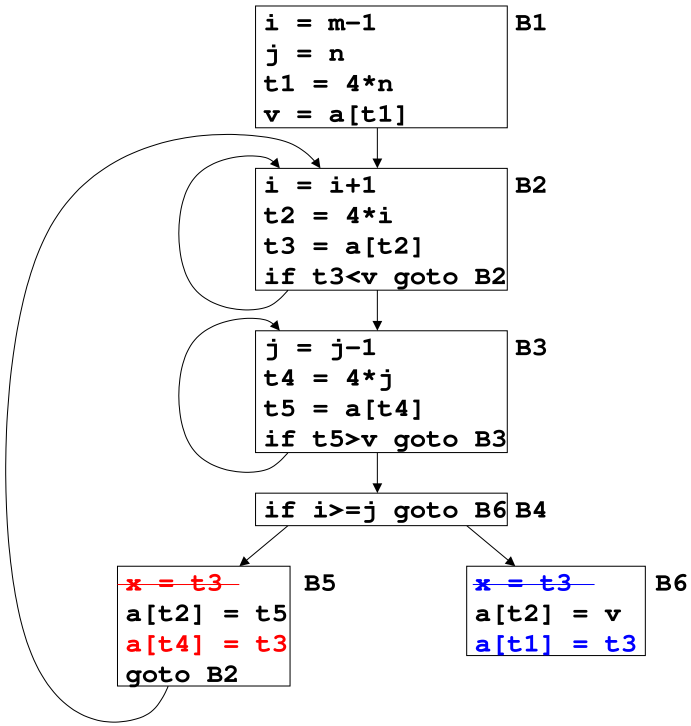




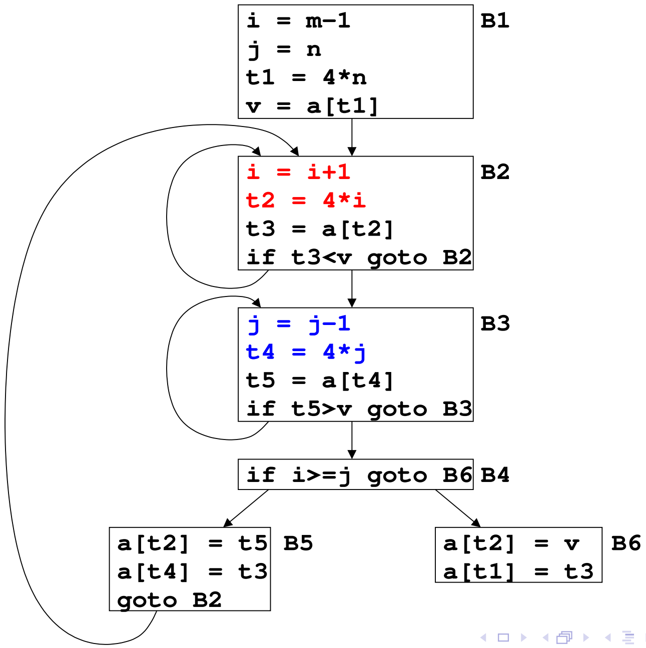

i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
x = a[t2]
t8 = 4*j
t9 = a[t8]
a[t2] = t9
a[t8] = x
goto B2
B5
x = a[t2]
t13 = 4*n
t14 = a[t13]
a[t2] = t14
a[t13] = x
B6

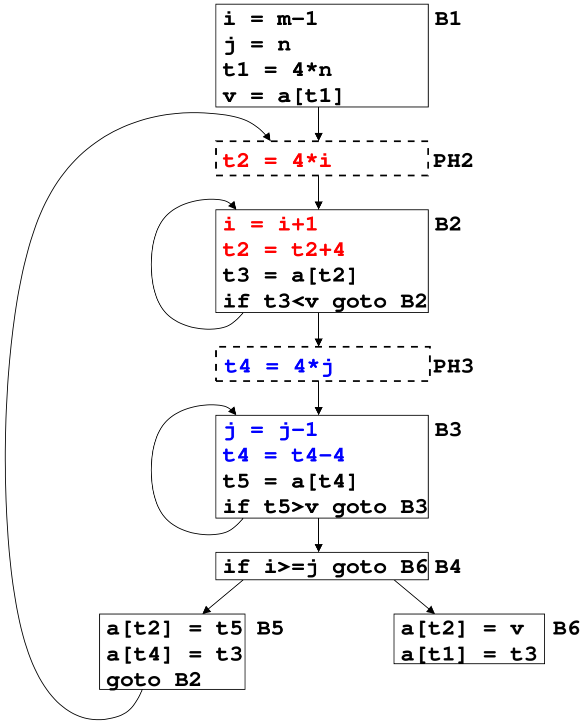

i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
x = t3
~~t8 = 4*j~~
t9 = a[t4]
a[t2] = t9
a[t4] = x
goto B2
B5
x = t3
~~t13 = 4*n~~
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6

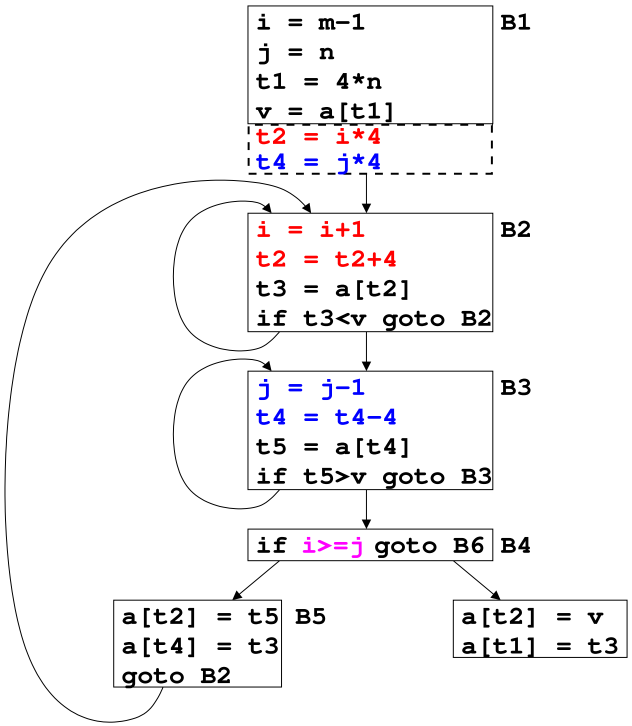

i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
x = t3
~~t9 = t5~~
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
~~t14 = v~~
a[t2] = v
a[t1] = x
B6


i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
~~x = t3~~
a[t2] = t5
a[t4] = t3
goto B2
B5
~~x = t3~~
a[t2] = v
a[t1] = t3
B6

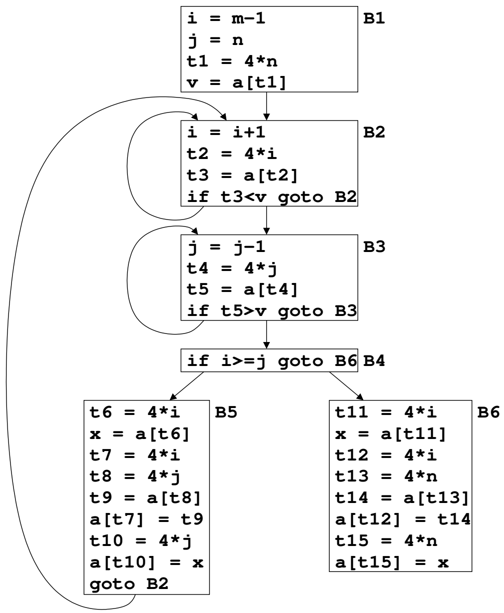
Loop improvement

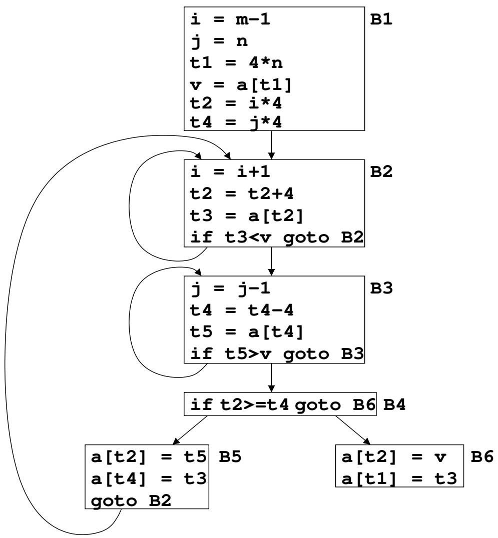






Impact of optimization





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
    $\text{a}[i, j] = 0.0;$
for i from 1 to 10 do
  $\text{a}[i, i] = 1.0;$

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

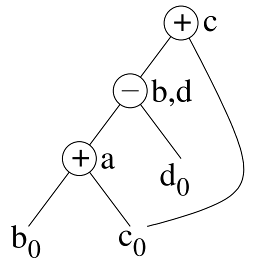
```
(1)  i = 1
--------------------------
(2)  j = 1
--------------------------
(3)  t1 = 10 * i
(4)  t2 = t1 + j
(5)  t3 = 8 * t2
(6)  t4 = t3 - 88
(7)  a[t4] = 0.0
(8)  j = j + 1
(9)  if j <= 10 goto (3)
--------------------------
(10) i = i + 1
(11) if i <= 10 goto (2)
--------------------------
(12)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



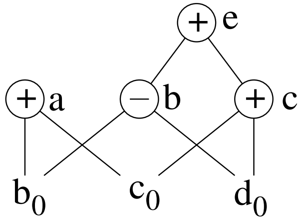

```
a = b + c
d = a - d
c = d + c
```

(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.

```
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

$x+0=0+x=x$
$x\times 1=1\times x=x$
$x-0=x$
$x/1=x$

Expensive		Cheaper
x^2	$=$	$x\times x$
$2\times x$	$=$	$x+x$
$x/2$	$=$	$x\times 0.5$

Constant folding: `2 * 3.14` replaced by `6.28`

Commutativity: $x*y=y*x$

Associativity: $(x+y)+z=x+(y+z)$

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

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

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

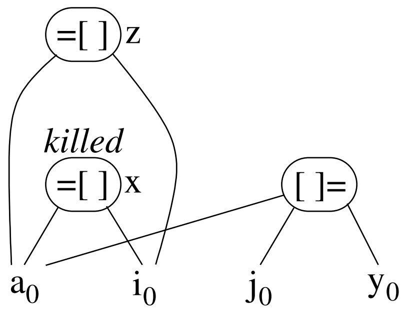
Array references

```
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
    ...
L1: goto L2

      ⇓

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

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

      ⇓

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

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

      ⇓

    if a < b goto L2
    goto L3
    ...
    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

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.

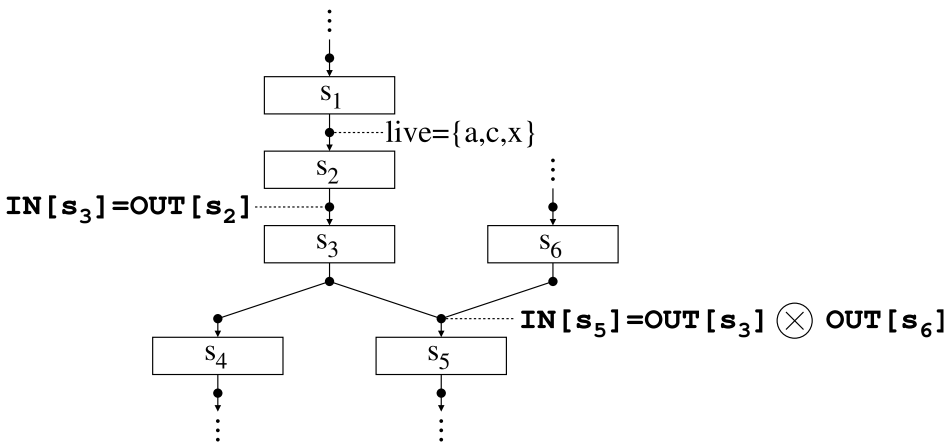
The domain of data-flow values will depend on the type of analysis to be performed (e.g. set of live variables).

For every statement s, the data-flow values before and after the statement are denoted by $\textsc{In}[s]$ and $\textsc{Out}[s]$, respectively.

Data-Flow Problem

Find a solution for all $\textsc{In}[s]$ and $\textsc{Out}[s]$ such that the constraints determined by the semantics of the statements and the flow of control are met.

Data-Flow Analysis Schema



Transfer functions

The relationship between $\textsc{In}[s]$ and $\textsc{Out}[s]$ is known as the transfer function.

Transfer functions:

- For forward propagation:

$$\textsc{Out}[s] = f_s(\textsc{In}[s])$$

- For backward propagation:

$$\textsc{In}[s] = f_s(\textsc{Out}[s])$$

Transfer functions

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

- $\textsc{In}[B] = \textsc{In}[s_1]$ and $\textsc{Out}[B] = \textsc{Out}[s_n]$.
- $f_B = f_{s_n} \circ \cdots \circ f_{s_2} \circ f_{s_1}$.

For forward-flow problems:

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

For backward-flow problems:

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

Data-Flow Problems

- Reaching definitions
- Live-Variable analysis
- Available expressions
- Use-def chains

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.

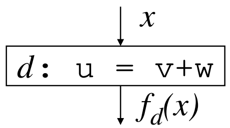
Transfer function for reaching definitions

Given the statement d: `u = v+w`.

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

Transfer function:

$$f_d(x) = gen_d \cup (x - kill_d)$$



Composition of transfer functions:

$$\begin{aligned} f_2(f_1(x)) &= gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2) \\ &= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2)) \end{aligned}$$

Transfer function for reaching definitions

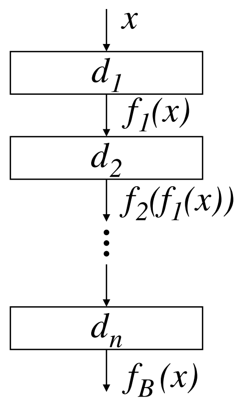
The transfer function for a basic block B is

$$f_B(x) = gen_B \cup (x - kill_B)$$

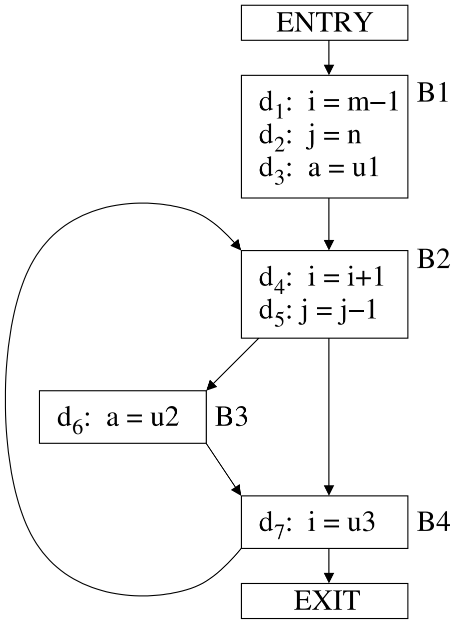
where:

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

$$\begin{aligned} gen_B \quad = \quad & gen_n \cup \\ & (gen_{n-1} - kill_n) \cup \\ & (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ & \ldots \\ & (gen_1 - kill_2 - kill_3 - \cdots - kill_n) \end{aligned}$$



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

A *least fixed point* is reached. $d_1 d_2 d_3 \;\; d_4 d_5 d_6 d_7$

Block	gen_B	$kill_B$	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$
B_1	111 0000	000 1111			
B_2	000 1100	110 0001			
B_3	000 0010	001 0000			
B_4	000 0001	100 1000			
EXIT	000 0000	000 0000			

Computing reaching definitions

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

A *least fixed point* is reached. $d_1 d_2 d_3 \;\; d_4 d_5 d_6 d_7$

Block	gen_B	$kill_B$	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$
B_1	111 0000	000 1111	000 0000	000 0000	111 0000
B_2	000 1100	110 0001	000 0000	000 0000	000 1100
B_3	000 0010	001 0000	000 0000	000 0000	000 0010
B_4	000 0001	100 1000	000 0000	000 0000	000 0001
EXIT	000 0000	000 0000	000 0000	000 0000	000 0000

Computing reaching definitions

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

A *least fixed point* is reached. $d_1 d_2 d_3 \;\; d_4 d_5 d_6 d_7$

Block	gen_B	$kill_B$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	111 0000	000 1111	111 0000	000 0000	111 0000
B_2	000 1100	110 0001	000 1100	111 0001	001 1100
B_3	000 0010	001 0000	000 0010	000 1100	000 1110
B_4	000 0001	100 1000	000 0001	000 1110	000 0111
EXIT	000 0000	000 0000	000 0000	000 0001	000 0001

Computing reaching definitions

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

A *least fixed point* is reached. $d_1 d_2 d_3 \;\; d_4 d_5 d_6 d_7$

Block	gen_B	$kill_B$	$\text{OUT}[B]^2$	$\text{IN}[B]^3$	$\text{OUT}[B]^3$
B_1	111 0000	000 1111	111 0000	000 0000	111 0000
B_2	000 1100	110 0001	001 1100	111 0111	001 1110
B_3	000 0010	001 0000	000 1110	001 1100	000 1110
B_4	000 0001	100 1000	000 0111	001 1110	000 0111
EXIT	000 0000	000 0000	000 0001	000 0111	000 0111

Computing reaching definitions

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

A *least fixed point* is reached. $d_1 d_2 d_3 \;\; d_4 d_5 d_6 d_7$

Block	gen_B	$kill_B$	$\text{OUT}[B]^3$	$\text{IN}[B]^4$	$\text{OUT}[B]^4$
B_1	111 0000	000 1111	111 0000	000 0000	111 0000
B_2	000 1100	110 0001	001 1110	111 0111	001 1110
B_3	000 0010	001 0000	000 1110	001 1110	000 1110
B_4	000 0001	100 1000	000 0111	001 1110	001 0111
EXIT	000 0000	000 0000	000 0111	000 0111	000 0111

Computing reaching definitions

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

A *least fixed point* is reached. $d_1 d_2 d_3 \;\; d_4 d_5 d_6 d_7$

Block	gen_B	$kill_B$	$\text{OUT}[B]^4$	$\text{IN}[B]^5$	$\text{OUT}[B]^5$
B_1	111 0000	000 1111	111 0000	000 0000	111 0000
B_2	000 1100	110 0001	001 1110	111 0111	001 1110
B_3	000 0010	001 0000	000 1110	001 1110	000 1110
B_4	000 0001	100 1000	001 0111	001 1110	001 0111
EXIT	000 0000	000 0000	000 0111	001 0111	001 0111

Computing reaching definitions

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

A *least fixed point* is reached. $d_1 d_2 d_3 \;\; d_4 d_5 d_6 d_7$

Block	gen_B	$kill_B$	$\text{OUT}[B]^5$	$\text{IN}[B]^6$	$\text{OUT}[B]^6$
B_1	111 0000	000 1111	111 0000	000 0000	111 0000
B_2	000 1100	110 0001	001 1110	111 0111	001 1110
B_3	000 0010	001 0000	000 1110	001 1110	000 1110
B_4	000 0001	100 1000	001 0111	001 1110	001 0111
EXIT	000 0000	000 0000	001 0111	001 0111	001 0111

Computing reaching definitions

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

A *least fixed point* is reached.

Block	gen_B	$kill_B$		IN$[B]$	OUT$[B]$
	$d_1 d_2 d_3$ $d_4 d_5 d_6 d_7$				
B_1	111 0000	000 1111		000 0000	111 0000
B_2	000 1100	110 0001		111 0111	001 1110
B_3	000 0010	001 0000		001 1110	000 1110
B_4	000 0001	100 1000		001 1110	001 0111
EXIT	000 0000	000 0000		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.

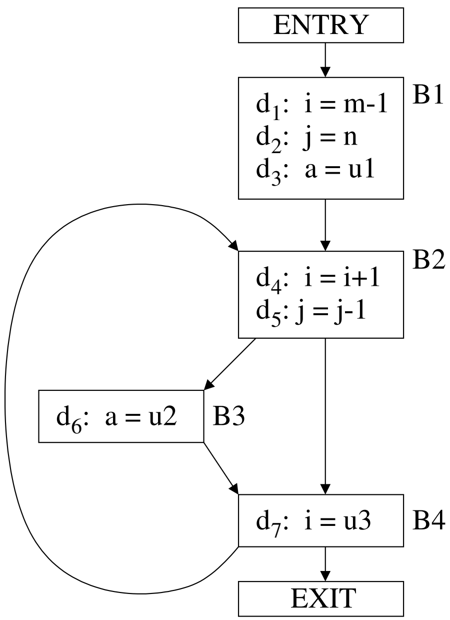
Live-variable analysis is essential for register allocation (only live variables must be kept in registers) and dead-code elimination.

Live-Variable analysis

Data-flow equations:

- IN$[B]$ and 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	use_B	def_B
$B1$	{m,n,u1}	{i,j,a}
$B2$	{i,j}	{}
$B3$	{u2}	{a}
$B4$	{u3}	{i}

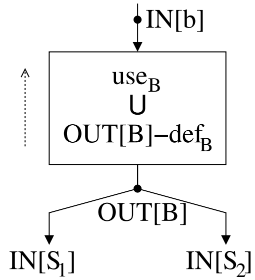
Live-Variable analysis

Data-flow equations:

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

For all other basic blocks B:

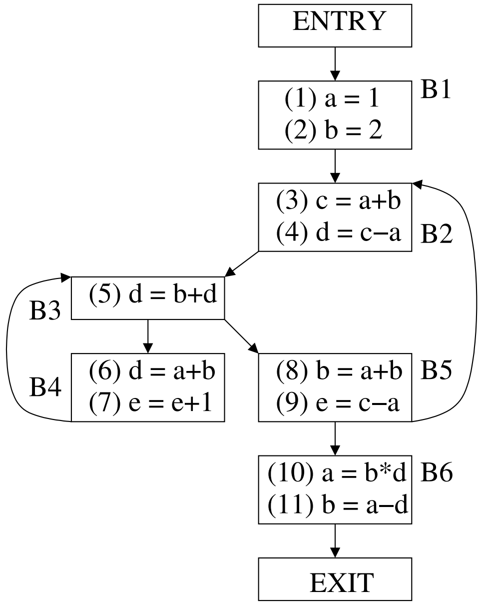
$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$$
$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$



Algorithm:

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

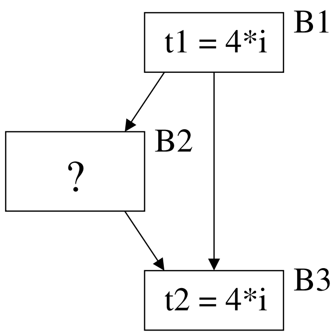
Live-Variable analysis



Exercise:
Find the variables alive at each point of the program.

Available expressions

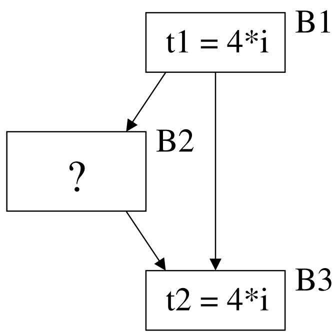
An expression $x + y$ is *available* at a point p if every path from the entry node to p evaluates $x + y$, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y.

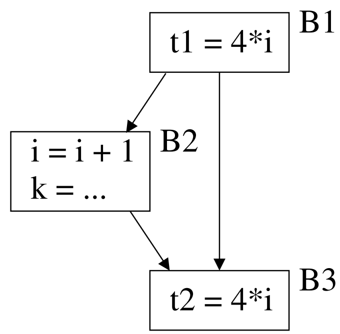


Is $4 * i$ available at the entry of B3 ?

Available expressions

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.

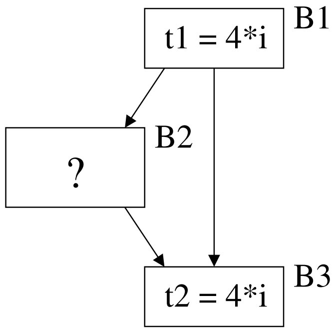


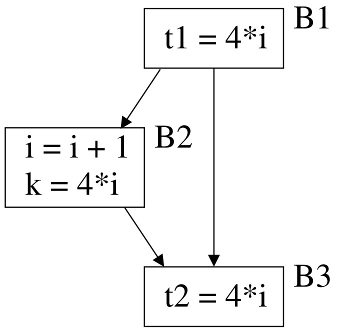


Is $4 * i$ available at the entry of B3 ?

Available expressions

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 `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}

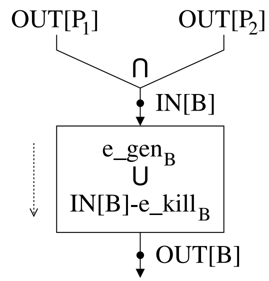
Available expressions

Data-flow equations:

$$\textsc{Out}[\text{ENTRY}] = \emptyset$$

For all other basic blocks B:

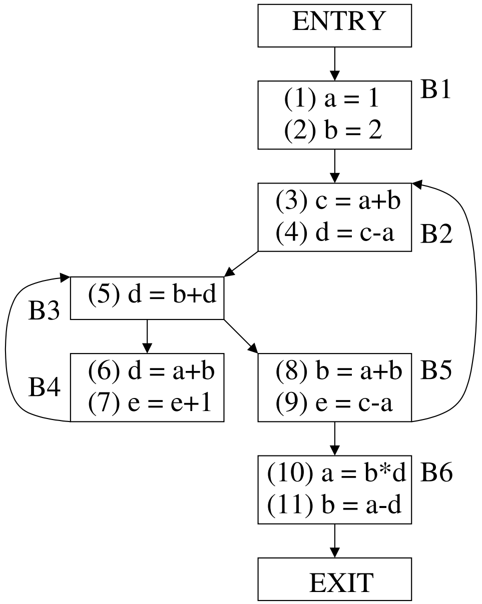
$$\begin{aligned} \textsc{Out}[B] &= e\_gen_B \cup (\textsc{In}[B] - e\_kill_B) \\ \textsc{In}[B] &= \bigcap_{P \text{ a predecessor of } B} \textsc{Out}[P] \end{aligned}$$



Algorithm:

- Initialize $\textsc{Out}[\textsc{Entry}] = \emptyset$ and $\textsc{Out}[B] = \mathbb{U}$ 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 = ...`*), 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.

Global Optimizations

- Elimination of common subexpressions
- Copy propagation
- Dead-code elimination
- Constant propagation

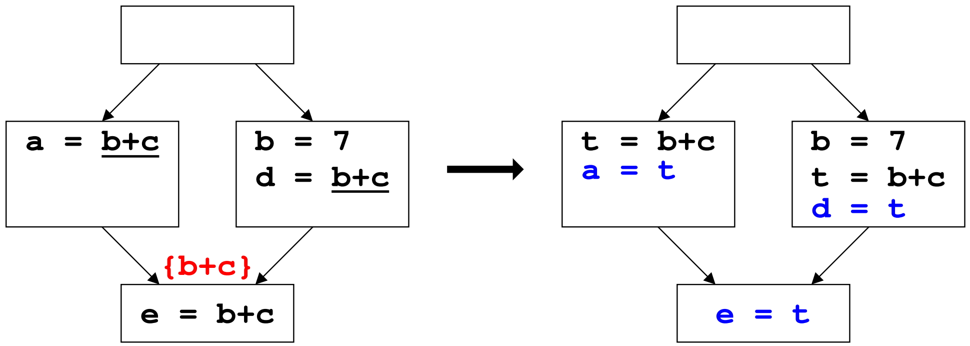
Elimination of common subexpressions

Let S be the instruction `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 `t`.
3. Substitute each found instruction `w=y+z` by
```
t = y+z
w = t
```
4. Substitute S by `x=t`.

Elimination of common subexpressions introduces *copies*.

Elimination of common subexpressions: example



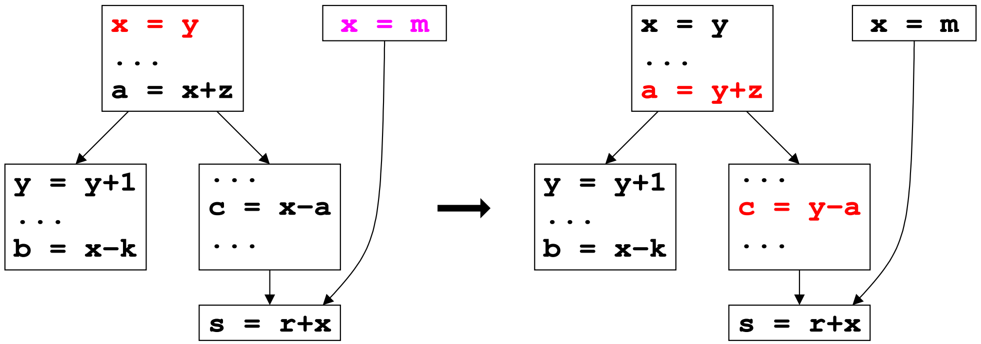
Copy propagation

Goal: Eliminate copy instructions (e.g. `x=y`).

Given an instruction `s: x=y` (definition):

- Calculate all *uses* of the definition.
- Every use of `x` can be substituted by `y` at the instruction `u` if:
  - `s` is the only definition of `x` that reaches `u`, and
  - for all paths `s` $\rightsquigarrow$ `u` there is no assignment to `y`.

Copy propagation: example



Dead-code elimination

Dead code may appear after some transformations.
Example:

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

Dead code often appears after copy propagation:

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

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.

Dead-code elimination

In some cases, the only *use* of a *def* is the same instruction. This situation may occur after the optimization of induction variables (see optimizations on loops).

Example:

```
i = 0;
...
while (t < n) {
  ...
  i = i+1
}
```

If the only *use* of `i = i+1` is the same instruction, then the instruction can be eliminated.

Notice that `i = 0` may also become dead after removing `i = i+1`.

Constant propagation

For every *use* of a variable, find all *defs* that reach that use. If all *defs* assign the same constant, then the use can be substituted by the constant.

Example:

```
a = 5                  a = 5
if (i < n) {           if (i < n) {
   ...                    ...
   a = 5        ⇒         a = 5
   ...                    ...
}                      }
...                    ...
b = a + 2              b = 5 + 2
```


Constant propagation

For every *use* of a variable, find all *defs* that reach that use. If all *defs* assign the same constant, then the use can be substituted by the constant.

Example:

```
a = 5                  a = 5
if (i < n) {           if (i < n) {
   ...                    ...
   a = 5        ⇒         a = 5
   ...                    ...
}                      }
...                    ...
b = a + 2              b = 7
```

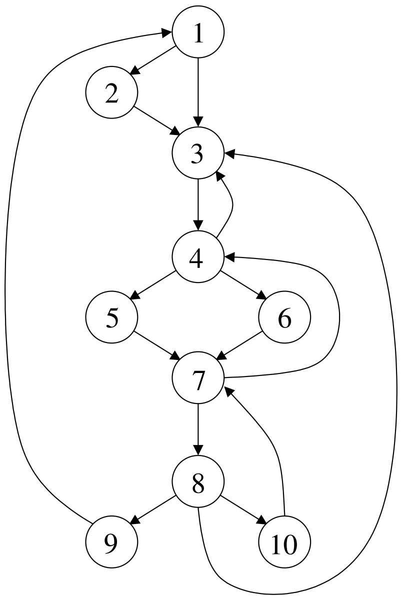

Optimization on loops

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

Outline:

- What is a loop?
  - Dominators and Dominator tree
  - Back edges and Natural loops
- Invariant computations.
- Induction variables and Reduction in strength.

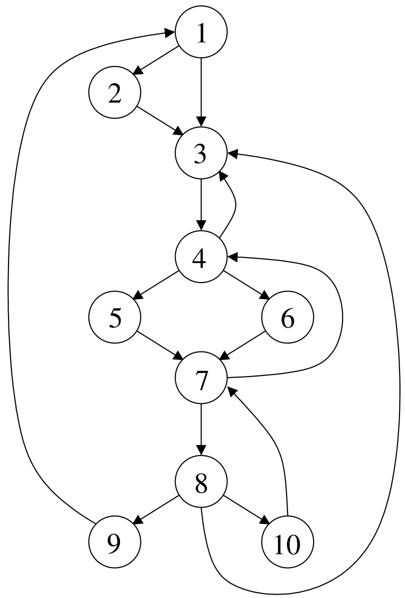
Dominators

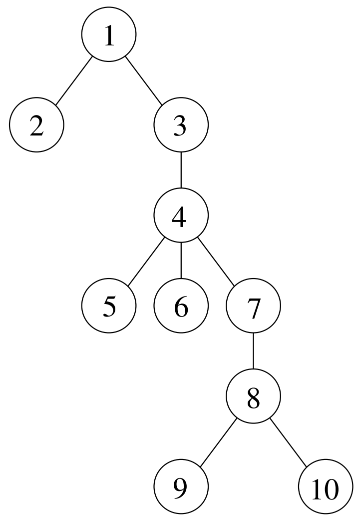


A node d of a flow graph *dominates* node n (d dom n) if every path from the entry node to n goes through d.

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:

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

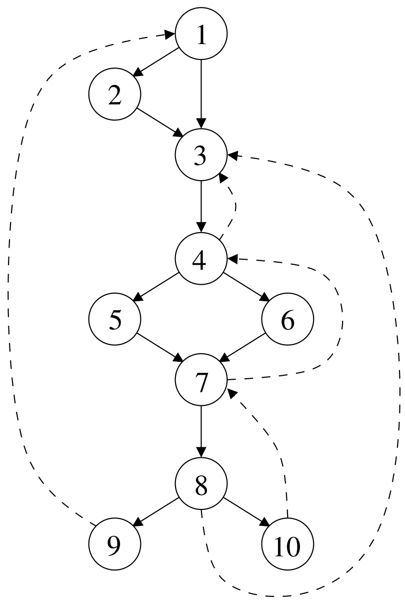
Algorithm:

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

Result:

- $\text{OUT}[n]$ is the set of nodes that dominate n.

Back edges



Back edge:

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

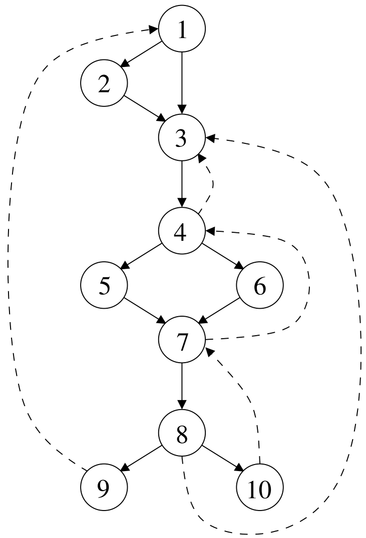
Natural loops

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 than enters the header.

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

Natural loops



Method to find the nodes of a natural loop:

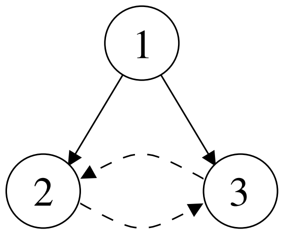
- Let $n \rightarrow d$ be the back edge.
- Mark d as visited.
- Do depth-first search from n on the reverse graph.
- The natural loop is the set of visited nodes.

Natural loops:

- $10 \rightarrow 7$: {7, 8, 10}
- $7 \rightarrow 4$: {4, 5, 6, 7, 8, 10}
- $4 \rightarrow 3$: {3, 4, 5, 6, 7, 8, 10}
- $8 \rightarrow 3$: {3, 4, 5, 6, 7, 8, 10}
- $9 \rightarrow 1$: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Reducible graphs

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



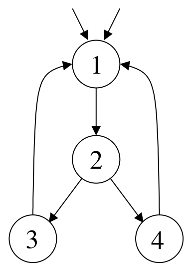
Nonreducible flow graph

Reducible graphs

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.

Innermost loops: loops that contain no other loops.

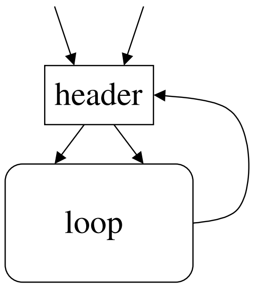


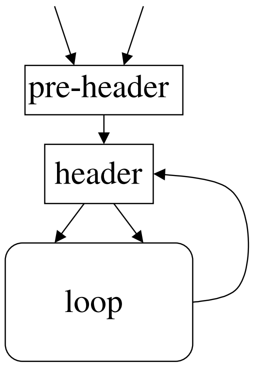
Two loops with the same header

Pre-header

Strategy:

- Create a pre-header before the header of the loop
- 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

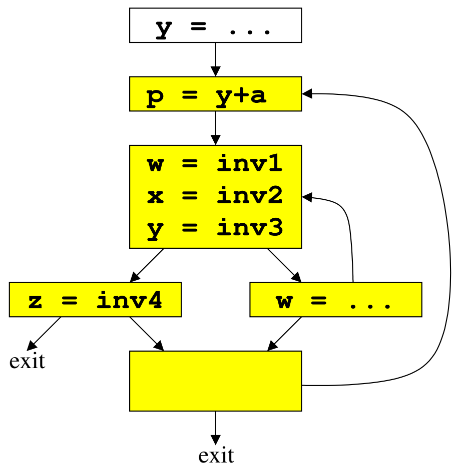
Assume that the statement `x=y+z` is invariant. The statement can be moved to the pre-header if:

- the block that contains the statement dominates all the exits of the loop, and
- there is no other definition of `x` inside the loop, and
- no other use of `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 `t=y+z` to the pre-header
- Keep the assignment `x=t` in the loop

Moving invariants to the pre-header



`inv1`, `inv2`, `inv3` and `inv4` represent invariant expressions.

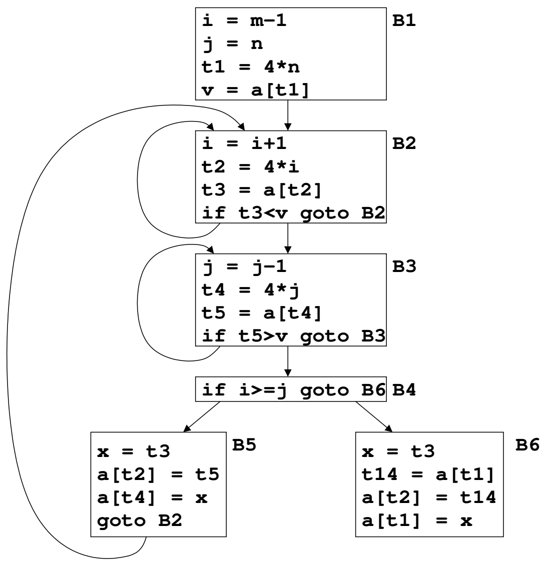
Only `x = inv2` can be extracted out of the loop "as it is".

The other invariants can be extracted using auxiliary variables.

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.

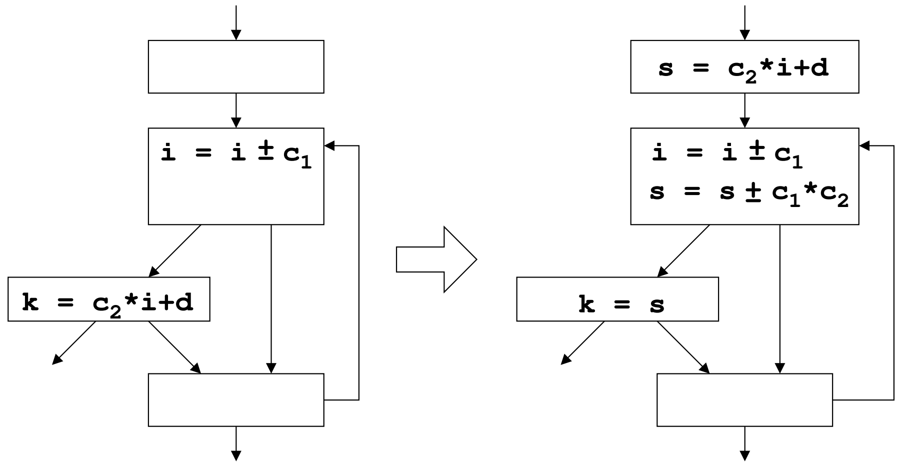
i and j are induction variables, but $t2$ and $t4$ are also induction variables.



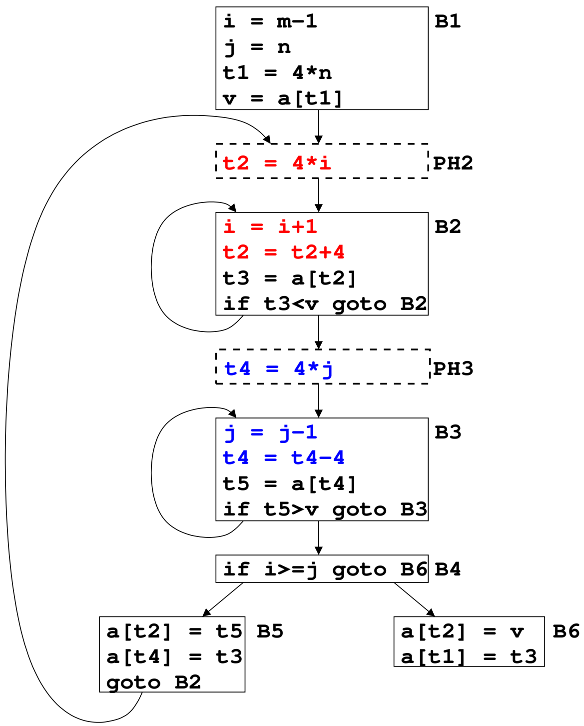
Induction variables and strength reduction

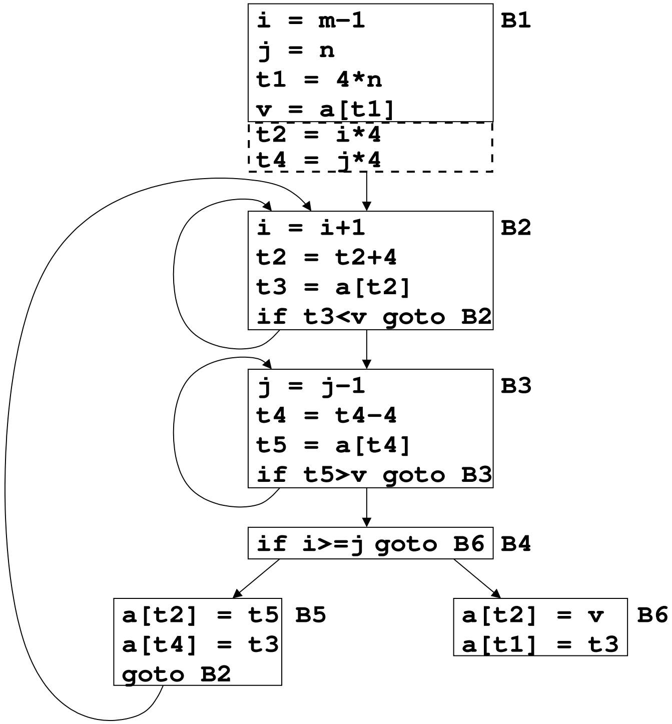
- Find variables with only one assignment inside the loop of the form $i = i \pm c_1$.
- Find variables with only one assignment inside the loop of the form $k = c_2 * i + d$, such that $i = i \pm c_1$ is the only definition reaching this assignment.
- Create a new variable s.
- Move $s = c_2 * i + d$ outside the loop (pre-header).
- Insert the instruction $s = s \pm (c_1 * c_2)$ immediately after $i = i \pm c_1$. Notice that $c_1 * c_2$ is a constant.
- Substitute by $k = s$ inside the loop.

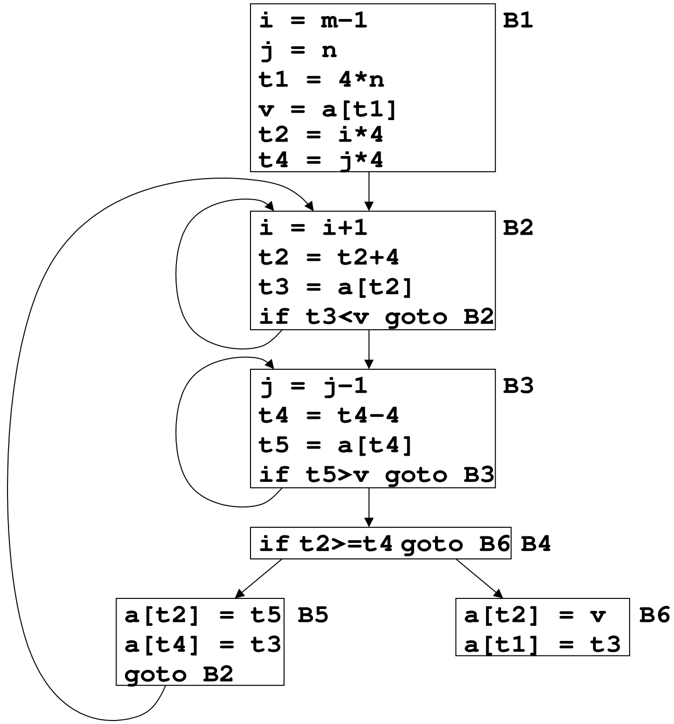
Induction variables and strength reduction



The product of constants $c_1 * c_2$ can be computed at compile time.
The copy `k=s` can be often removed by copy propagation.




i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
t2 = i*4
t4 = j*4
i = i+1
t2 = t2+4
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = t4-4
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
a[t2] = t5
a[t4] = t3
goto B2
B5
a[t2] = v
a[t1] = t3
B6


i = m-1
j = n
t1 = 4*n
v = a[t1]
t2 = i*4
t4 = j*4
B1
i = i+1
t2 = t2+4
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = t4-4
t5 = a[t4]
if t5>v goto B3
B3
if t2>=t4 goto B6
B4
a[t2] = t5
a[t4] = t3
goto B2
B5
a[t2] = v
a[t1] = t3
B6