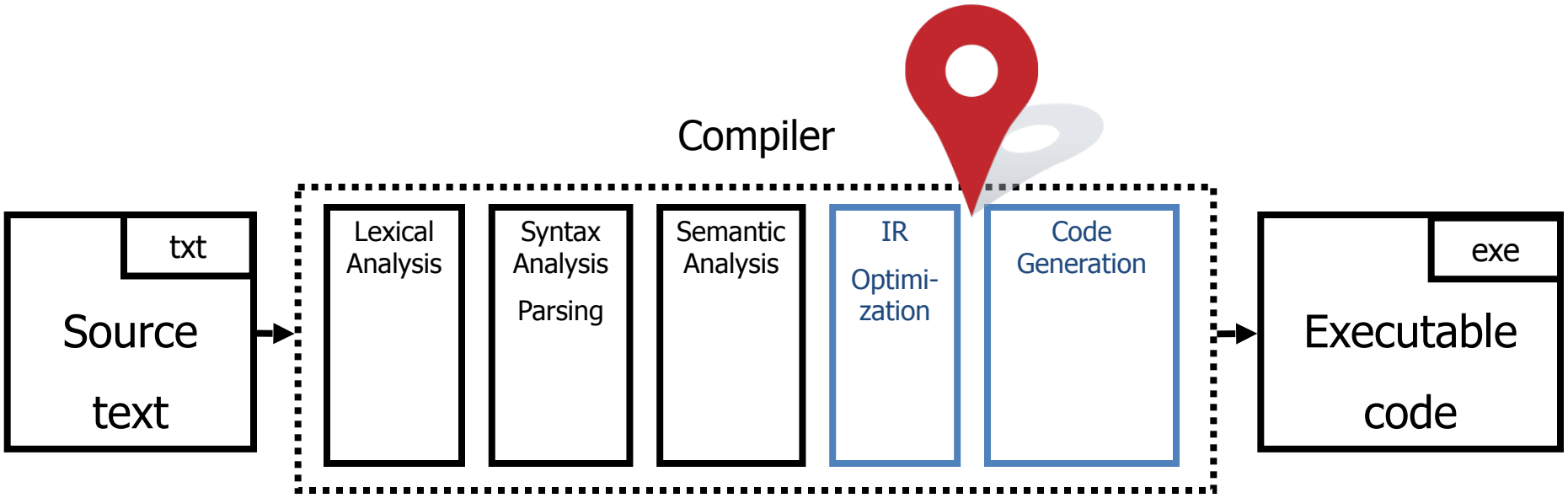
THEORY OF COMPILATION

LECTURE 08

You are here


Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code Generation
exe
Executable code

Optimizations

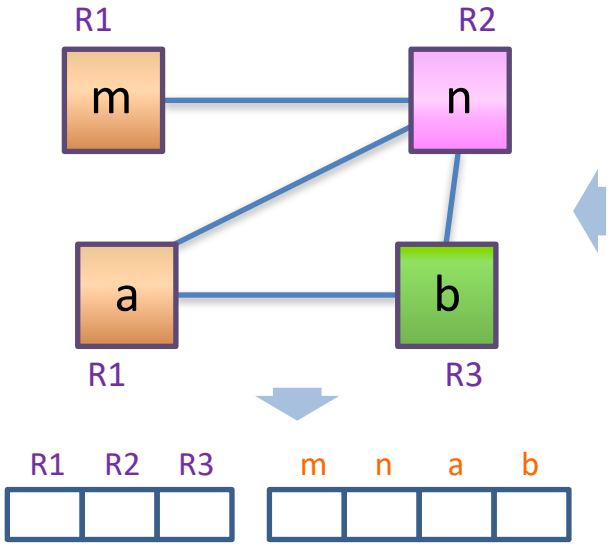
First, do no harm

- Improve program performance
- **Must maintain original semantics**
  - The observable behavior of the optimized program should be equivalent to that of the original program
- Typically cannot obtain the “optimal program”
- Can optimize various aspects of program execution
  - Memory
  - Time
  - Code size
  - Power
  - …

Why Optimize?

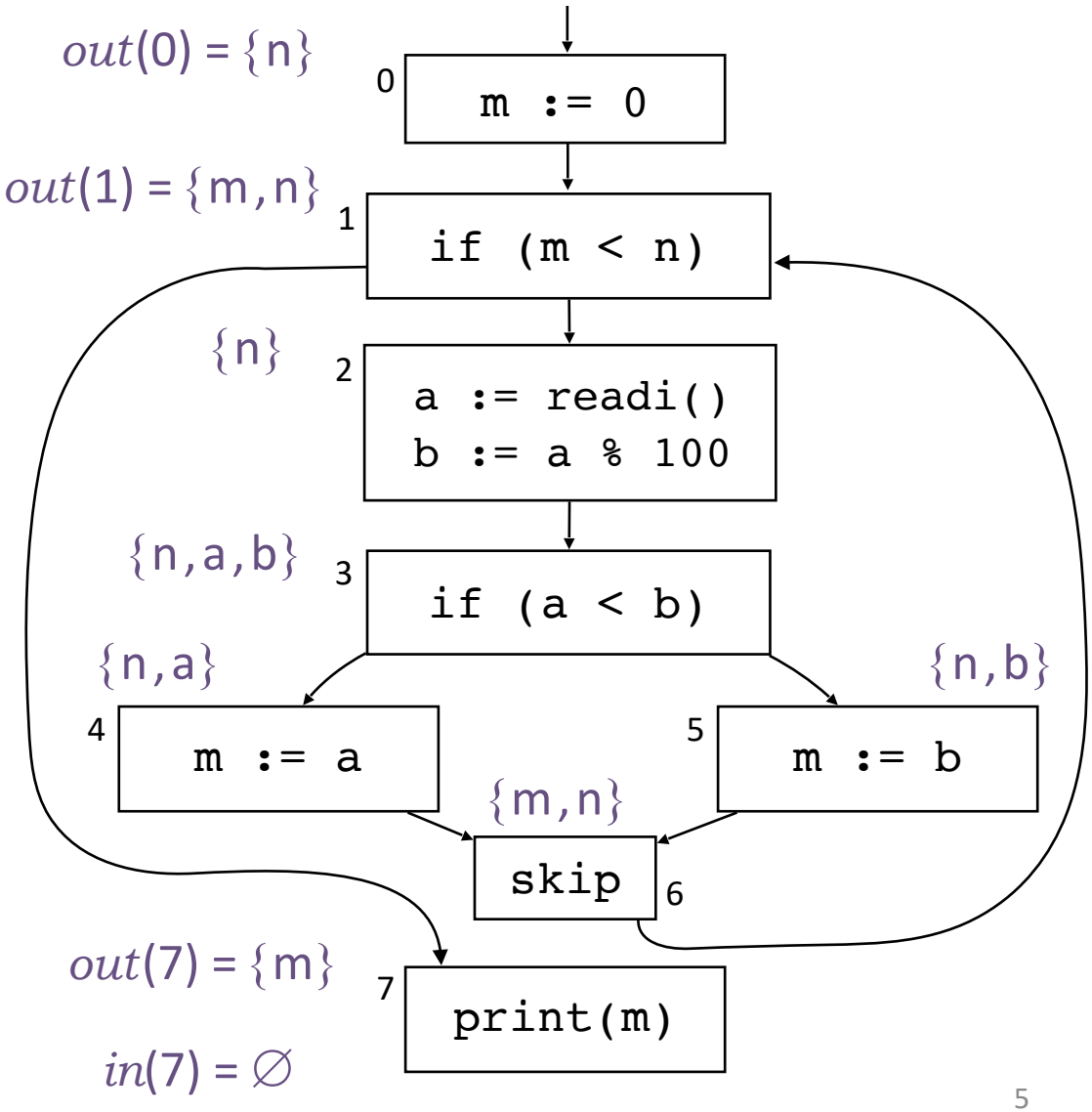
- Programmer may introduce inefficiencies
  - For various reasons; *e.g.* encapsulation in software design
- Compiler may introduce inefficiencies
  - Make earlier compiler stages easier to implement and maintain

Reminder: Data-flow Analysis (DFA)



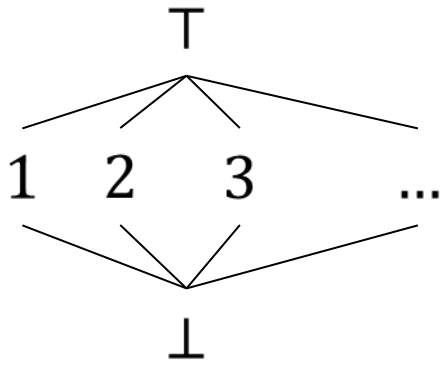
$L = \mathcal{P}(\text{Var})$ $\quad \sqsubseteq = \subseteq$

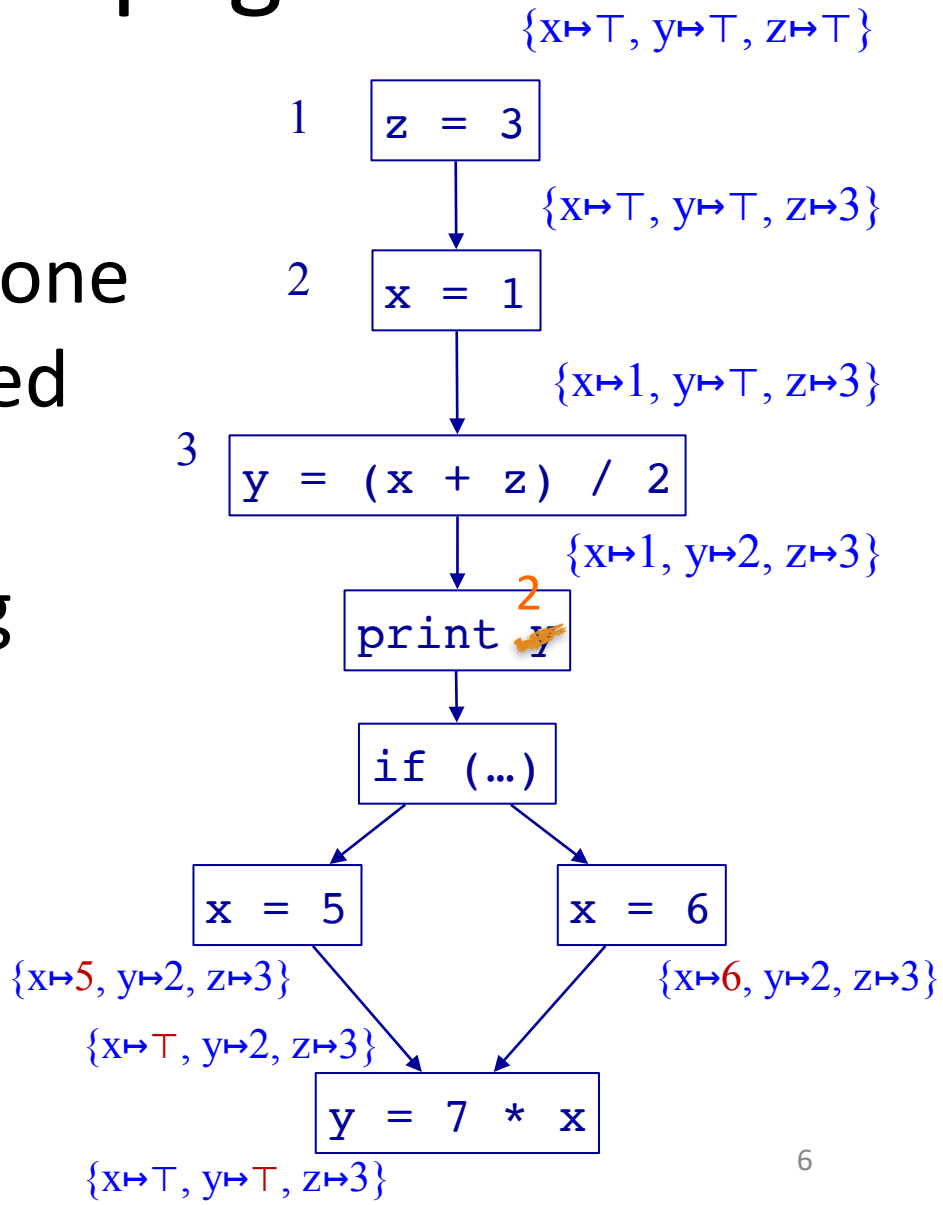
Stmt	out(ℓ)
x := *expr*	in(ℓ) \ {x} ∪ FV(*expr*)
if *cond*	in(ℓ) ∪ FV(*cond*)
print(*expr*)	in(ℓ) ∪ FV(*expr*)
skip	in(ℓ)



Constant Propagation

- Idea: if *on every run*, a variable can only take one value, it can be replaced with a constant.
- DFA over the following lattice:





Constant Propagation

Stmt	out(ℓ)
x = *expr*	in(ℓ)[x ↦ eval(*expr*, in(ℓ))]
anything else	in(ℓ)

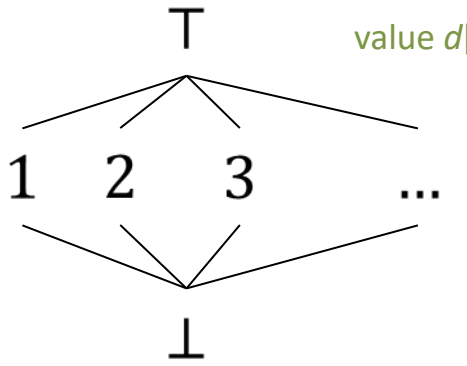
eval(*expr*, *d*) ≜

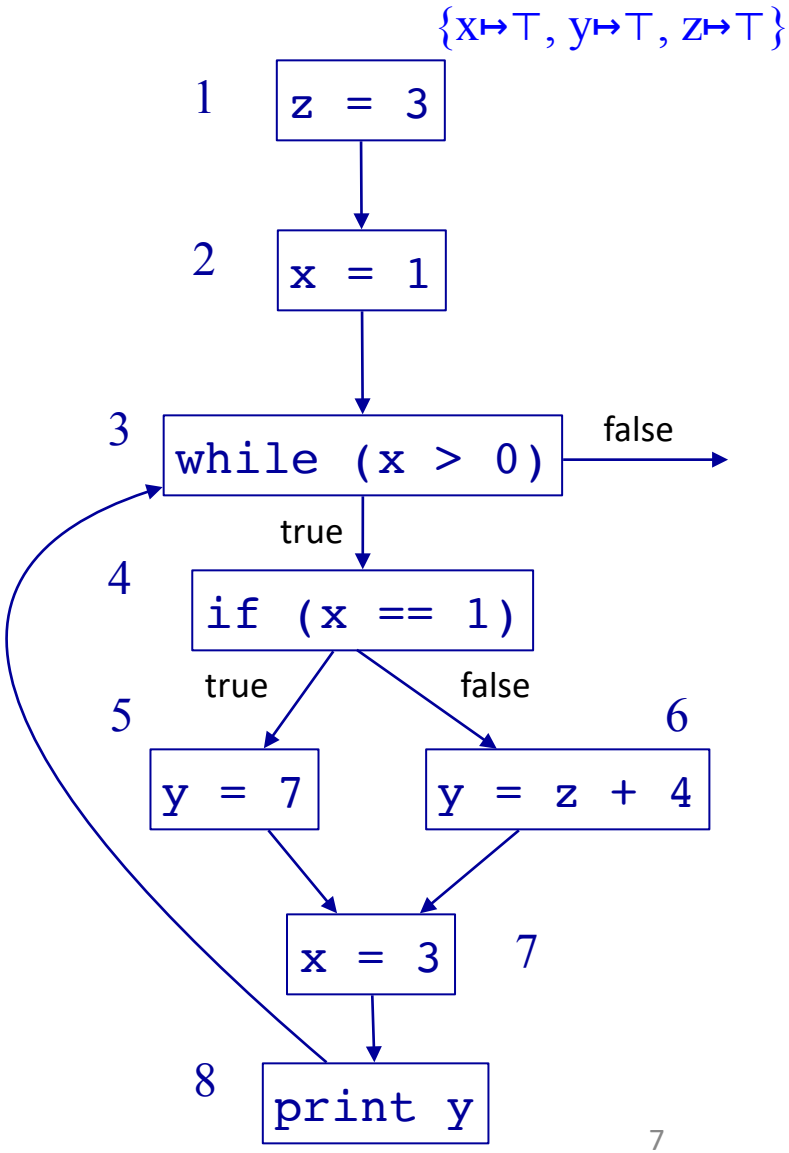
if $\forall v \in \text{FV}(expr), d[v] \neq \top$

then eval($expr[v \mapsto d[v]]$)

else $\top$

replace every v in *expr* with the (constant) value $d[v]$



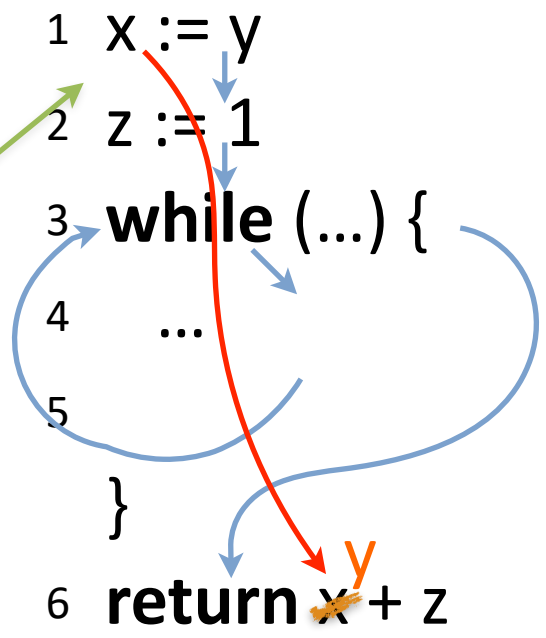


Copy Propagation

- Idea: Following an assignment x = y, try to use "y" whenever "x" is used (if possible)
  - Potentially makes "x" into a dead variable, saving the assignment x = y
  - Can be achieved using Reaching Definitions from previous lecture

if this is the only reaching definition of **x**

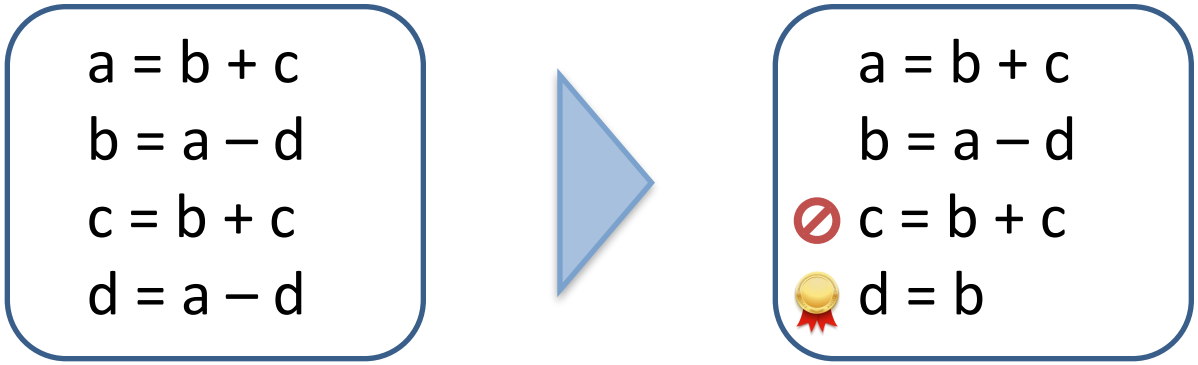
…and as long as there are no redefinitions of **y** along this path



  - Or a simpler *aliasing analysis* (in the tutorial)

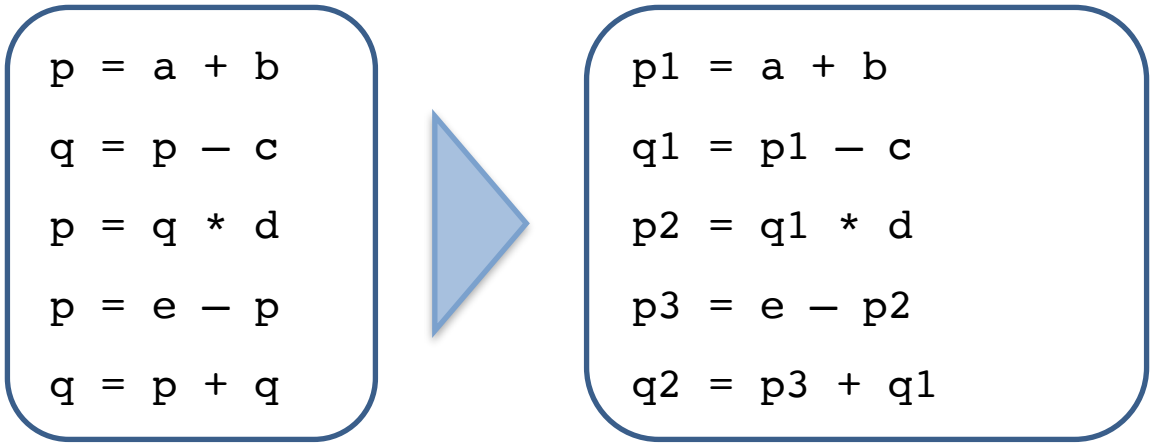
Common Subexpression Elimination

- Avoid recomputations
  - …but be careful:



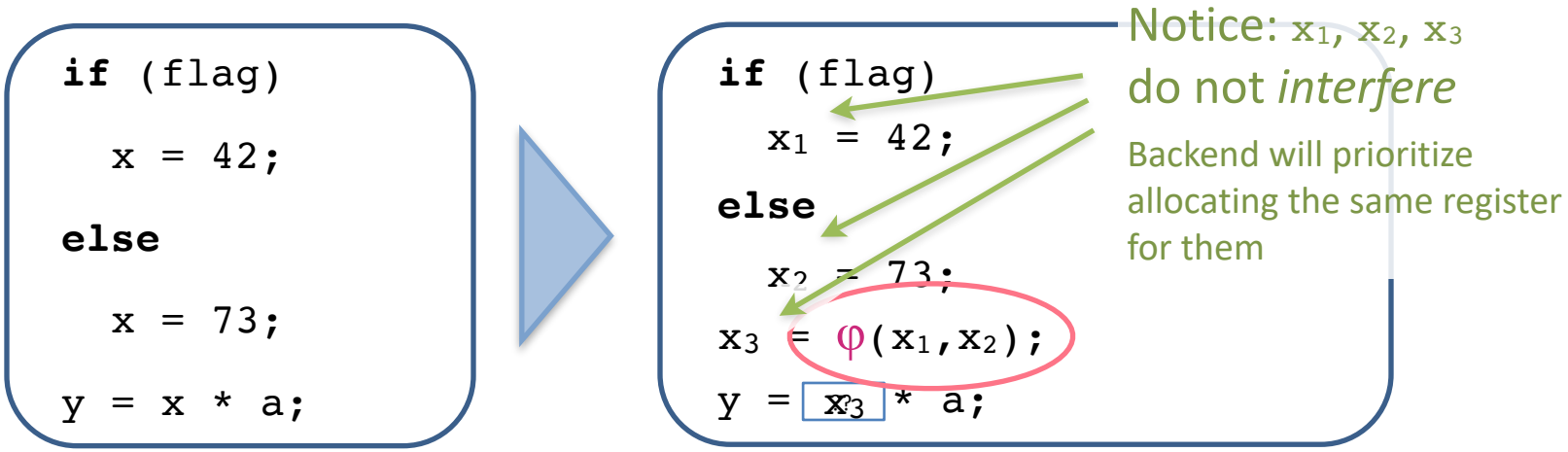
Static Single-Assignment Form (SSA)

- Every assignment writes to a distinct variable
- Every variable is only assigned once



SSA

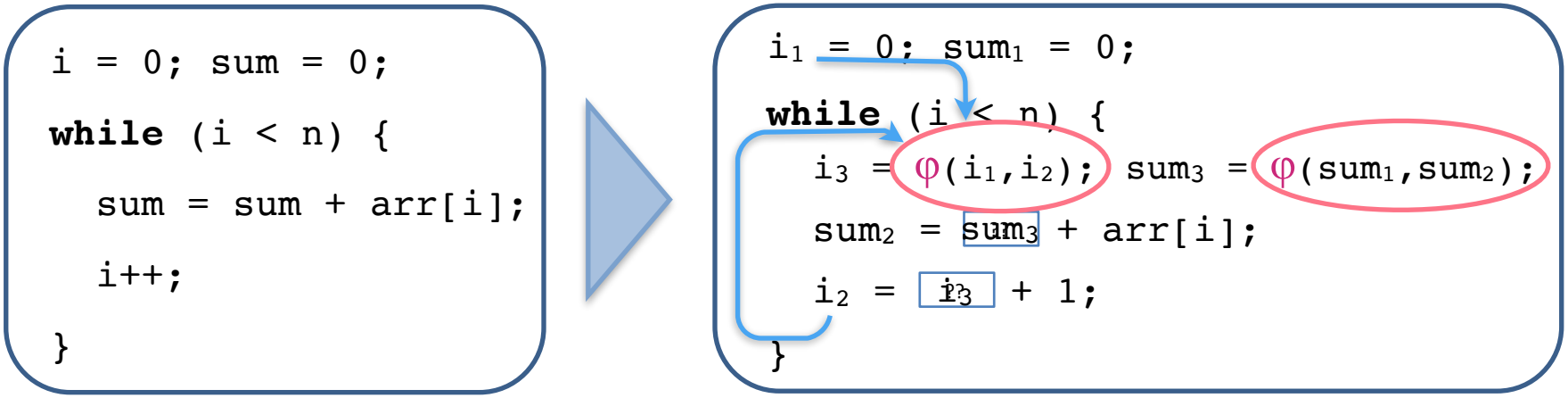
- Branches?



- φ (*phi*) function combines different definitions
- φ returns the value of x_1 if control passes through the true branch and the value of x_2 if it passes through the false branch

SSA

- Loops?



- φ returns the value of i_1 if entering the loop through pre-header (init block), i_2 if coming back from end of body

Transforming IR to SSA form

Using Reaching-Definitions Analysis

```
cond:
   if i < j goto then
   goto else

then:
   max = j
   goto exit

else:
   max = i
   goto exit

exit:
   print(max)
```

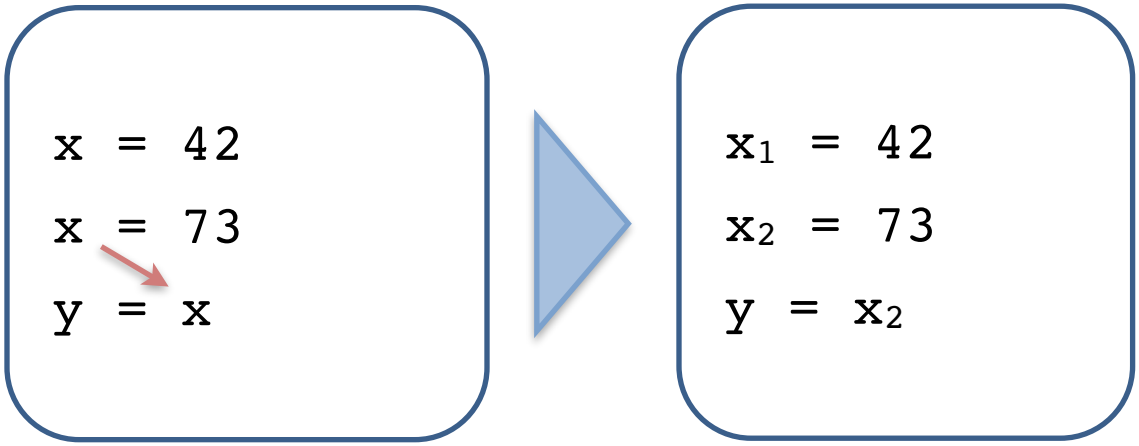
each **definition** is given a new name

```
cond:
   if i < j goto then
   goto else

then:
   max_then = j
   goto exit

else:
   max_else = i
   goto exit

exit:
   max_exit =
    φ(max_then, max_else)
   print(max_exit)
```

in(exit) = {(i, ...), (j, ...), (max, then), (max, else)}

- **uses** are modified to match the definition that reached them.
- in case of multiple reaching definitions, a φ operation is inserted

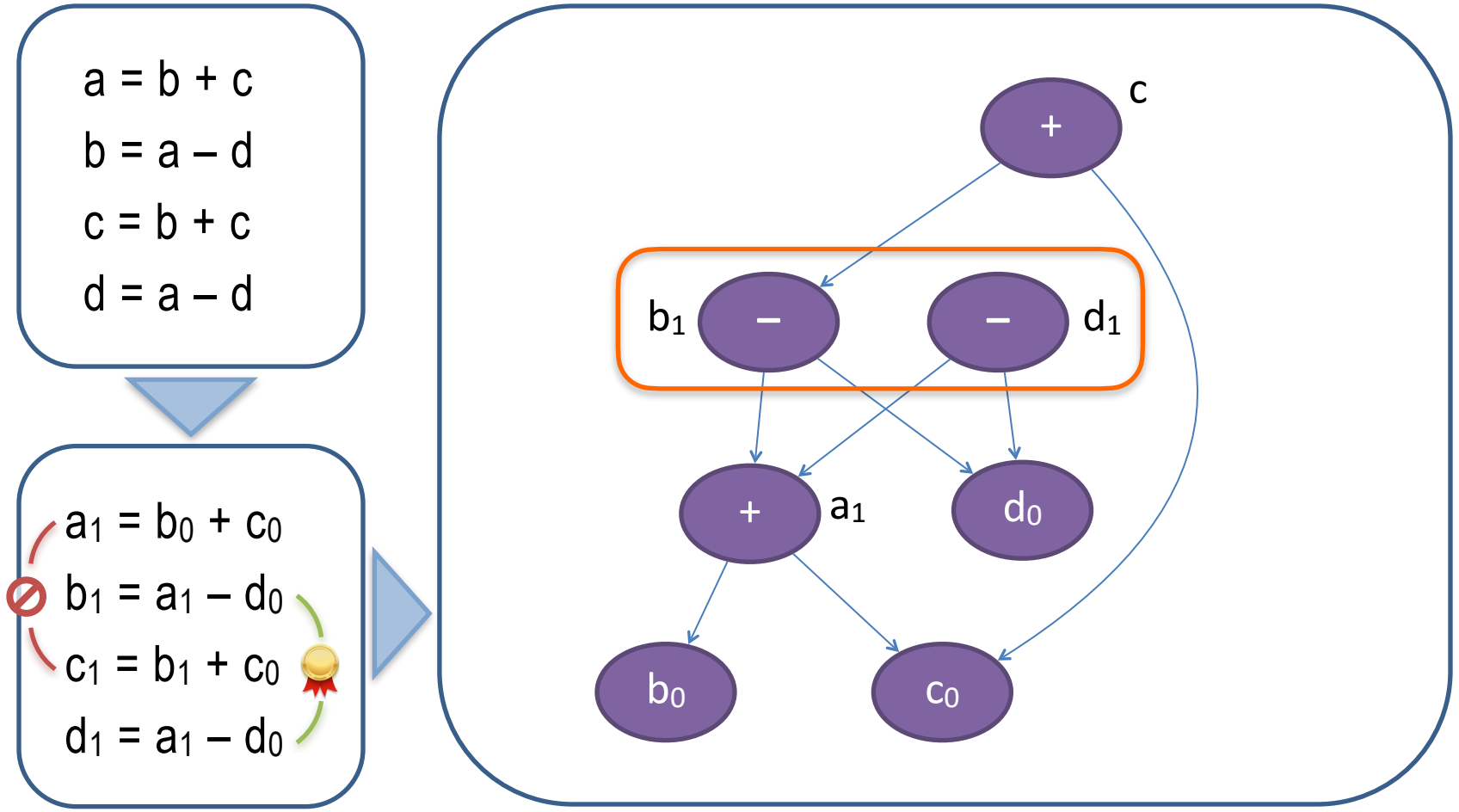
SSA — why should we care?



- Makes it easier to apply many optimizations
  - constant propagation, copy propagation, dead code elimination, common subexpression elimination…

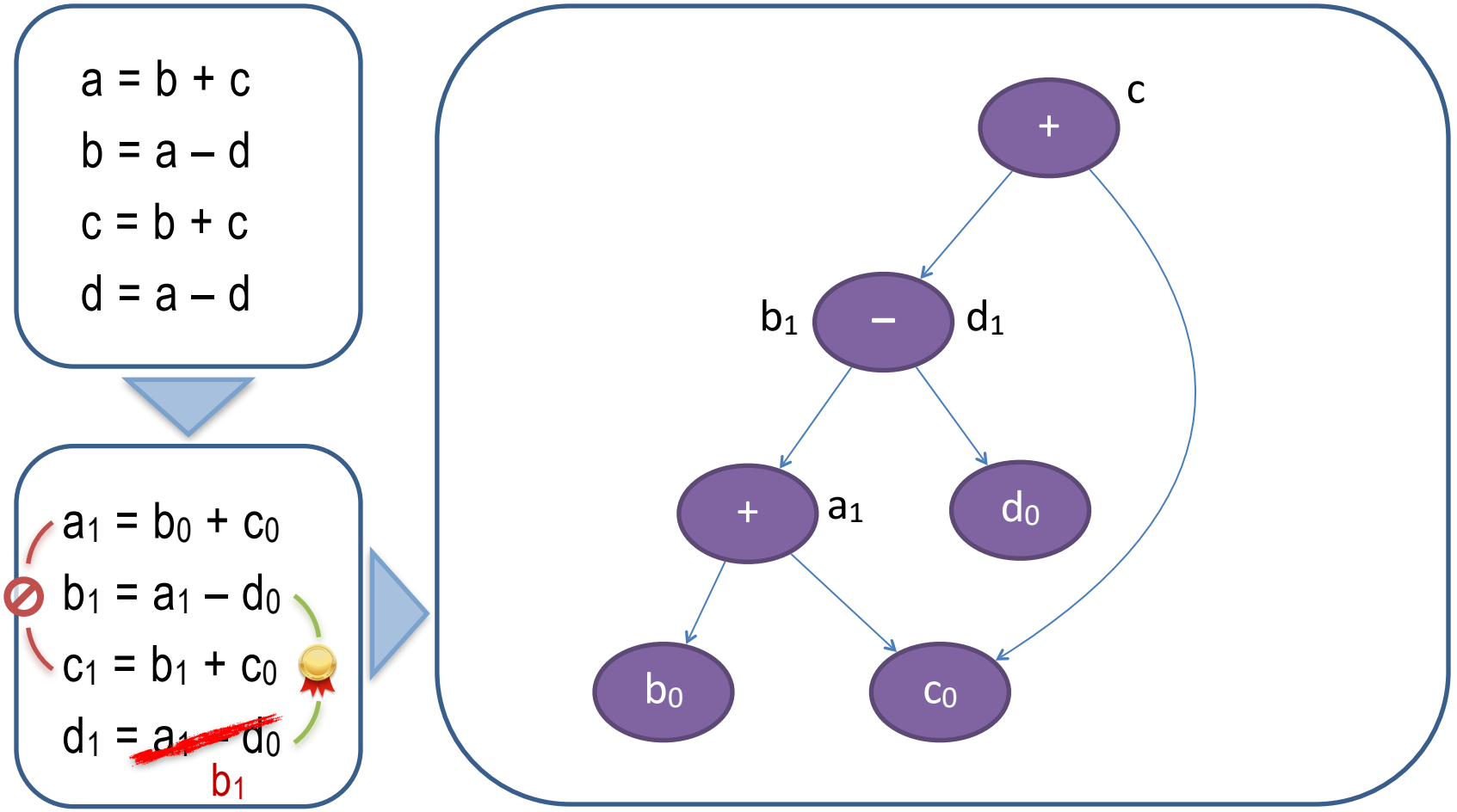
Common Subexpression Elimination

Illustration as a DAG



Common Subexpression Elimination

Illustration as a DAG



Common Subexpression Elimination

By viewing the basic block as a DAG

$a = b + c$

$b = b - d$

$c = c + d$

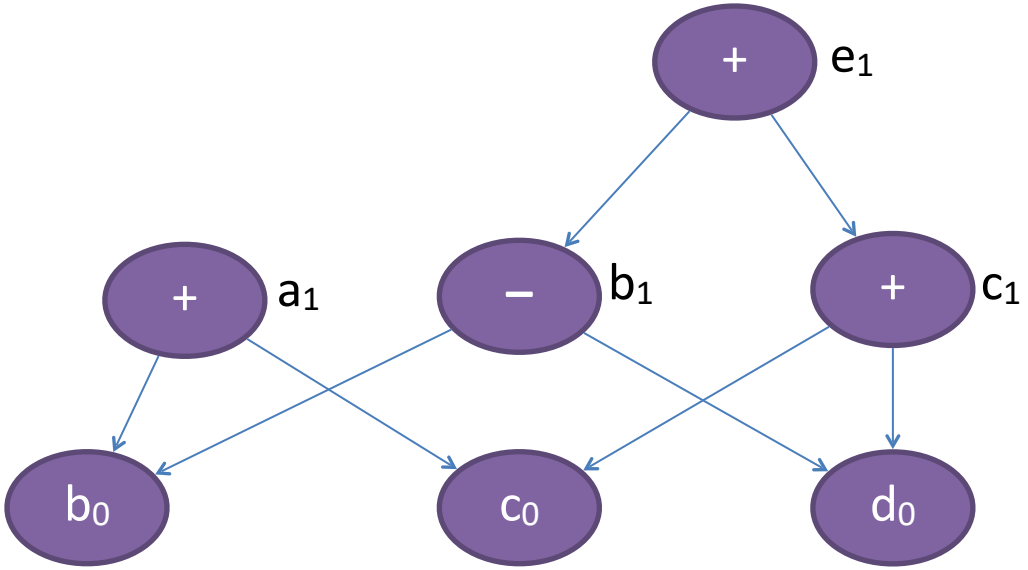
$e = b + c$

$a_1 = b_0 + c_0$

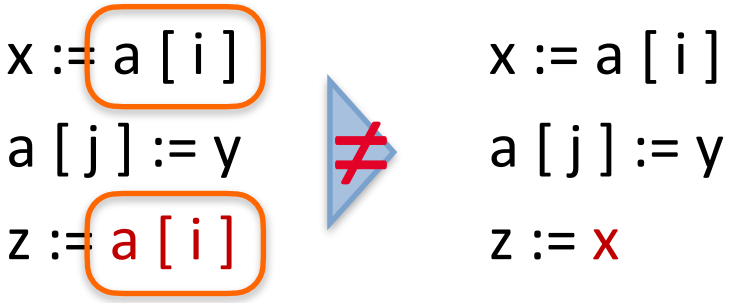
$b_1 = b_0 - d_0$

$c_1 = c_0 + d_0$

$e_1 = b_1 + c_1$

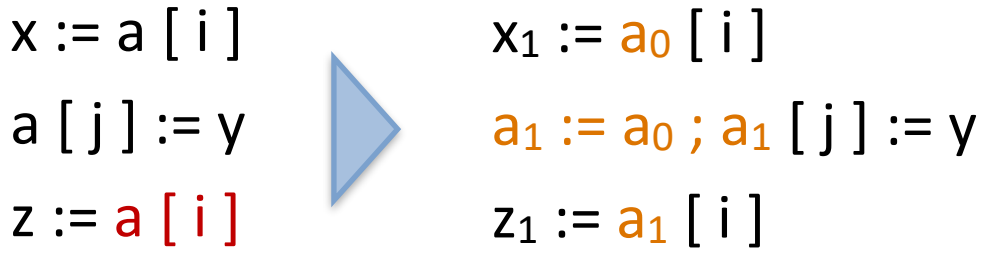


Aliasing is a problem



- We don't know whether i = j
- Have to handle it *conservatively*
  - Don't know that it's safe ⇒ must assume that it's (maybe) conflicting

Aliasing is a problem



- In SSA — whenever *any element* of an array is modified, generate a new name for the *entire* array.
  - (When generating the code, of course we won't copy the array; the same memory can be reused. This is only for the compiler's eyes.)
- Same thing happens with pointers
  - A major obstacle to effective optimization

Eliminating Redundant Code

```
1:  a := x
2:  x := a
3:  a := someFunction(a)
4:  x := someOtherFunction(a, x)
5:  if a > x goto (3)
```

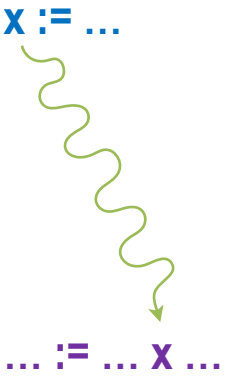
(2)?

```
1:  a := x

3:  a := someFunction(a)
4:  x := someOtherFunction(a, x)
5:  if a > x goto (3)
```

- Local optimizations are only performed within basic block boundaries

Eliminating Redundant Code

- Any assignment after which the assigned variable is dead, is redundant



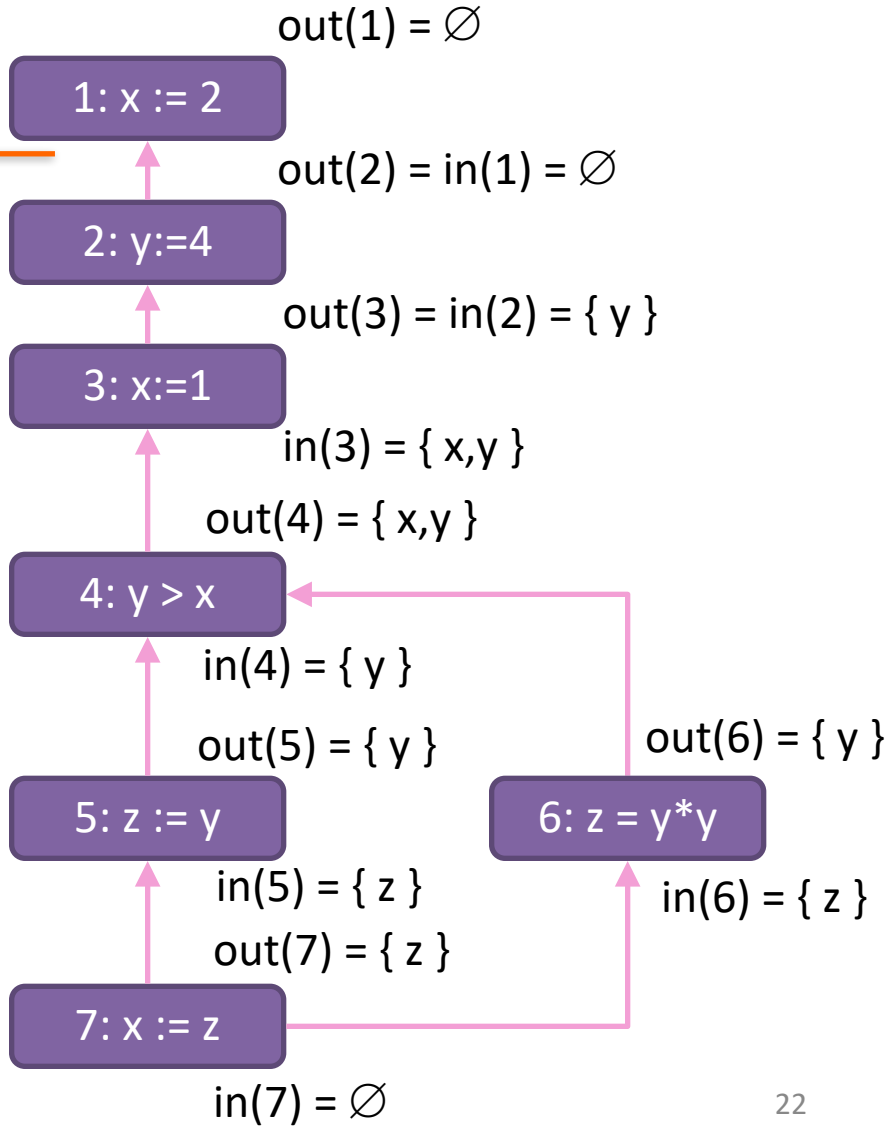

```
1:  y := 4
2:  x := 1
3:  if y <= x goto 7
4:  x := x + 1
5:  z := y
6:  goto 8
7:  z := y * y
8:  x := z
```

x ∉ in(4) = {y}

```
1:  y := 4;
2:  x := 1;
3:  if y <= x goto 6
4:  z := y
5:  goto 7
6:  z := y * y
7:  x := z
```

Live Variables Analysis — Reminder

```
1:  x := 2;
2:  y := 4;
3:  x := 1;
4:  if y > x
5:        then z := y
6:        else  z := y * y;
7:  x := z
```



Backward analysis using these transfer functions:

Stmt	out(ℓ)
x := *expr*	in(ℓ) \ { x } ∪ FV(*expr*)
if *cond*	in(ℓ) ∪ FV(*cond*)

Peephole Optimizations

- Optimizing long code sequences is hard
- A simple and efficient (but sub-optimal) alternative: discover improvements locally
  - Examine a small window ("peep hole") over the code
  - Identify local optimization opportunities
  - Rewrite code "in the window"
- Requires **manually written** rules
  - E.g. (local) algebraic simplifications

Peephole Optimizations

- Algebraic simplification

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

```
a = b + c
b = 0
c = c * b
e = a + c
```

```
a = b + c
b = 0
c = c * 0
e = a + c
```

```
a = b + c
b = 0
c = 0
e = a - 0
```

- Flow of control

```
1: if d == 1 goto 3
2: goto 4
3: print(x)
4: e = a + c
```

```
1: if d != 1 goto 3

2: print(x)
3: e = a + c
```

Loop Optimization

Loop Optimization

- Step 1: identify loops (header, body, back edge)

- Step 2: Apply loop optimizations:
  - Loop invariant code motion
  - Strength reduction of induction variable
  - Induction variable elimination

Loop Invariant Code Motion

- Identify repeated computation inside a loop that depends only on values that are not modified inside the loop
- Move computation outside of the loop

```
while (x – 3 < y) {
  // … instructions that do not change x
}
```

```
t1 = x – 3;
while (t1 < y) {
 //… instructions that do not change x or t1
}
```

Induction Variables & Strength Reduction

```
(1) i = 0;
(2) t1 = i * 4
(3) t2 = a[t1]
(4) if (t2 > 100) goto (19)
(5) …
      …
(17) i = i + 1
(18) goto (2)
(19) …
```

```
(1) i = 0;
(2) t1 = 0
(3) t2 = a[t1]
(4) if (t2 > 100) goto (20)
(5) …
      …
(17) i = i + 1
(18) t1 = t1 + 4
(19) goto (3)
(20) …
```

- Identify loop counters and relationship with other variables

Program Transformations

- Code motion and strength reduction can improve performance, provided that they **maintain program semantics**.
  - When does an optimization apply to a given program?
  - How can this be done automatically (and efficiently)?

Loop Detection

- Finding loops? Cakewalk!

~~Just look for "while", "do", "for", etc...~~

Or is it?!

```
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 ;
}
x = a [ i ] ; a [ i ] = a [ n ] ;
a [ n ] = x ;
```

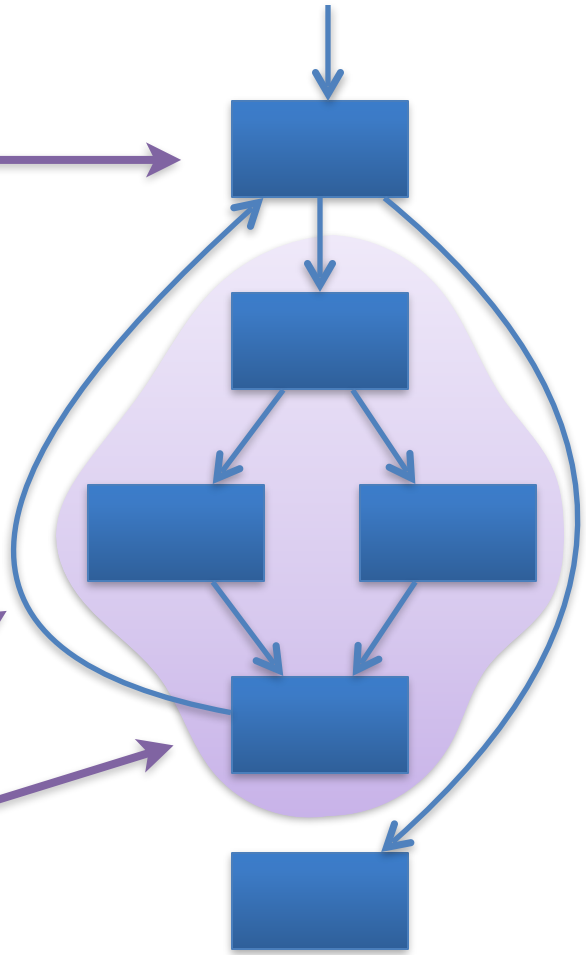
```
(6)  ...
(7)  t1 = i * 4
(8)  t2 = a[t1]
(9)  if (t2 > 100) goto (19)
(10) ...
     ...
(17) i = i + 1
(18) goto (7)
(19) ...
```

- Remember: optimizations are the responsibility of the **back-end.**

Loop Detection

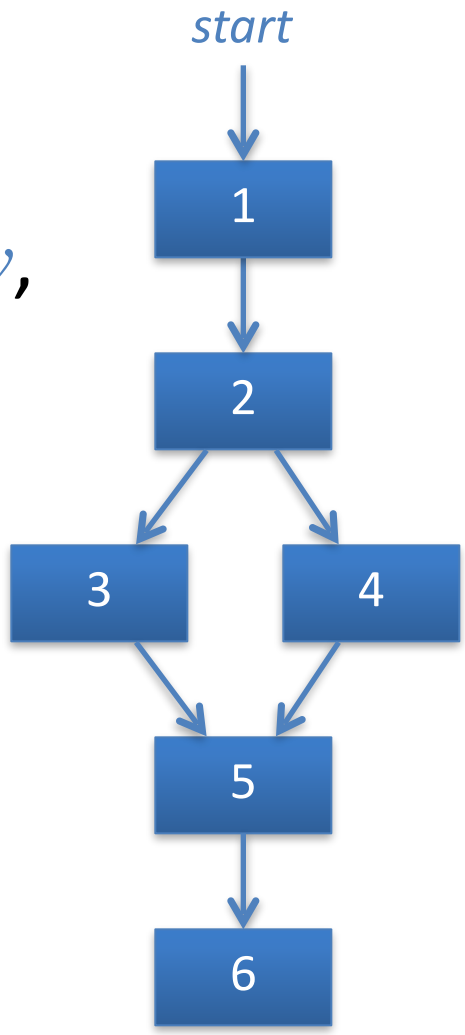
- Challenge: detect loops in (low-level) CFG

- Loop consists of:
  - Loop header
  - Back edge
  - Set of nodes comprising the body



Dominators

- Use *dominators* to identify loops
- Definition. Node u *dominates* node v, when *all paths* from *start* to v go through u.

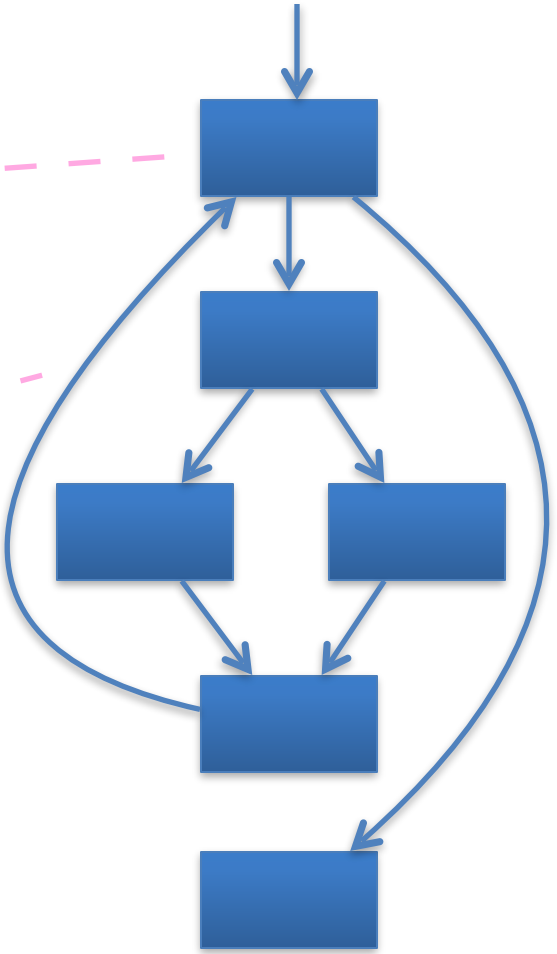


E.g.,

1 dominates {1, 2, 3, 4, 5, 6}
2 dominates {2, 3, 4, 5, 6}
3 dominates {3}
4 dominates {4}
5 dominates {5, 6}
6 dominates {6}

Loop Detection

- Header
  - dominates loop body
- Back edge
  - Target dominates source
- Find the back edge to identify the loop



Computing Dominators

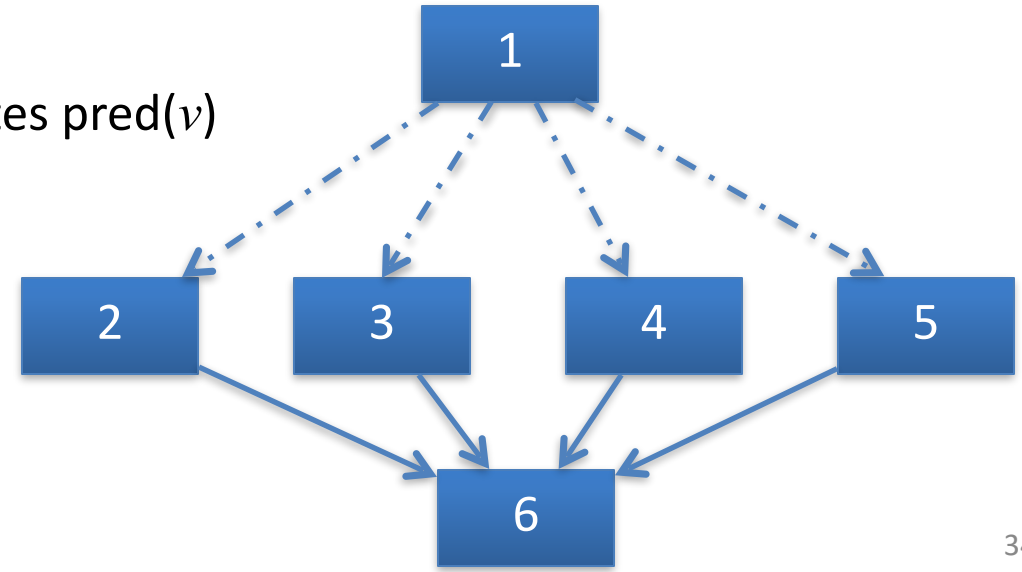
- Using Dataflow Analysis (previous lecture)
- Observation:

1 dominates 6 ⇔ 1 dominates {2, 3, 4, 5}

in general:

u dominates v ⇔ u dominates pred(v)

(pred(v) are v's predecessors in the CFG)



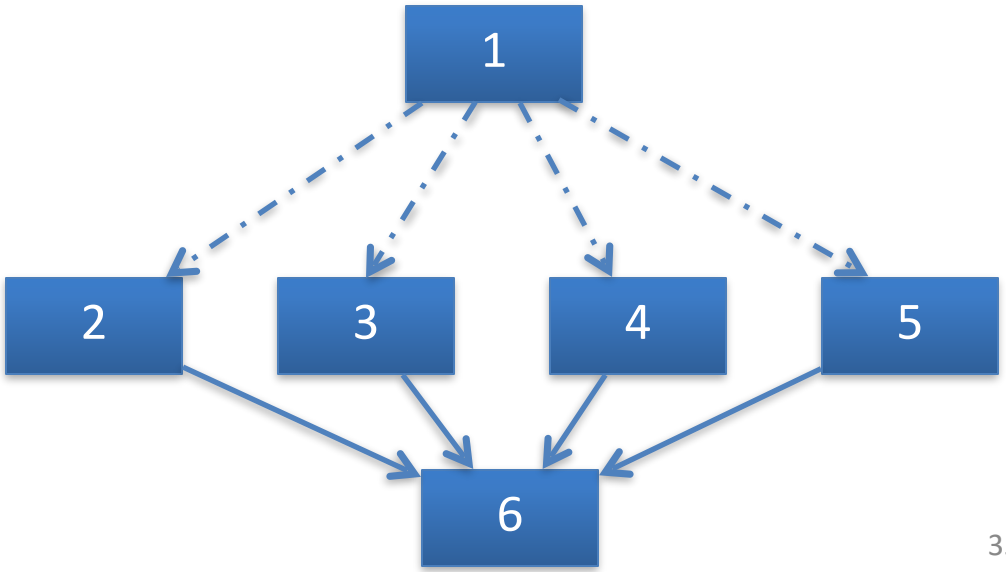
Computing Dominators

Using DFA!

- $\mathbb{L} = \mathcal{P}(Lab)$ (*Lab* is the set of block labels)
  - Intuition: $\text{out}(u)$ will be the set of dominators of u
- $\sqcup = \cap$ $(\sqsubseteq = \supseteq)$ $\Rightarrow \text{in}(\ell) = \cap\{\text{out}(u) \mid u \in \text{pred}(\ell)\}$
- Forward analysis, init = $\varnothing$
- Transfer functions:

  $\text{out}(\ell) = \text{in}(\ell) \cup \{\ell\}$

Regardless of what statement(s) are in the block ℓ 🤓



Computing Dominators

Using DFA!

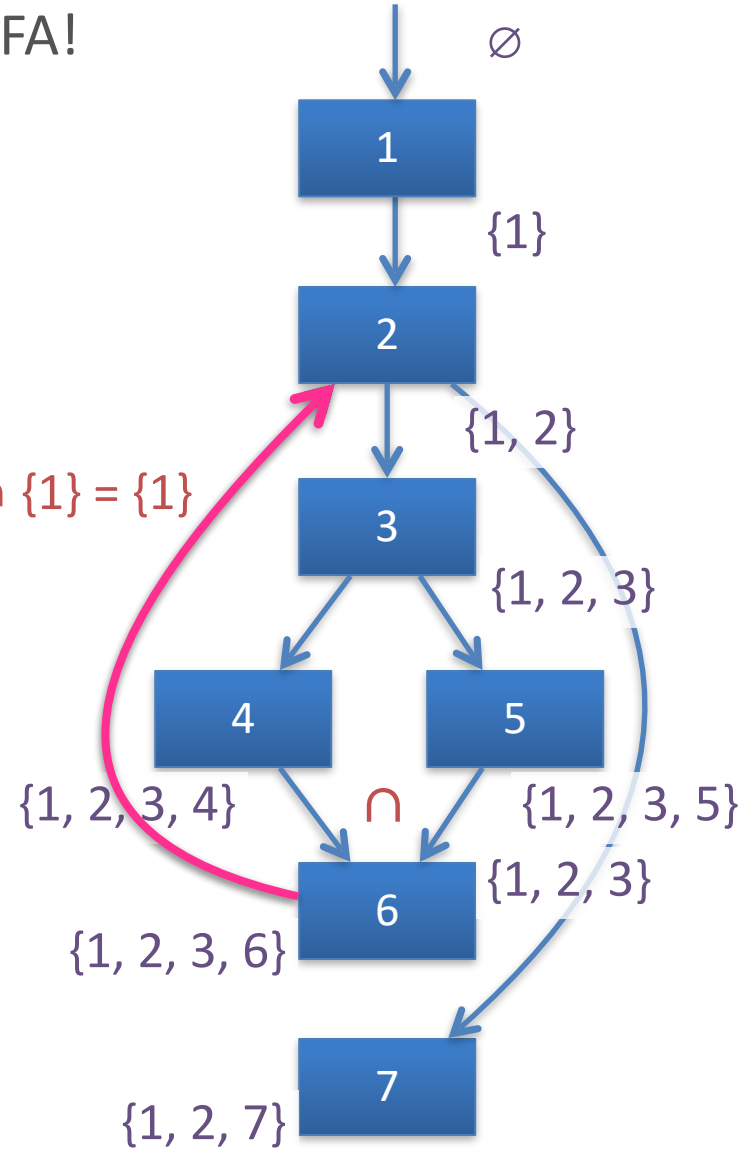
$\text{in}(1) = \varnothing$

$\text{in}(\ell) = \cap\{\text{out}(u) \mid u \in \text{pred}(\ell)\}$

$\text{out}(\ell) = \text{in}(\ell) \cup \{\ell\}$

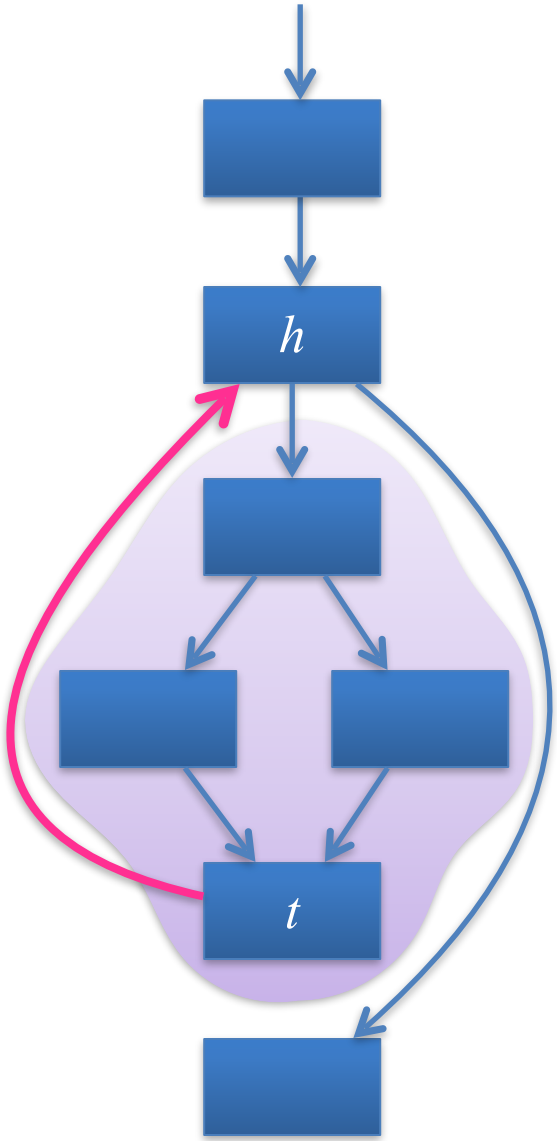
- Back edge
  - Edge $t \rightsquigarrow h$ *s.t.* $h \in \text{out}(t)$

    (*h dominates t*)



Identify Loops

- Back edge
  - Edge $t \rightsquigarrow h$ *s.t.* h dominates t
- Loop of a back edge $t \rightsquigarrow h$
  - h is the loop header
  - Loop body
    - all nodes that can reach t *without* going through h



Loop Optimization

✓ Step 1: identify loops (header, body, back edge)

- Step 2: Apply loop optimizations:
  - Loop invariant code motion
  - Strength reduction of induction variable
  - Induction variable elimination

Code Motion — Loop Invariant

```
for (i=0; i <10; ++i)
    a[i] = 10*i + x*x;
```

```
t = x*x;
for (i=0; i <10; ++i)
    a[i] = 10*i + t;
```

- An expression is loop invariant if it does not change throughout the execution of the loop
  - Can be hoisted — computed only once

Loop Invariant Computation

- a := b ◇ c is loop invariant if b and c:
  - Are constant, or
  - Have only definitions outside the loop, or
  - Have only one definition (each) that are themselves loop invariant

- Reaching definitions analysis can be used (DFA)
  - (with a small variation)

Loop Invariant Computation

INV := ∅

repeat

for each definition ℓ in loop such that $\ell \notin$ INV

if each operand in ℓ:

is constant, **or**

has no definitions inside the loop, **or**

has exactly one definition d $\in$ INV

then INV := INV $\cup$ {ℓ}

until no changes in INV

Code Motion — Loop Invariant

- Assume a := b ◇ c is loop invariant
  - When can we hoist it out of the loop?
    - *(pssst: not always)*

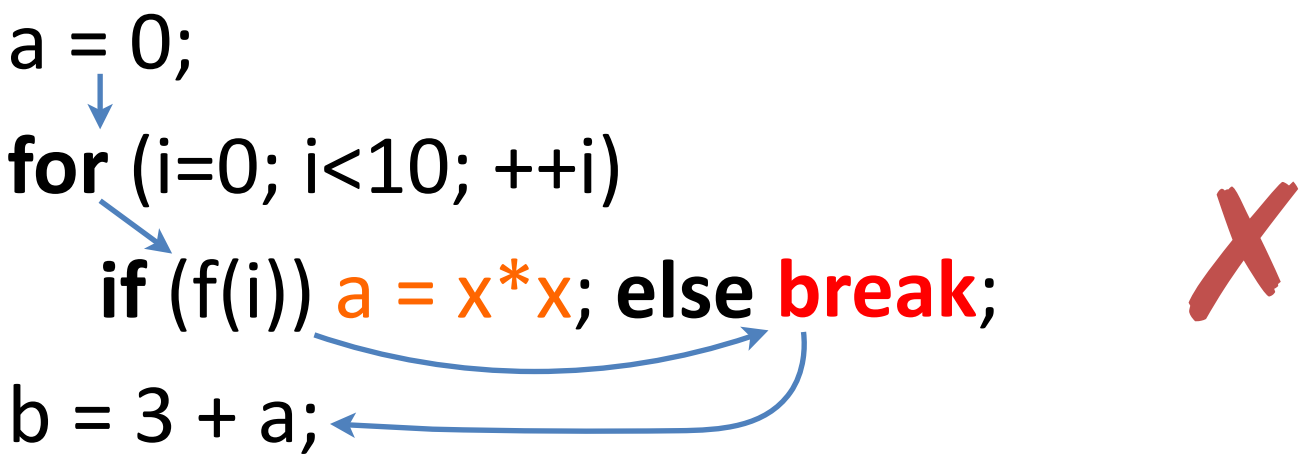
Valid Code Motion

Move of d: a := b ◇ c to pre-header is valid when:

1. d dominates all loop exits where a is live
2. d is the only definition of a in the loop
3. All uses of a in loop can only be reached from d

Valid Code Motion

1. d dominates all loop exits where a is live



Use liveness + dominator info to check whether each loop exit is dominated by d

Valid Code Motion

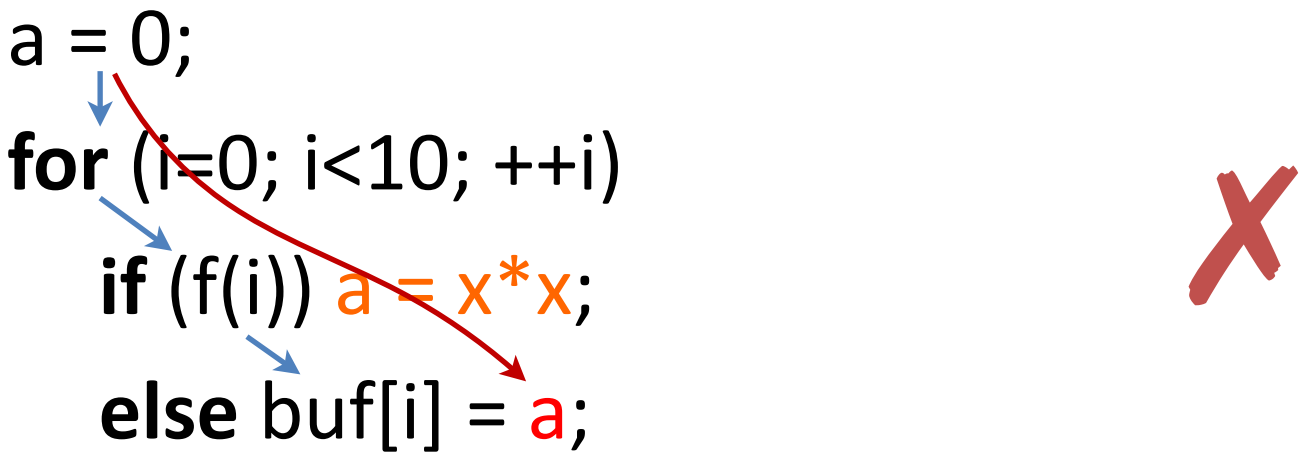
2. d is the only definition of a in the loop

```
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
    else a = 0;
```

Scan loop body for any other definitions of a

Valid Code Motion

3. All uses of a in loop can only be reached from d



Apply reaching definitions analysis and check each use of a for any definitions of a other than d

Valid Code Motion

d: a := b ◇ c

1. d dominates all loop exits where a is live
2. d is the only definition of a in the loop
3. All uses of a in loop can only be reached from d

```
for (i=0; i<10; ++i)
    a[i] = 10*i + x*x;
    t2 := x*x
    a[i] := t1 + t2
```

pre-header

```
t2 := x*x
for (i=0; i<10; ++i)
    t1 := 10*i
    a[i] := t1 + t2
```

(t1, t2 ∉)

Strength Reduction

- If an expression *e* is **not** loop invariant, code motion may **still** be applied:
  - Compute *e* once in loop pre-header
  - Whenever any variable occurring in *e* is modified, update computed value
- When is this beneficial?
  - If the cost of updating *e* is smaller ("weaker") than recomputing *e* anew

Strength Reduction

- Differencing rules:

‘+’ ≺ ‘*’

$t_1 := c * i$

⋮

$i := i + 1$

$t_1 := c * i$

(c is loop invariant)

$t_1 := c * i$

⋮

$i := i + 1$

$t_1 := t_1 + c$

$t_1 := i * j$

⋮

$i := i + 1$

$t_1 := i * j$

⋮

$j := j + 1$

$t_1 := i * j$

(at all changes to i, j)

$t_1 := i * j$

⋮

$i := i + 1$

$t_1 := t_1 + j$

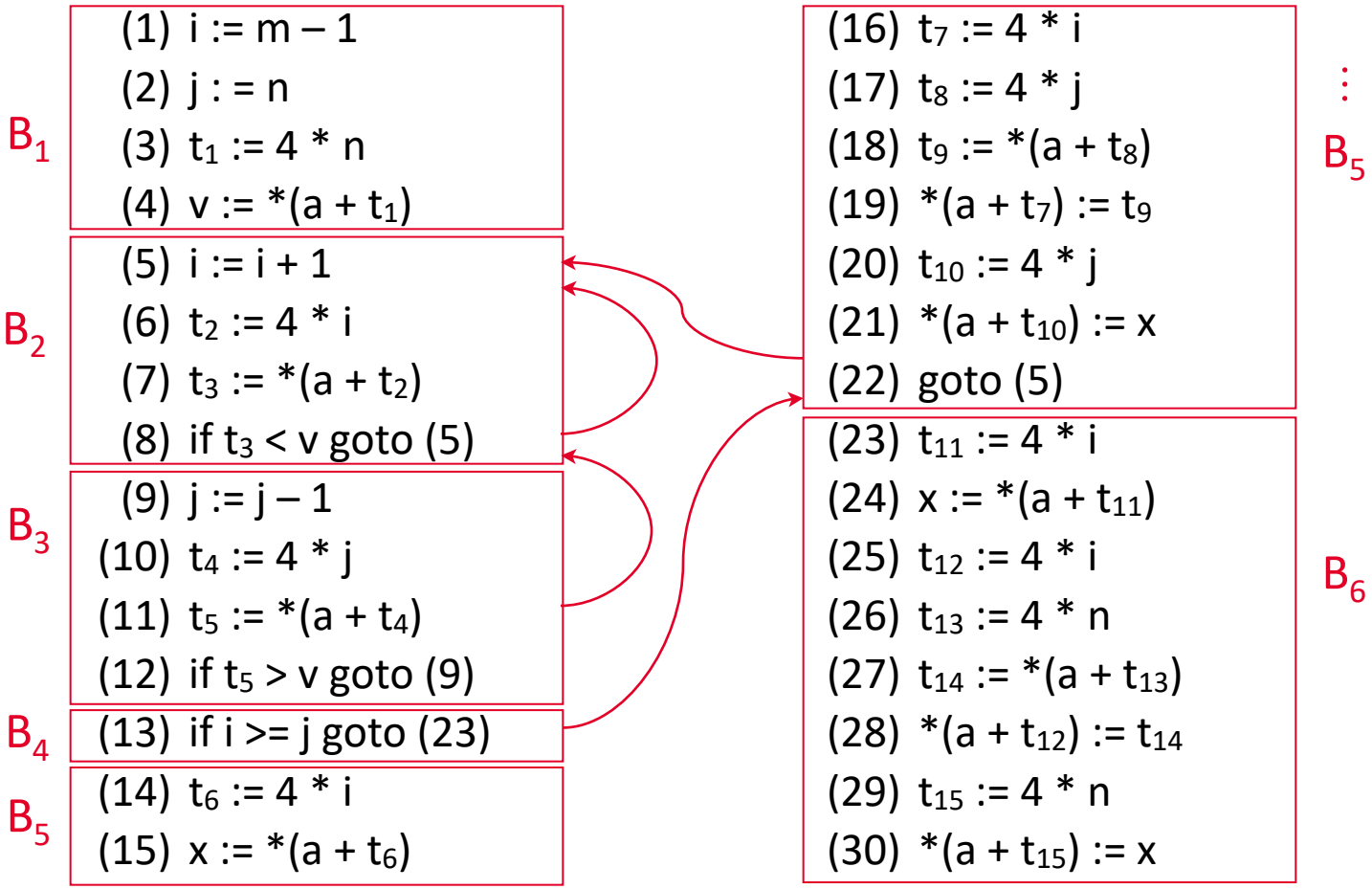
⋮

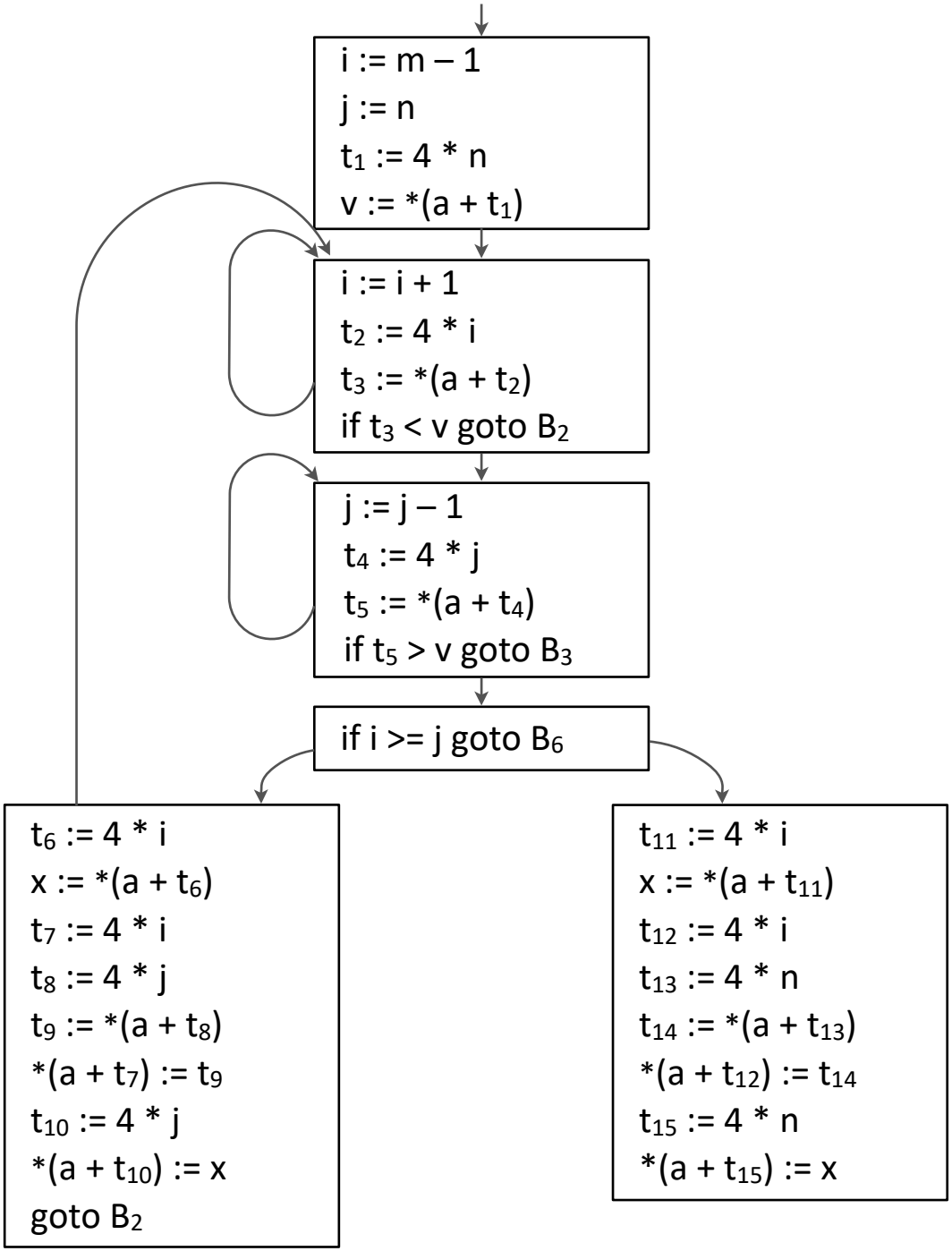
$j := j + 1$

$t_1 := t_1 + i$

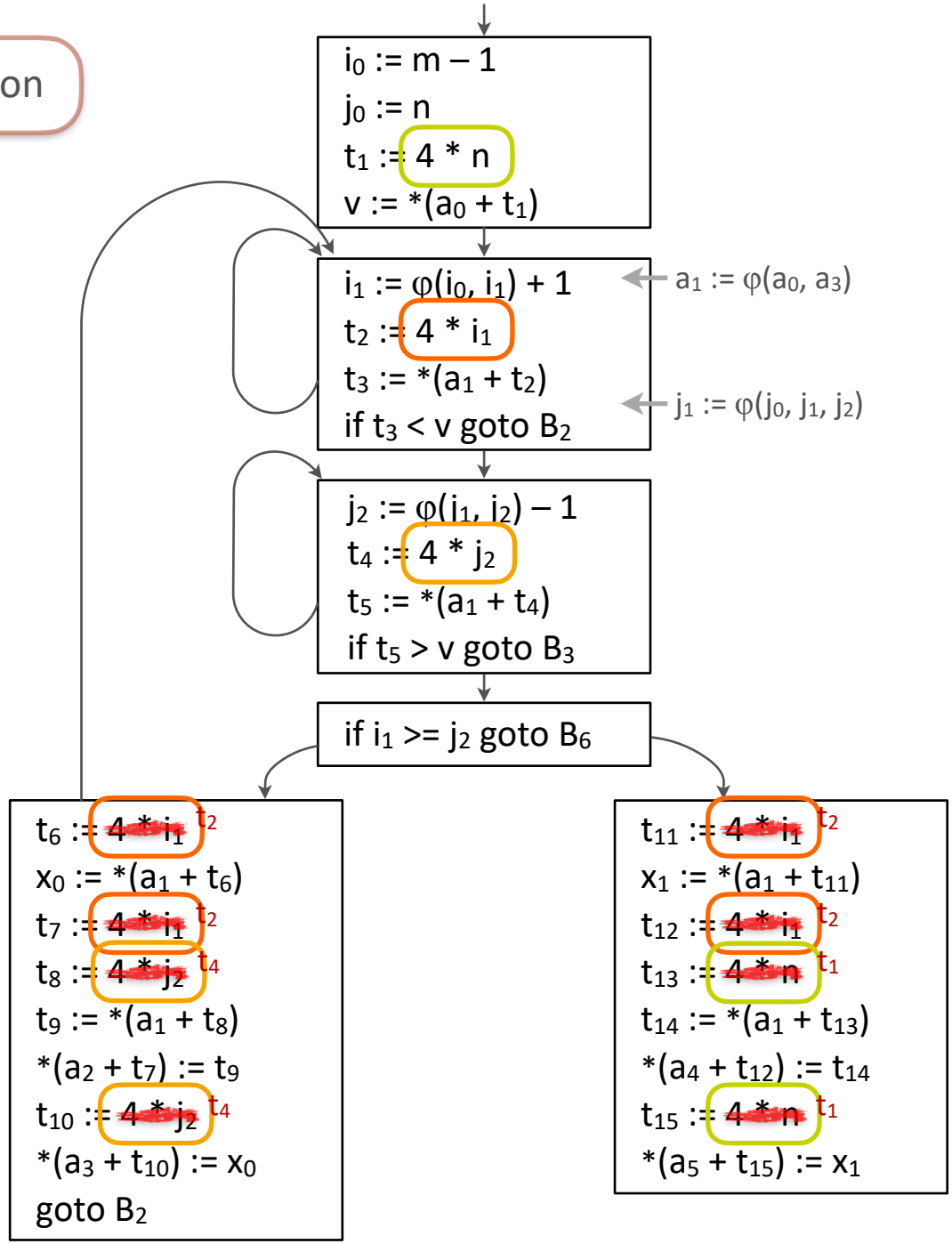
```
void quicksort ( m , n )
} ; int m , n
    int i , j ;
    int v , x ;
    if ( n <= m ) return ;
        code fragment
    quicksort ( m , j ) ; quicksort ( i + 1 , n ) ; }
```

```
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 ; }
x = a [ i ] ; a [ i ] = a [ n ] ; a [ n ] = x ;
```



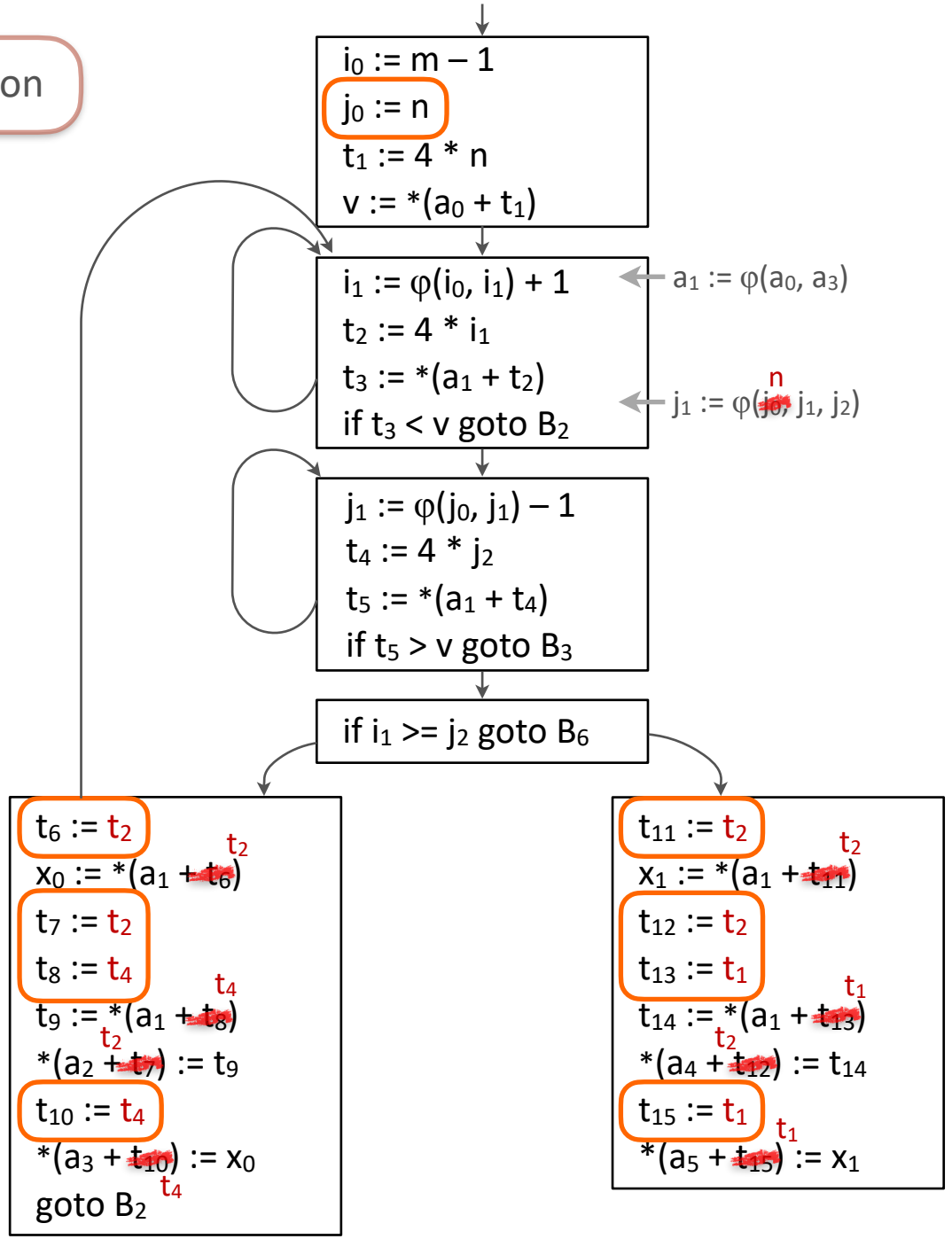

i := m – 1
j := n
t1 := 4 * n
v := *(a + t1)
i := i + 1
t2 := 4 * i
t3 := *(a + t2)
if t3 < v goto B2
j := j – 1
t4 := 4 * j
t5 := *(a + t4)
if t5 > v goto B3
if i >= j goto B6
t6 := 4 * i
x := *(a + t6)
t7 := 4 * i
t8 := 4 * j
t9 := *(a + t8)
*(a + t7) := t9
t10 := 4 * j
*(a + t10) := x
goto B2
t11 := 4 * i
x := *(a + t11)
t12 := 4 * i
t13 := 4 * n
t14 := *(a + t13)
*(a + t12) := t14
t15 := 4 * n
*(a + t15) := x

Common subexpression elimination



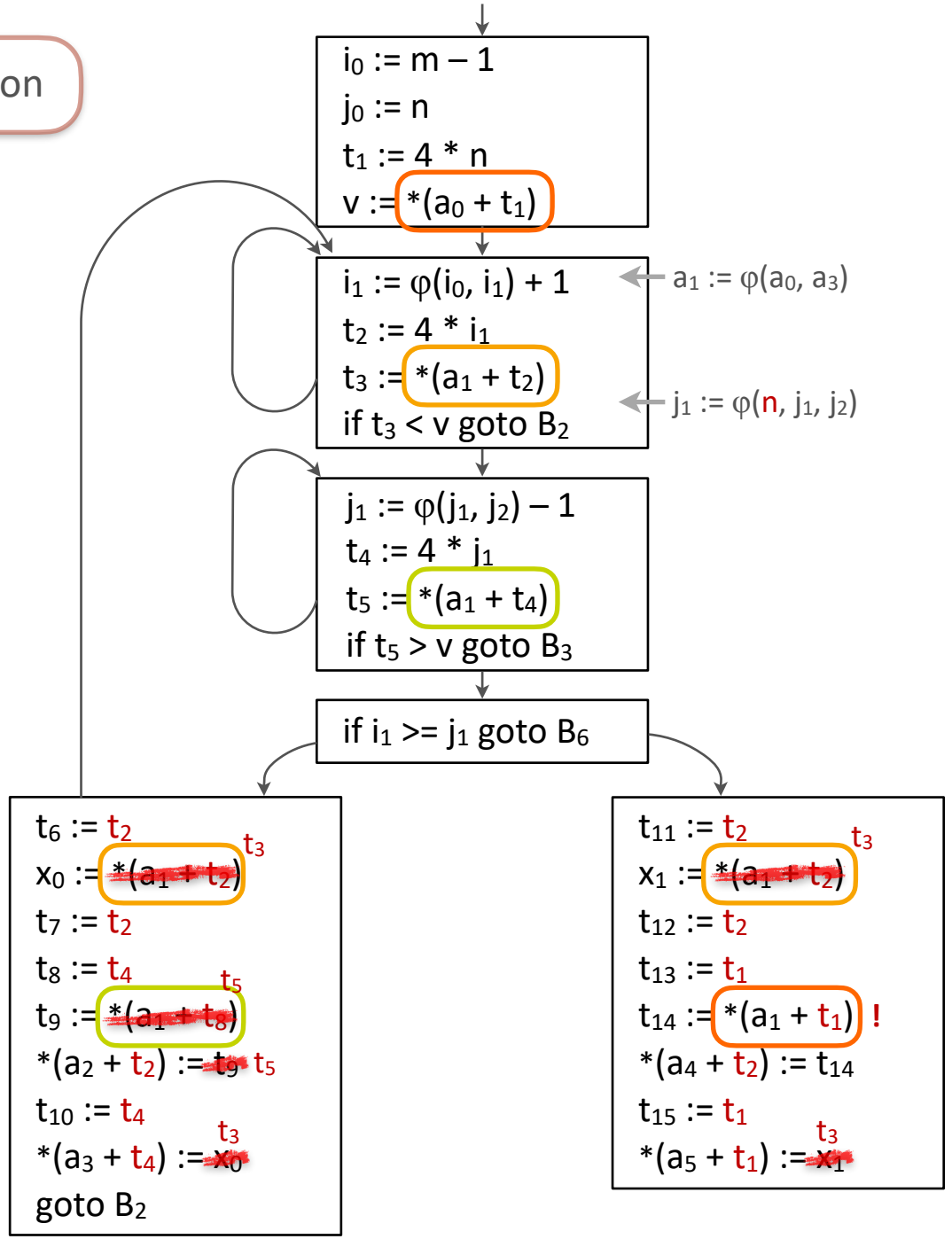
Common subexpression elimination

Copy propagation



Common subexpression elimination

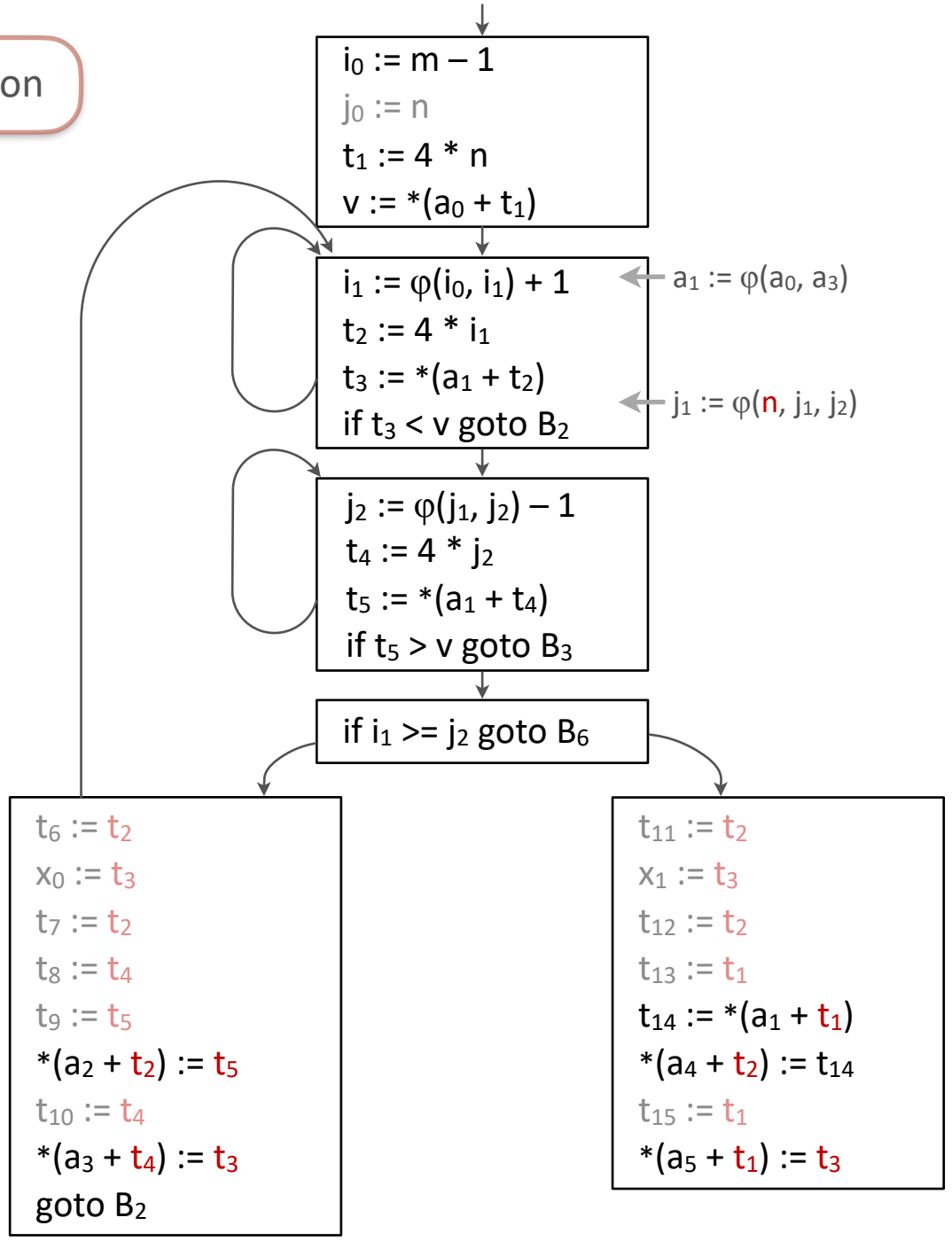
Copy propagation



Common subexpression elimination

Copy propagation

Dead code elimination

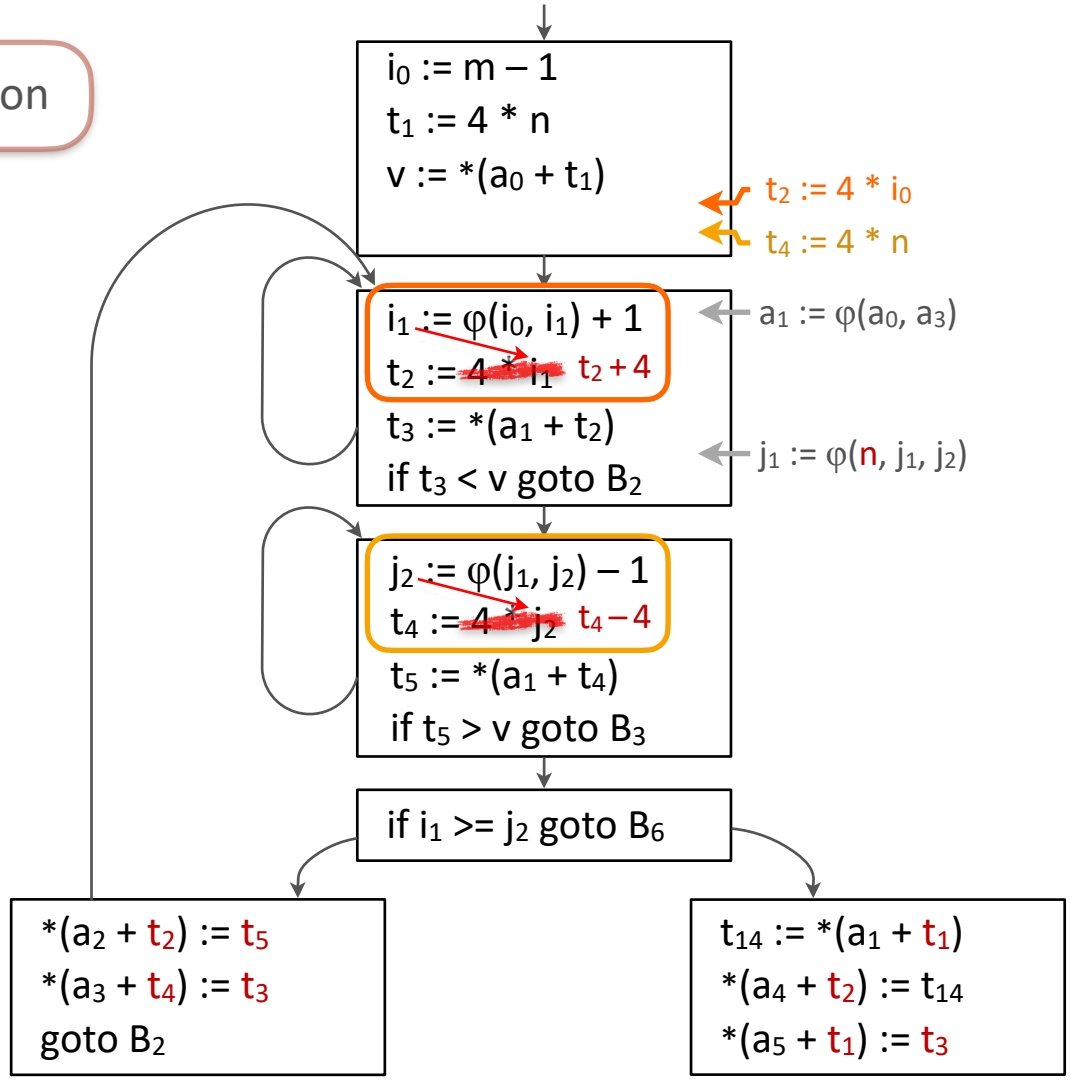


Common subexpression elimination

Copy propagation

Dead code elimination

Strength reduction

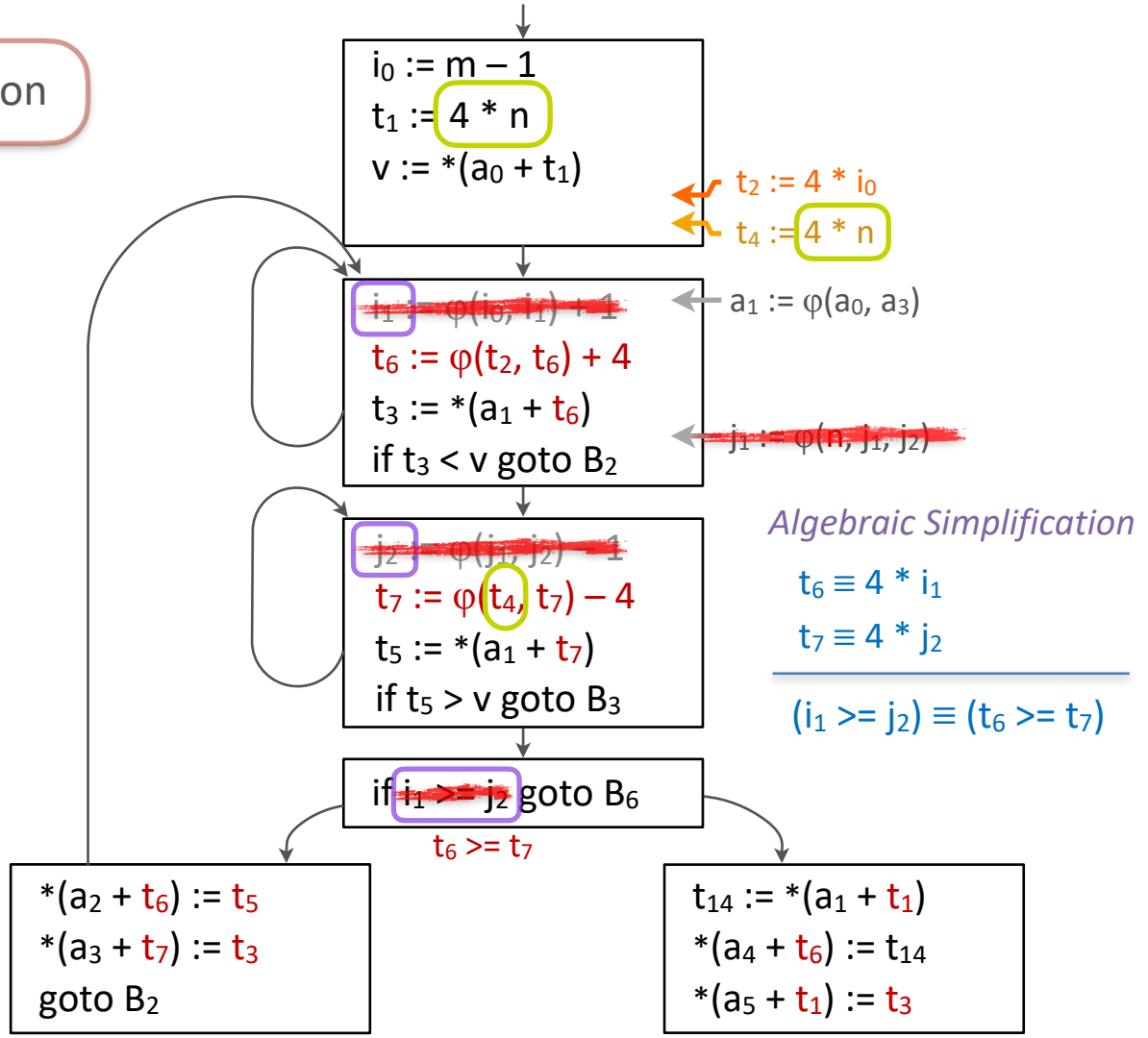


Common subexpression elimination

Copy propagation

Dead code elimination

Strength reduction



Algebraic Simplification

$t_6 \equiv 4 * i_1$

$t_7 \equiv 4 * j_2$

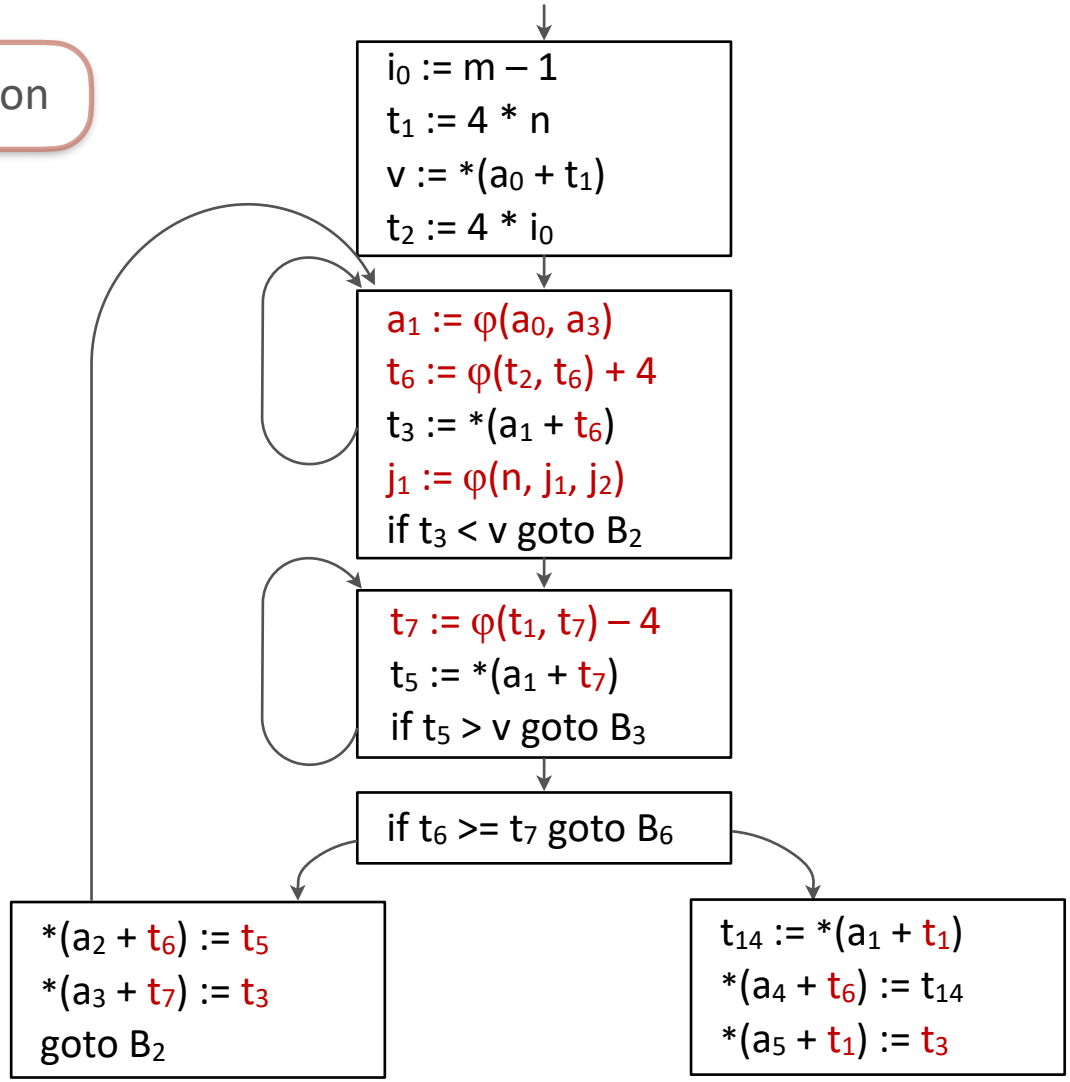
$(i_1 >= j_2) \equiv (t_6 >= t_7)$

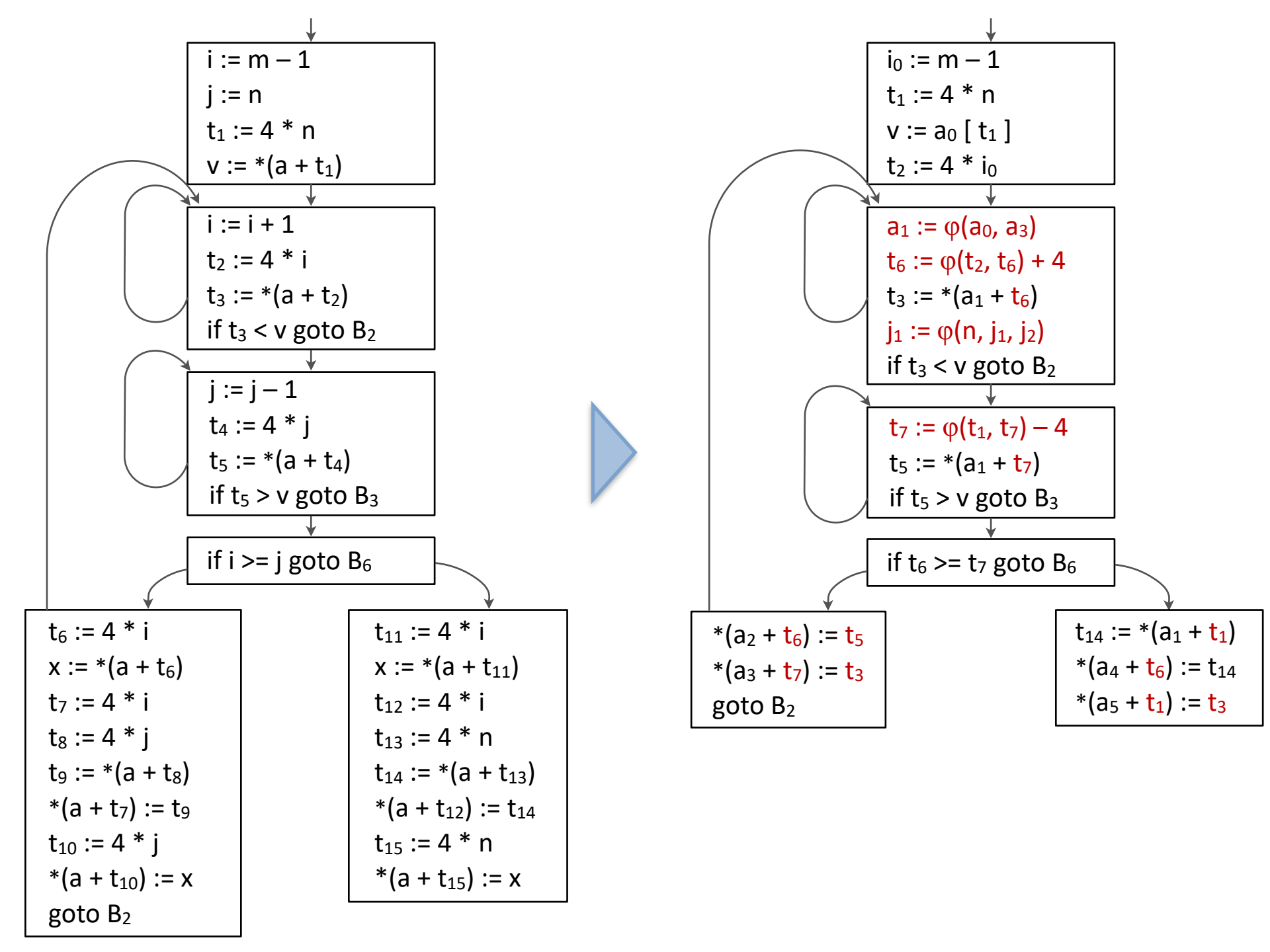
Common subexpression elimination

Copy propagation

Dead code elimination

Strength reduction




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
i0 := m – 1
t1 := 4 * n
v := a0 [t1]
t2 := 4 * i0
a1 := φ(a0, a3)
t6 := φ(t2, t6) + 4
t3 := *(a1 + t6)
j1 := φ(n, j1, j2)
if t3 < v goto B2
t7 := φ(t1, t7) – 4
t5 := *(a1 + t7)
if t5 > v goto B3
if t6 >= t7 goto B6
*(a2 + t6) := t5
*(a3 + t7) := t3
goto B2
t14 := *(a1 + t1)
*(a4 + t6) := t14
*(a5 + t1) := t3

Summary

- Optimizations
  - ✓ Not real “optimal” code
  - ✓ But code that is “better” in some respect
  - ✓ Often focuses on runtime
- Techniques
  - ✓ Local: expressions in basic block, peephole
  - ✓ Global: (function-level) Code motion, strength reduction — based on running a DFA first

Next