Code Optimization

Part II

Global Data Flow Analysis

Examples:

Reaching Definitions:

Which DEFINITIONs reach which USEs?

LIVE Variable Analysis:

Which variables are live at a given point, P?

Global Sub-Expression Elimination:

Which expressions reach point P
and do not need to be re-computed?

Copy Propagation:

Which copies reach point P?
Can we do copy propagation?

Terminology

A "point"

between two adjacent statements in a basic block,

or directly before the basic block,

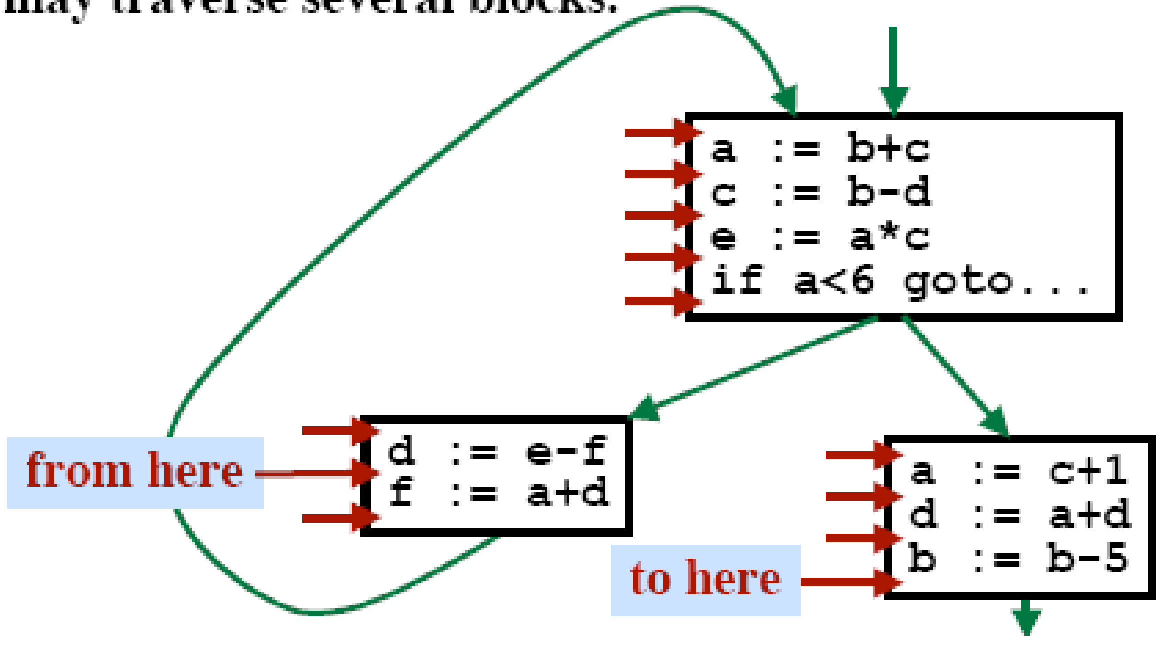
or directly after the basic block.

A "path"

is a sequence of points from P_1 to P_N such that...

control *could* flow from P_1 to P_N.

The path may traverse several blocks.



Reaching Definitions

A "definition" of variable x

A statement that assigns to x (or *might* assign to x).

Ambiguous Definitions -- Might assign
Unambiguous Definitions -- Will definitely assign

Examples

```
x := ...;
read (x);
call foo (... x ...)
call foo ()
*p := ...;
y := ...;
```

Unambiguous; will definitely change x

Where x is passed by reference, by copy-restore, or by name

Where the function may access X as a non-local

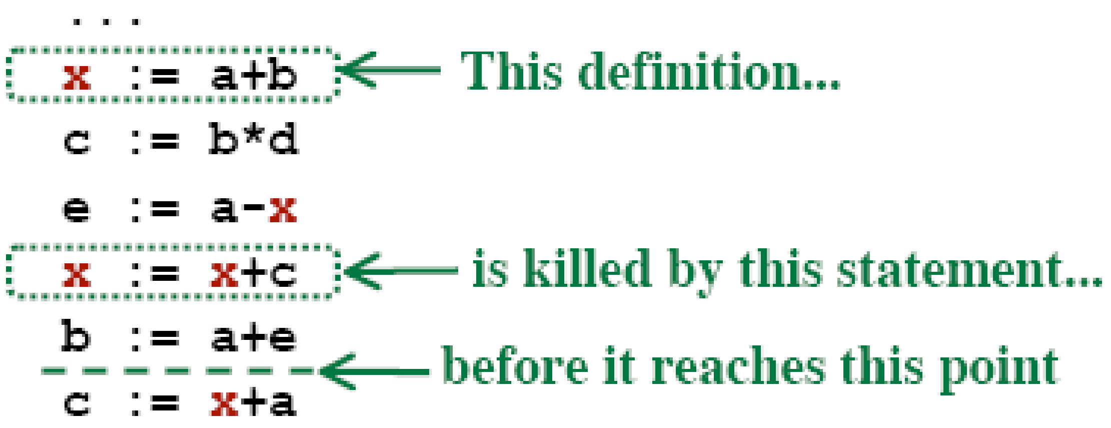
Pointer assignment

Aliasing

Killing Definitions

A definition is "killed" along a path...

if there is an unambiguous definition of the variable.



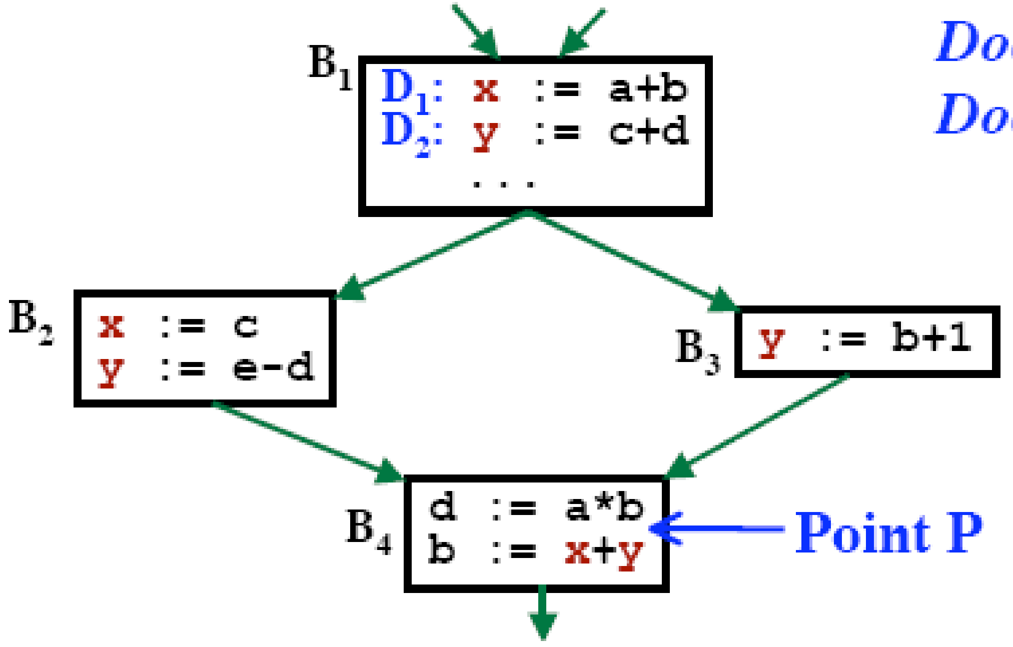
```
  ...
x := a+b
c := b*d
e := a-x
x := x+c
b := a+e
- - - - - - - -
c := x+a
```

Reach

A definition D "reaches" a point P...
if there is a path from D to P along which D is not killed.

If "x" is defined at D, then the value given to "x" *might* be the value of "x" at point P.

When D reaches P, it means...
The value of "x" *might* reach P at runtime.



Does D_1 reach P? YES

Does D_2 reach P? NO

Safe, Conservative Estimates

Will the value of x reach point P?

The runtime value of variables may cause some paths to NEVER be taken.

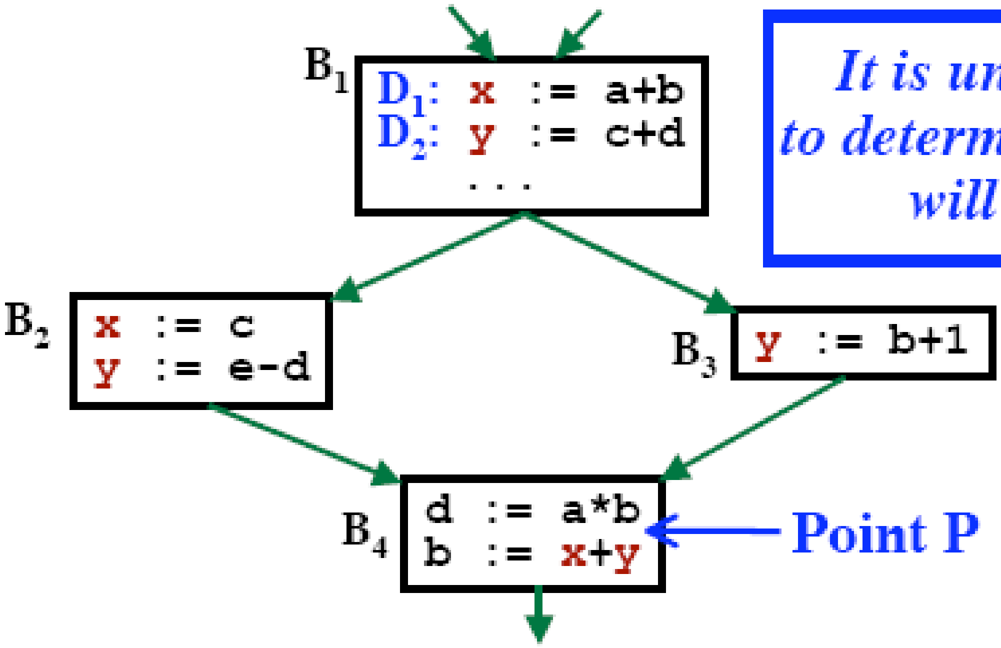
It may be the case that...

In ALL executions, control ALWAYS passes through B2...

D may get killed in every execution!

The value of "x" may never reach point P!

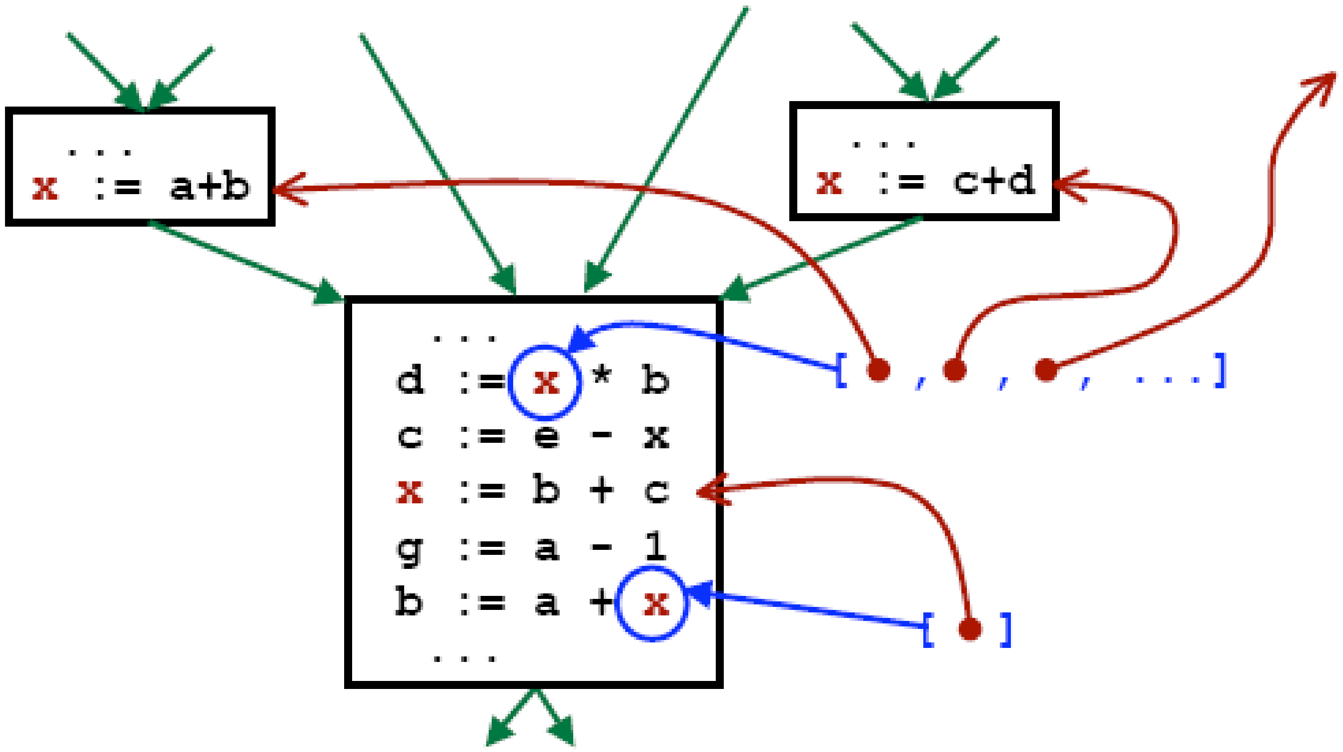
Nevertheless, we say "D reaches P".



It is undecideable (in general) to determine statically which paths will be taken at runtime.

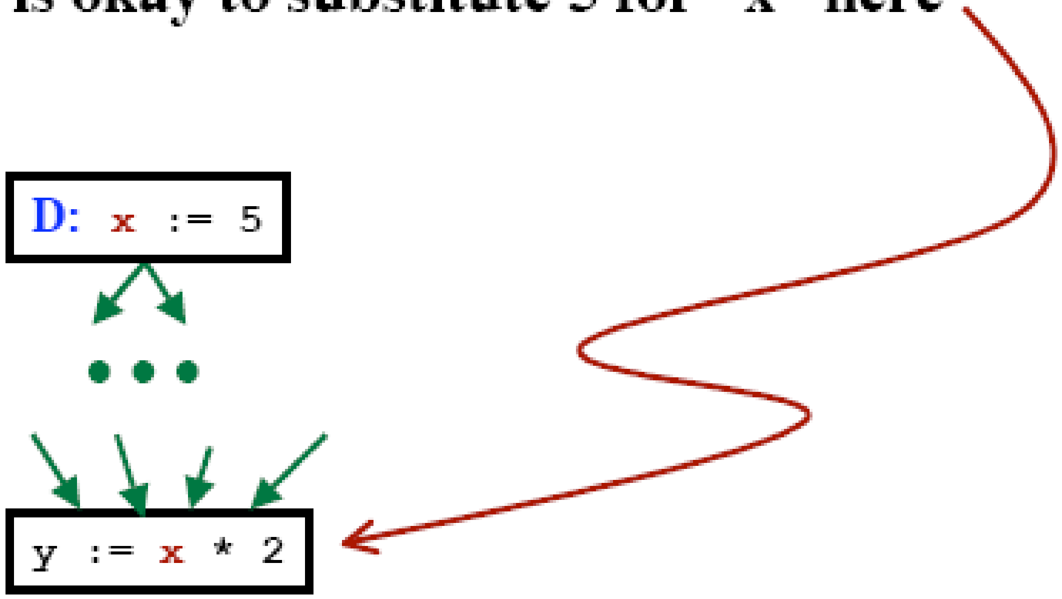
USE-DEFINITION Chains (U-D Chains)

For each USE of some variable "x"...
build a list of all the DEFINITIONs of "x"
that reach this USE.



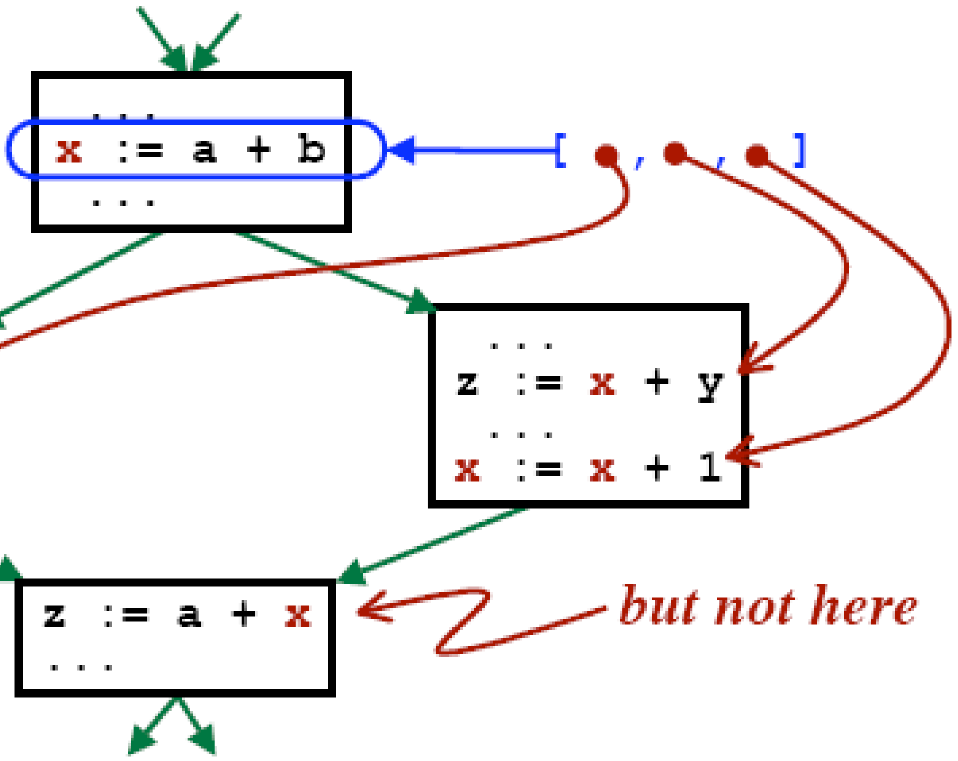
USE-DEFINITION Chains (U-D Chains)

If we can deduce that the set of definitions reaching this point contains *ONLY* the assignment D to "x", then it is okay to substitute 5 for "x" here



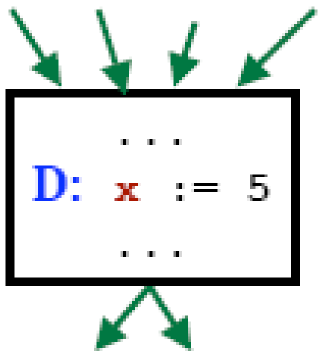
DEFINITION-USE Chains (D-U Chains)

A variable is USED at statement S if
its value *may be* required.
For each DEFINITION of a variable...
compute a list of all possible USEs of that variable.



DEFINITION-USE Chains (D-U Chains)

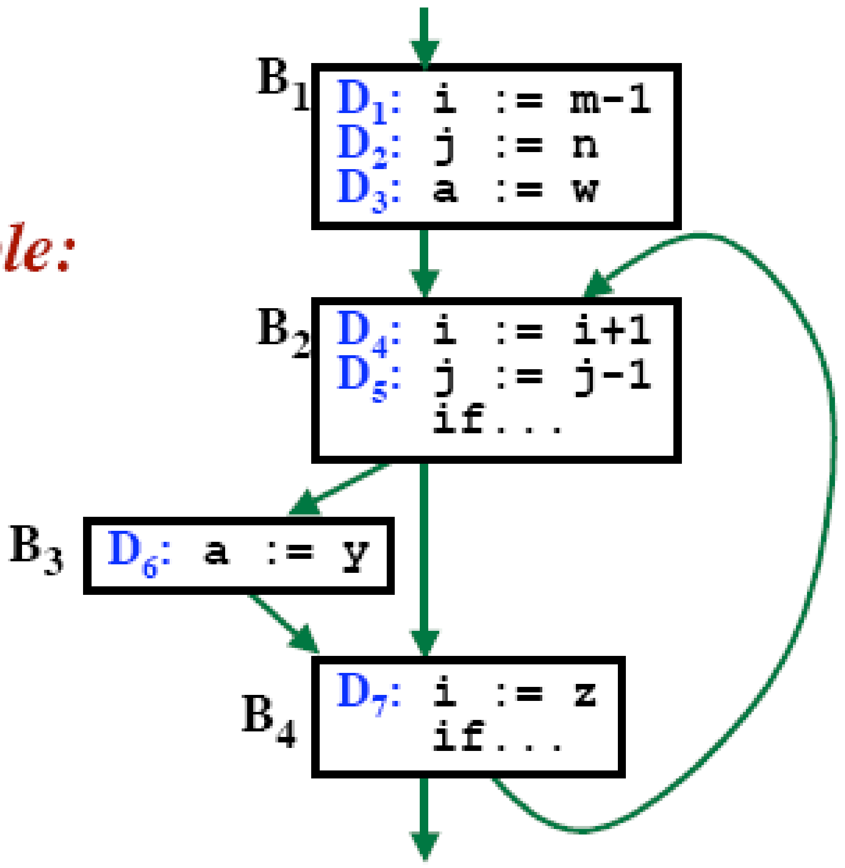
If we can deduce that the definition D has *NO POSSIBLE USES* then D is "DEAD" (useless code) and can be eliminated !



The Universe

$\mathbb{U}$ = Universe
= the set of all DEFINITIONs in the program / CFG
Number them D_1, D_2, D_3, ...

Example:



Representing Sets

We will work with sets.

How to represent?

Each set is represented with a Bit Vector

D_1	D_2	D_3	D_4	D_5	D_6	D_7

Example

$A = \{ D_2, D_4, D_7 \}$

$A' =$

0	1	0	1	0	0	1

How to compute set operations?

Set Union

$A \cup B \Rightarrow A'$ or B'

Set Intersection

$A \cap B \Rightarrow A'$ and B'

Set Difference

$A - B \Rightarrow A'$ and (not B')

Approach

Figure out what happens in each basic block...

GEN[B] =

In the text: `DEDef()`

- The set of definitions appearing in block B
  which reach the end of B
  (without being KILLed before the end of the block)

In the text: `DefKill()`

KILL[B] =

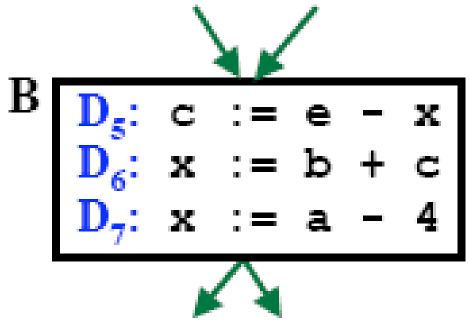
- The set of definitions KILLed by statements in block B.
- If B contains an unambiguous definition of variable "x",
  then add all definitions of "x" to KILL[B].
  (unless the definition D of "x" also occurs in B and
  there are no unambiguous definitions between D
  and the end of B).

Use this info to do the entire flow graph...

Using DATA FLOW EQUATIONS

Example of GEN [B]

Consider this Basic Block:



Consider D_5, a definition of “c”...

Add D_5 to GEN [B].

Consider D_6, a definition of “x”...

But this is KILLed before the end of the block.

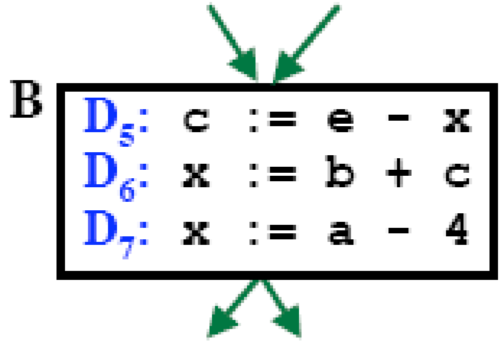
Consider D_7, a definition of “x”...

Add D_7 to GEN [B].

GEN [B] = $\{D_5, D_7\}$

Example of KILL [B]

Consider this Basic Block:



Consider D_5, an unambiguous defintion of “c”...
Add all other definitions of “c” to KILL [B].
(Except, do not add D_5 itself,
since this definition “makes it to the end of the block”.)

Consider D_7, an unambiguous defintion of “x”...
Add all other definitions of “x” to KILL [B]
(Except, do not add D_7 itself,
since this definition “makes it to the end of the block”.)

Overview of the Computation

For every point in the program...
we want to know which definitions can reach that point.

We will compute the set of definitions that can
reach the beginning of a basic block:

IN [B]

In the text: `Reaches()`

Then, using GEN [B] and KILL [B], we will compute the set of
definitions reaching the end of the basic block:

OUT [B]

Then we will use OUT [B] to compute the set of definitions
that can reach other basic blocks.

... And we will repeat, until we learn which
definitions could possibly reach which blocks.

The Data Flow Algorithm

Approach:

Build the IN and OUT sets simultaneously,
by successive approximations!

Given:

A control flow graph of basic blocks.

Assume:

GEN[B] and KILL[B] have already be computed
for each basic block.

Output:

IN[B] and OUT[B] for each basic block.

The Data Flow Algorithm

Start by setting IN[B] to {} for each basic block.

Then compute OUT[B] from the previous estimate of IN[B].

Finally, propagate OUT[B] to the IN[B’]
for all successor blocks to B.

Repeat, until no more changes.

As the definitions “flow through the graph”,
the IN and OUT sets grow and grow.

The approximation gets closer and closer.

Conservative: May overestimate
how far definitions will reach.

(i.e., the results may be larger than “truly” necessary.)

The Data Flow Algorithm

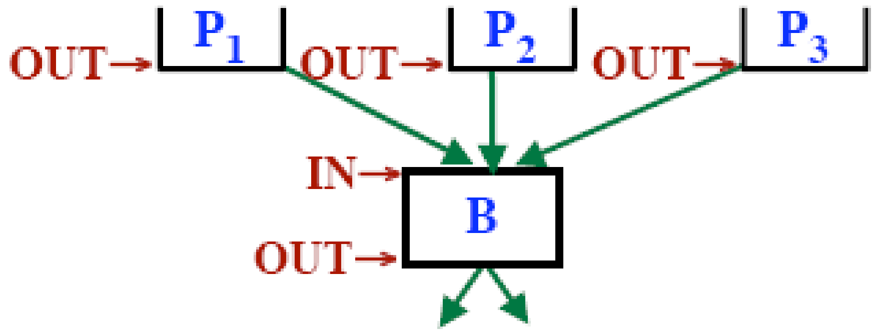
A Recurrence

(a set of simultaneous equations)

$$\sum_{0<i<N} f(i)$$

```
IN[B] := ⋃        OUT[P]
     P is a predecessor of B

OUT[B] := GEN[B] ∪ ( IN[B] - KILL[B])
```



The Data Flow Algorithm

$$IN[B] := \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] := GEN[B] \cup (IN[B] - KILL[B])$$

```
for each block B do
   OUT[B] := GEN[B]
endfor
```

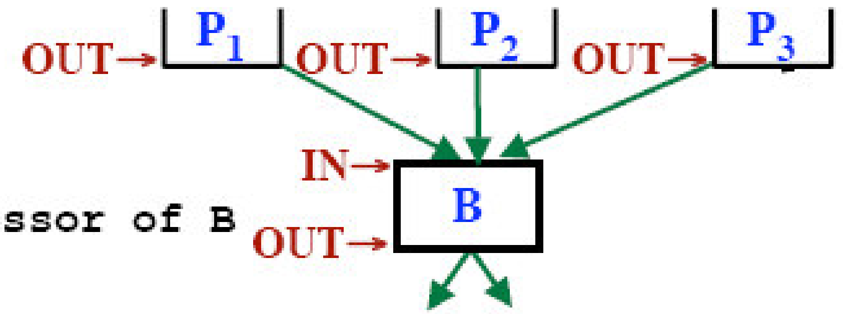
Initialize OUT on the assumption that IN[B] = {} *for all blocks.*

```
while change do

   for each block B do
      IN[B] := $\bigcup_{P \text{ is a predecessor of } B}$ OUT[P]

      OUT[B] := GEN[B] $\cup$ (IN[B] - KILL[B])

   endfor
endwhile
```



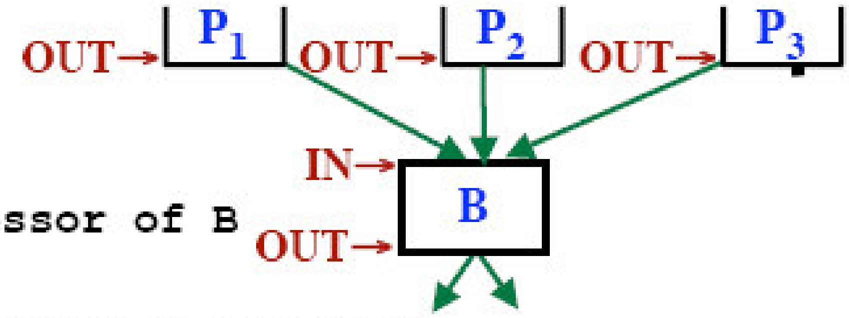
The Data Flow Algorithm

$$IN[B] := \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] := GEN[B] \cup (IN[B] - KILL[B])$$

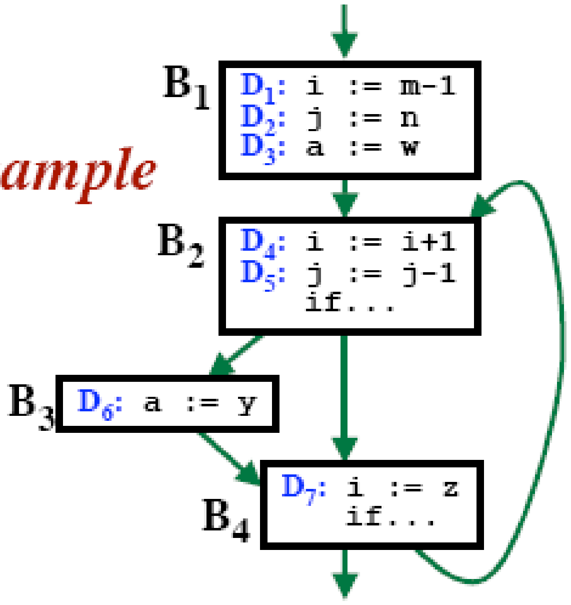
```
for each block B do
   OUT[B] := GEN[B]
endfor
change := true
while change do
   change := false
   for each block B do
      IN[B] := $\bigcup_{P \text{ is a predecessor of } B} OUT[P]$
      OLD_OUT := OUT[B]
      OUT[B] := GEN[B] ∪ (IN[B] - KILL[B])
      if OUT[B] ≠ OLD_OUT then
         change := true
      endif
   endfor
endwhile
```

Initialize OUT on the assumption that IN[B] = {} for all blocks.



The Data Flow Analysis: Example

Example



$GEN[B_1] = \{ D_1, D_2, D_3 \}$
111 0000

$KILL[B_1] = \{ D_4, D_5, D_6, D_7 \}$
000 1111

$GEN[B_2] = \{ D_4, D_5 \}$
000 1100

$KILL[B_2] = \{ D_1, D_2, D_7 \}$
110 0001

$GEN[B_3] = \{ D_6 \}$
000 0010

$KILL[B_3] = \{ D_3 \}$
001 0000

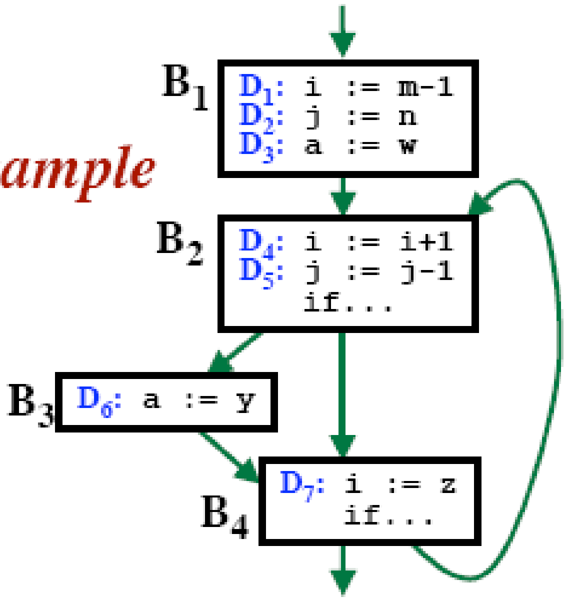
$GEN[B_4] = \{ D_7 \}$
000 0001

$KILL[B_4] = \{ D_1, D_4 \}$
100 1000

	B_1	B_2	B_3	B_4
OUT	111 0000	000 1100	000 0010	000 0001

The Data Flow Analysis: Example

Example



$GEN[B_1] = \{ D_1, D_2, D_3 \}$
111 0000
$KILL[B_1] = \{ D_4, D_5, D_6, D_7 \}$
000 1111

$GEN[B_2] = \{ D_4, D_5 \}$
000 1100
$KILL[B_2] = \{ D_1, D_2, D_7 \}$
110 0001

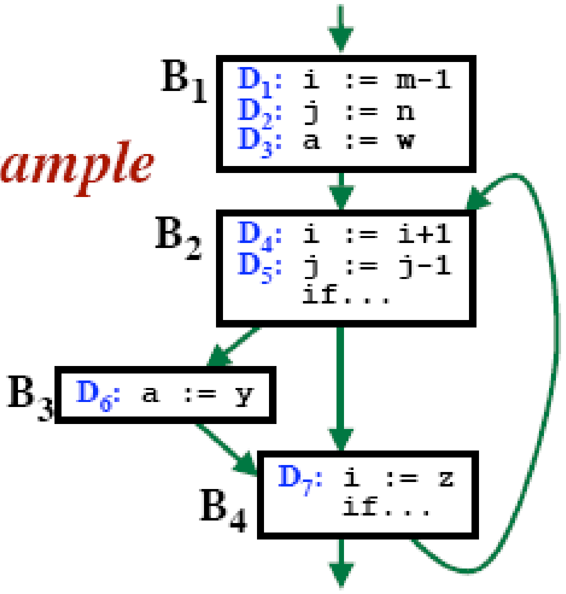
$GEN[B_3] = \{ D_6 \}$
000 0010
$KILL[B_3] = \{ D_3 \}$
001 0000

$GEN[B_4] = \{ D_7 \}$
000 0001
$KILL[B_4] = \{ D_1, D_4 \}$
100 1000

	B_1	B_2	B_3	B_4
OUT	111 0000	000 1100	000 0010	000 0001
IN	000 0000	111 0001	000 1100	000 1110

The Data Flow Analysis: Example

Example



GEN[B_1] = { D_1, D_2, D_3 }
111 0000
KILL[B_1] = { D_4, D_5, D_6, D_7 }
000 1111

GEN[B_2] = { D_4, D_5 }
000 1100
KILL[B_2] = { D_1, D_2, D_7 }
110 0001

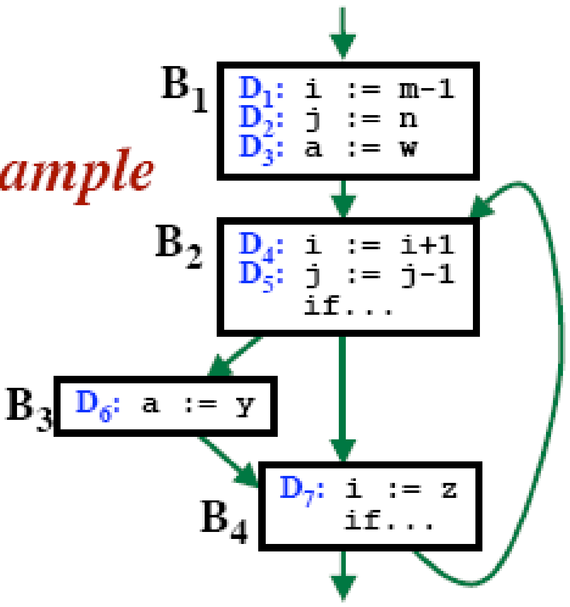
GEN[B_3] = { D_6 }
000 0010
KILL[B_3] = { D_3 }
001 0000

GEN[B_4] = { D_7 }
000 0001
KILL[B_4] = { D_1, D_4 }
100 1000

	B_1	B_2	B_3	B_4
OUT	111 0000	000 1100	000 0010	000 0001
IN	000 0000	111 0001	000 1100	000 1110
OUT	111 0000	001 1100	000 1110	000 0111

The Data Flow Analysis: Example

Example



```
B1: D1: i := m-1
    D2: j := n
    D3: a := w
B2: D4: i := i+1
    D5: j := j-1
        if...
B3: D6: a := y
B4: D7: i := z
        if...
```

GEN[B_1] = { D_1, D_2, D_3 }
111 0000
KILL[B_1] = { D_4, D_5 , D_6 , D_7 }
000 1111

GEN[B_2] = { D_4, D_5 }
000 1100
KILL[B_2] = { D_1, D_2, D_7 }
110 0001

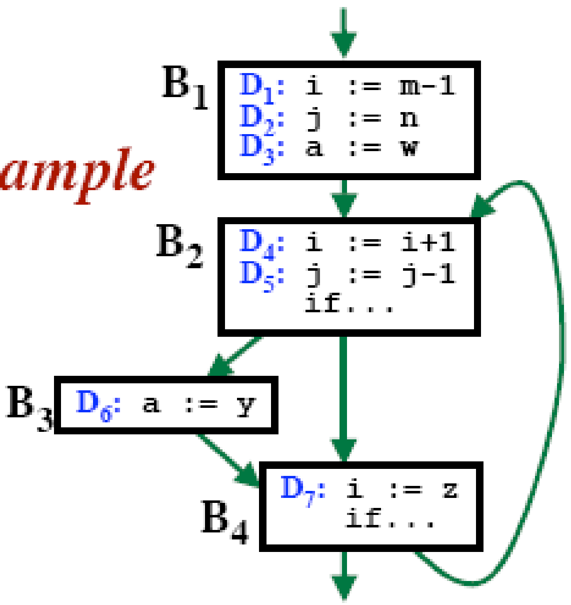
GEN[B_3] = { D_6 }
000 0010
KILL[B_3] = { D_3 }
001 0000

GEN[B_4] = { D_7 }
000 0001
KILL[B_4] = { D_1, D_4 }
100 1000

	B_1	B_2	B_3	B_4
OUT	111 0000	000 1100	000 0010	000 0001
IN	000 0000	111 0001	000 1100	000 1110
OUT	111 0000	001 1100	000 1110	000 0111
IN	000 0000	111 0111	001 1100	001 1110

The Data Flow Analysis: Example

Example



```
B1:
D1: i  := m-1
D2: j  := n
D3: a  := w

B2:
D4: i  := i+1
D5: j  := j-1
    if...

B3:
D6: a  := y

B4:
D7: i  := z
    if...
```

$GEN[B_1] = \{ D_1, D_2, D_3 \}$
111 0000
$KILL[B_1] = \{ D_4, D_5, D_6, D_7 \}$
000 1111

$GEN[B_2] = \{ D_4, D_5 \}$
000 1100
$KILL[B_2] = \{ D_1, D_2, D_7 \}$
110 0001

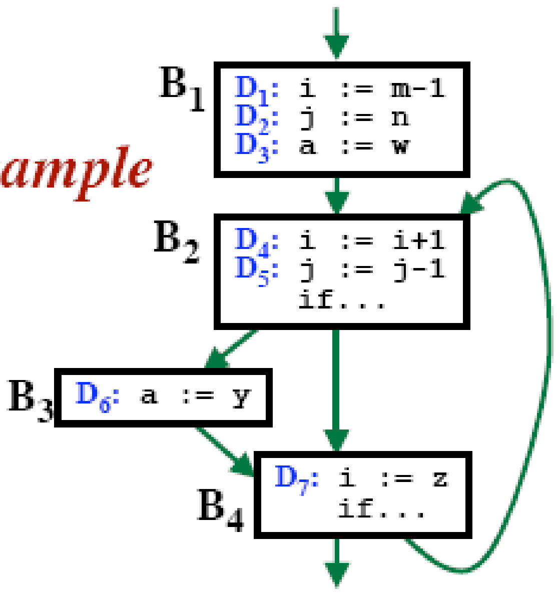
$GEN[B_3] = \{ D_6 \}$
000 0010
$KILL[B_3] = \{ D_3 \}$
001 0000

$GEN[B_4] = \{ D_7 \}$
000 0001
$KILL[B_4] = \{ D_1, D_4 \}$
100 1000

	B_1	B_2	B_3	B_4
OUT	111 0000	000 1100	000 0010	000 0001
IN	000 0000	111 0001	000 1100	000 1110
OUT	111 0000	001 1100	000 1110	000 0111
IN	000 0000	111 0111	001 1100	001 1110
OUT	111 0000	001 1110	000 1110	001 0111

The Data Flow Analysis: Example

Example



$GEN[B_1] = \{ D_1, D_2, D_3 \}$
111 0000
$KILL[B_1] = \{ D_4, D_5, D_6, D_7 \}$
000 1111

$GEN[B_2] = \{ D_4, D_5 \}$
000 1100
$KILL[B_2] = \{ D_1, D_2, D_7 \}$
110 0001

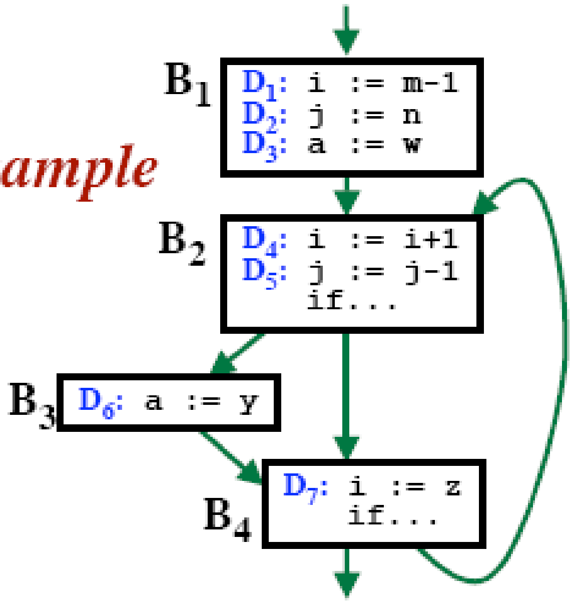
$GEN[B_3] = \{ D_6 \}$
000 0010
$KILL[B_3] = \{ D_3 \}$
001 0000

$GEN[B_4] = \{ D_7 \}$
000 0001
$KILL[B_4] = \{ D_1, D_4 \}$
100 1000

	B_1	B_2	B_3	B_4
OUT	111 0000	000 1100	000 0010	000 0001
IN	000 0000	111 0001	000 1100	000 1110
OUT	111 0000	001 1100	000 1110	000 0111
IN	000 0000	111 0111	001 1100	001 1110
OUT	111 0000	001 1110	000 1110	001 0111
IN	000 0000	111 0111	001 1110	001 1110

The Data Flow Analysis: Example

Example



$GEN[B_1] = \{ D_1, D_2, D_3 \}$
111 0000
$KILL[B_1] = \{ D_4, D_5, D_6, D_7 \}$
000 1111

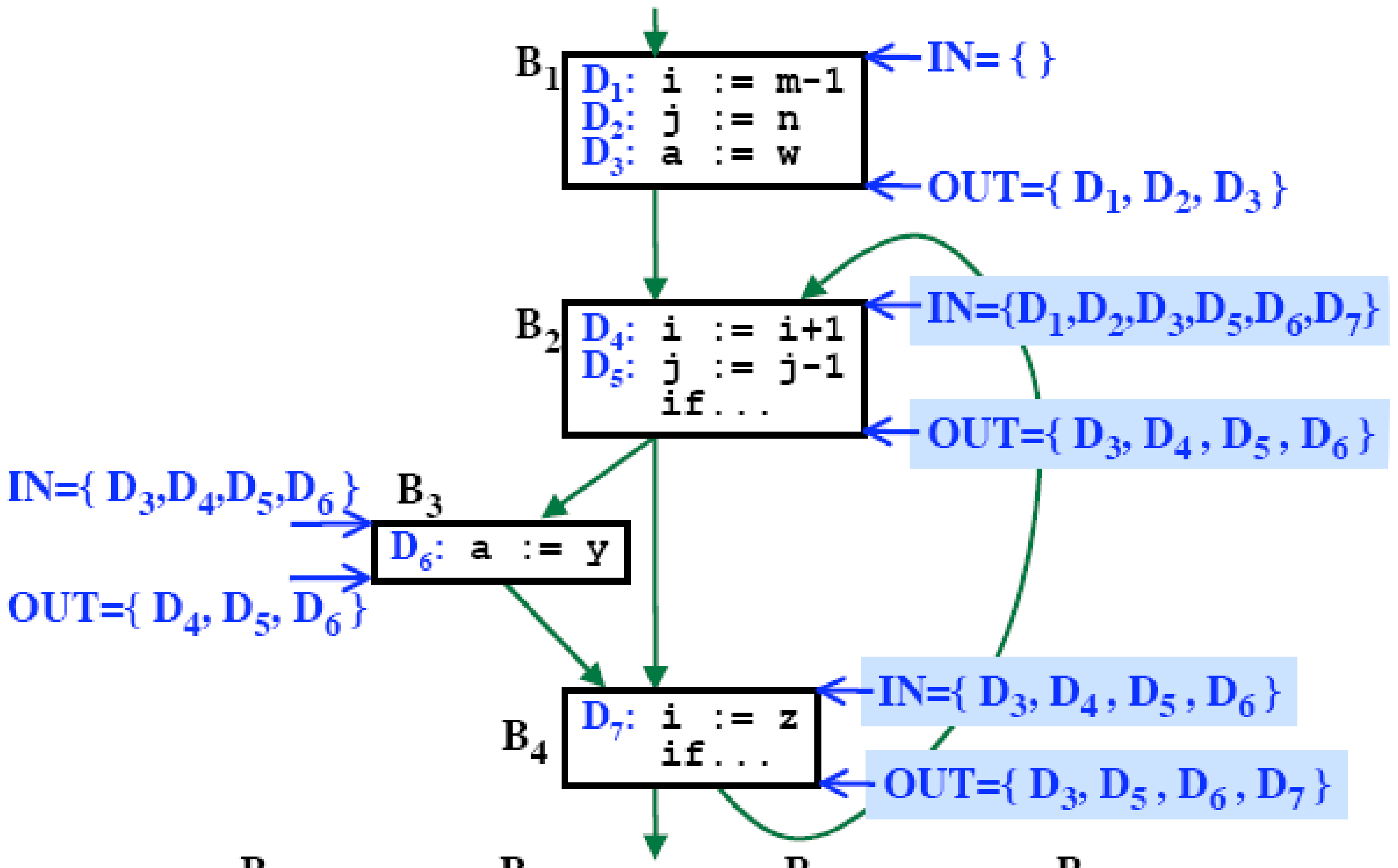
$GEN[B_2] = \{ D_4, D_5 \}$
000 1100
$KILL[B_2] = \{ D_1, D_2, D_7 \}$
110 0001

$GEN[B_3] = \{ D_6 \}$
000 0010
$KILL[B_3] = \{ D_3 \}$
001 0000

$GEN[B_4] = \{ D_7 \}$
000 0001
$KILL[B_4] = \{ D_1, D_4 \}$
100 1000

	B_1	B_2	B_3	B_4
OUT	111 0000	000 1100	000 0010	000 0001
IN	000 0000	111 0001	000 1100	000 1110
OUT	111 0000	001 1100	000 1110	000 0111
IN	000 0000	111 0111	001 1100	001 1110
OUT	111 0000	001 1110	000 1110	001 0111
IN	000 0000	111 0111	001 1110	001 1110
OUT	111 0000	001 1110	000 1110	001 0111

The Data Flow Analysis: Example



B_1
D_1: i := m-1
D_2: j := n
D_3: a := w
IN={ }
OUT={ D_1, D_2, D_3 }
B_2
D_4: i := i+1
D_5: j := j-1
if...
IN={D_1,D_2,D_3,D_5,D_6,D_7}
OUT={ D_3, D_4 , D_5 , D_6 }
IN={ D_3,D_4,D_5,D_6 }
B_3
D_6: a := y
OUT={ D_4, D_5, D_6 }
B_4
D_7: i := z
if...
IN={ D_3, D_4 , D_5 , D_6 }
OUT={ D_3, D_5 , D_6 , D_7 }

	B_1	B_2	B_3	B_4
IN	000 0000	111 0111	001 1110	001 1110
OUT	111 0000	001 1110	000 1110	001 0111

The Data Flow Analysis

This algorithm converges.

- OUT[B] never decreases...
  - Once in OUT[B] a definition stays there.
  - Eventually, no changes will be made to OUT[B].

An upper bound on the "while" loop?

- Number of nodes in the flow graph.
- Each iteration propagates reaching definitions.

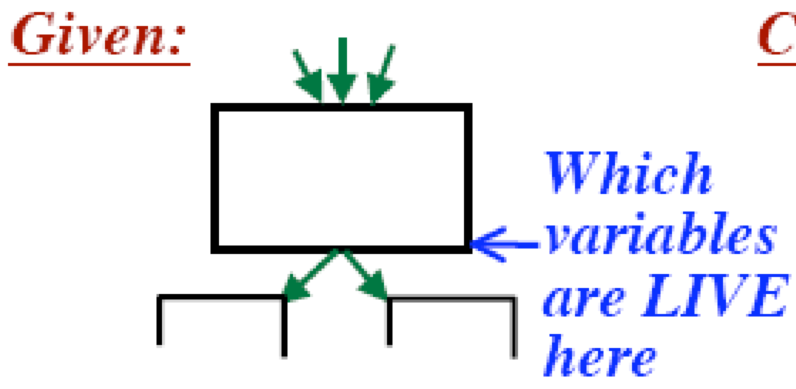
The "while" loop will converge quickly

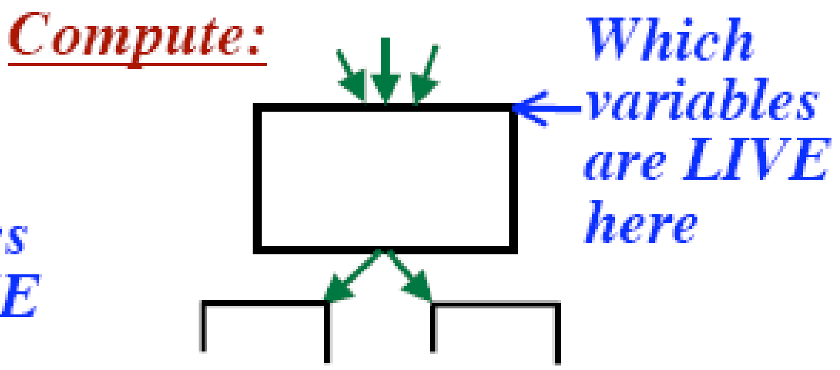
- ...if you select a good order for the nodes in the "for" loop.

This algorithm is efficient in practice.

Live Variable Analysis

A similar Data Flow Algorithm
Goal: Compute IN[] and OUT[]
However, it will work backwards!
(i.e., data will flow "upwards", against the arrow directions)



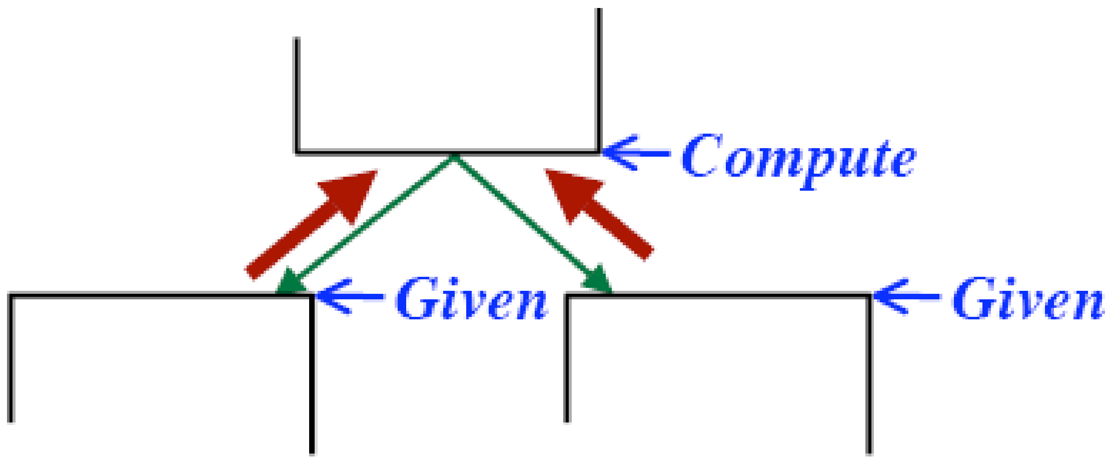


Live Variable Analysis

Then:

Compute the OUT set from

all the IN sets of the block's successors!



Info flows upwards !

"against" the flow graph edges

Definitions

Variable "x" is LIVE at some point P
if its value *might be* used at some point later,
on a path starting at P.

DEF [B] = the set of variables definitely assigned values in block B (prior to any use in B)

USE [B] = the set of variables whose values may be used in B (prior to any definitions of the variable)

IN [B] = the set of variables LIVE at the beginning of B

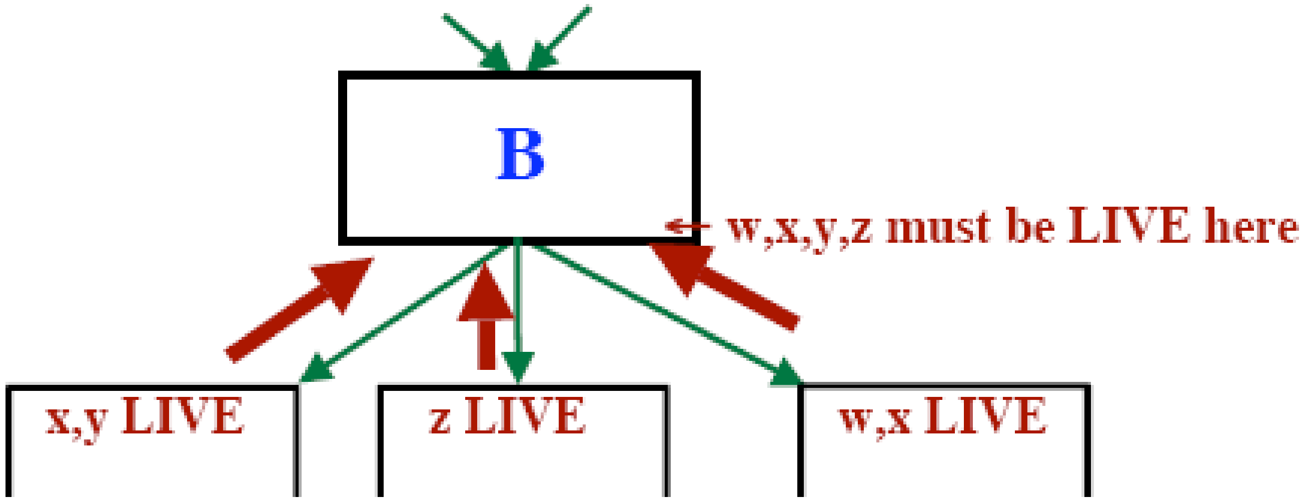
OUT [B] = the set of variables LIVE at the end of B

Note these re-definitions

Recurrence Equations to be Solved

$$IN[B] := USE[B] \cup (OUT[B] - DEF[B])$$

$$OUT[B] := \bigcup_{S \text{ is a successor of } B} IN[S]$$



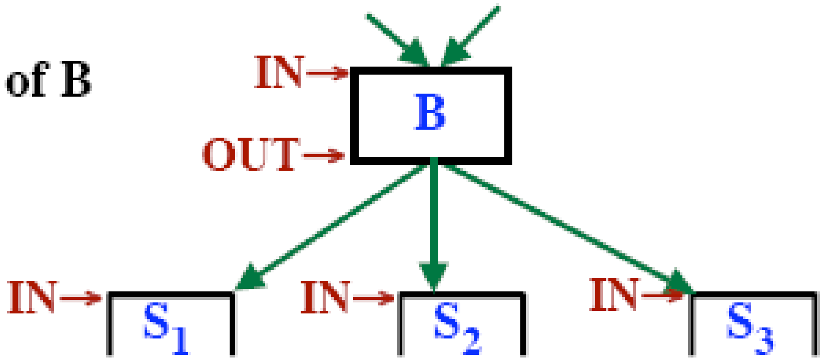
Algorithm to Compute LIVE Variables

Input:

Flow graph of basic blocks
DEF and USE for each block

Output:

OUT[B] = Live variables at end of B

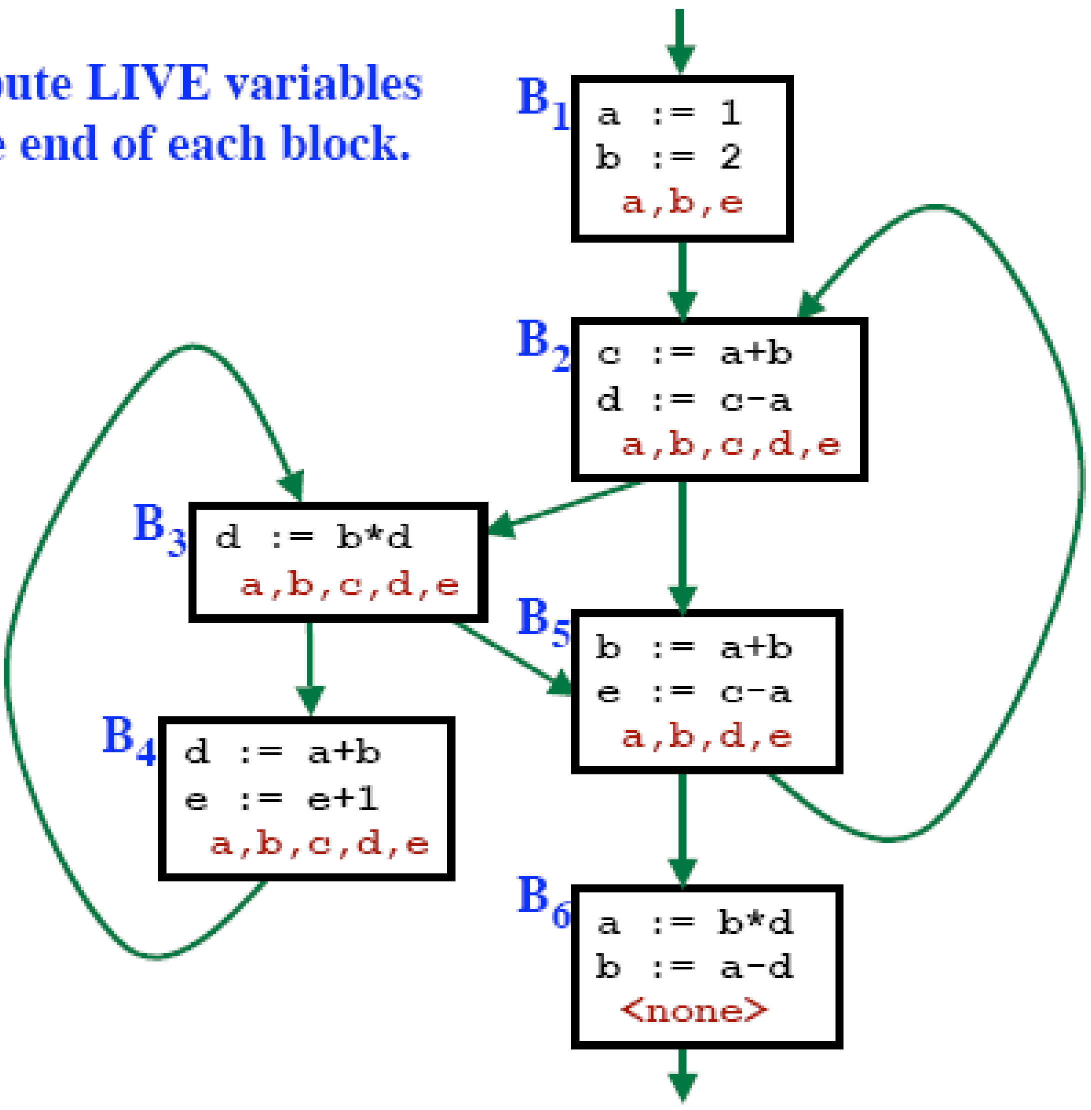


Algorithm:

```
for each block B do
   IN[B] := {}
endfor
while changes occur for any IN set do
   for each block B do
      OUT[B] := $\bigcup_{S \text{ is a successor of } B}$ IN[S]
      IN[B] := USE[B] $\cup$ (OUT[B] - DEF[B])
   endfor
endwhile
```

Algorithm to Compute LIVE Variables

Compute LIVE variables
At the end of each block.



Computing Available Expressions

An "expression":

$x \oplus y$

Binary expressions only

Any operator: +, –, *, ...

Examples: a–b, w+x, y*4, ...

An expression is "available" at point P if every path to P computes it
and there are no subsequent assignments to x or y
(between the last evaluation of $x \oplus y$ and P)

A block "generates" $x \oplus y$ if it evaluates $x \oplus y$
and does not subsequently assign to x or y.

A block "kills" $x \oplus y$ if it assigns to x or y
without subsequently recomputing $x \oplus y$.

Example

Which expressions are available?

```
x := y + z
y := x - w
a := w + z
z := x - w
y := y + z
```

Example

Which expressions are available?

$\mathbb{U}$ = { y+z , x-w , w+z }

```
                ←  Avail = { }
x  := y + z
                ←  Avail = { y+z }
y  := x - w
                ←  Avail = { x-w }
a  := w + z
                ←  Avail = { x-w, w+z }
z  := x - w
                ←  Avail = { x-w }
y  := y + z
                ←  Avail = { x-w }
```

Computing Available Expressions

The Universe

= The set of all expressions appearing in the flow graph

Example: $\mathbb{U}$ = { a–b, w+x , y*4 , x+1 , b–c }

E_GEN [B] =

The set of expressions computed in the block

x ⊕ y is included if some statement in B evaluates it

and the block does not assign to x or y after that.

E_KILL [B] =

The set of expressions that are invalidated because

the block contains an assignment to a variable they use.

E_IN [B] =

The set of expressions available at the beginning of block B.

E_OUT [B] =

The set of expressions available at the end of block B.

Recurrence Equations to be Solved

$$\texttt{E\_OUT[B]} := \texttt{E\_GEN[B]} \cup (\texttt{E\_IN[B]} - \texttt{E\_KILL[B]})$$

$$\left.\texttt{E\_IN[B]} := \bigcap_{\texttt{P is a predecessor of B}} \texttt{E\_OUT[P]}\right\} \textit{For } B \neq B1 \textit{ (the initial block)}$$

$$\texttt{E\_IN[}B_1\texttt{]} = \{\} \quad \textit{Nothing available before the initial block}$$

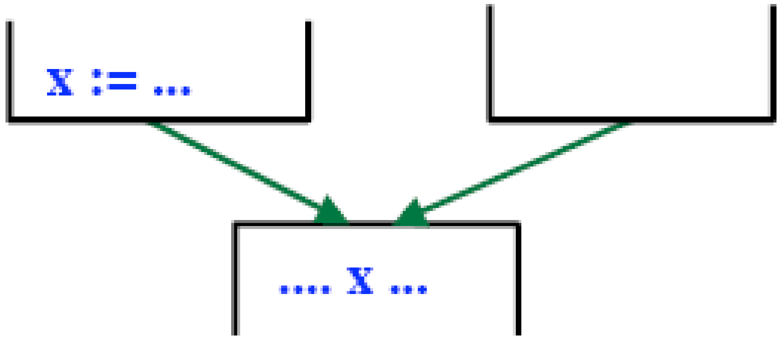
Forward Propagation

(like reaching definitions, but ∩ instead of ∪)

Reaching Definitions

Start with estimates that are too small, and enlarge them.

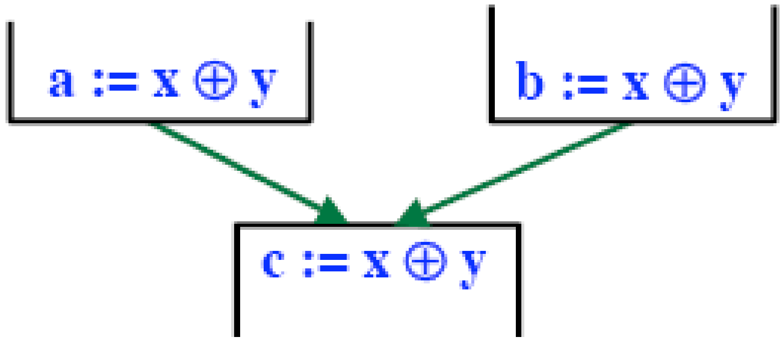
$$\texttt{IN[B]} = \bigcup_{p=\text{predecessor}} \texttt{OUT[P]}$$



Available Expressions

Start with estimates that are too large, and shrink them.

$$\texttt{E\_IN[B]} = \bigcap_{p=\text{predecessor}} \texttt{E\_OUT[P]}$$



Algorithm to Compute Available Expressions

Input:

E_GEN and E_KILL for each block

Output:

E_IN[B] = Expressions available at begining of B

Algorithm:

```
E_IN[$B_1$] := {}
E_OUT[$B_1$] := E_GEN[$B_1$]
for each block B except $B_1$ do
  E_OUT[B] := $U$ - E_KILL[B]
endfor
while changes occur for any E_OUT set do
  for each block B except $B_1$ do
    E_IN[B] := $\bigcap_{P \text{ is a predecessor of } B}$ E_OUT[P]
    E_OUT[B] := E_GEN[B] $\cup$ ( E_IN[B] - E_KILL[B] )
  endfor
endwhile
```

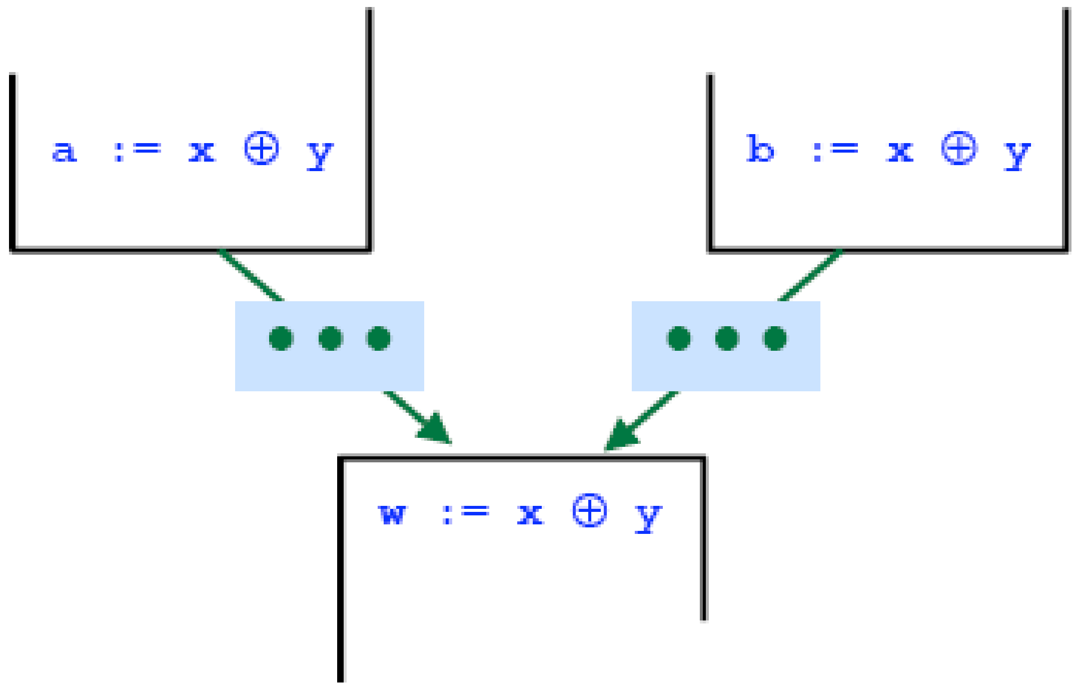
Conservative, Safe Estimates

- **Begin by assuming all expressions available anywhere.**
- **Work toward a smaller solution.**
- **If there is a *possible* definition of x or y then consider $x \oplus y$ as not available.**
- **We will tend to err by eliminating too many expressions from E_IN and E_OUT.**
- **Our computed result will be a subset of the expressions that are truly available at point P.**
- **If our computation determines that $x \oplus y$ is available at point P, then it surely is.**

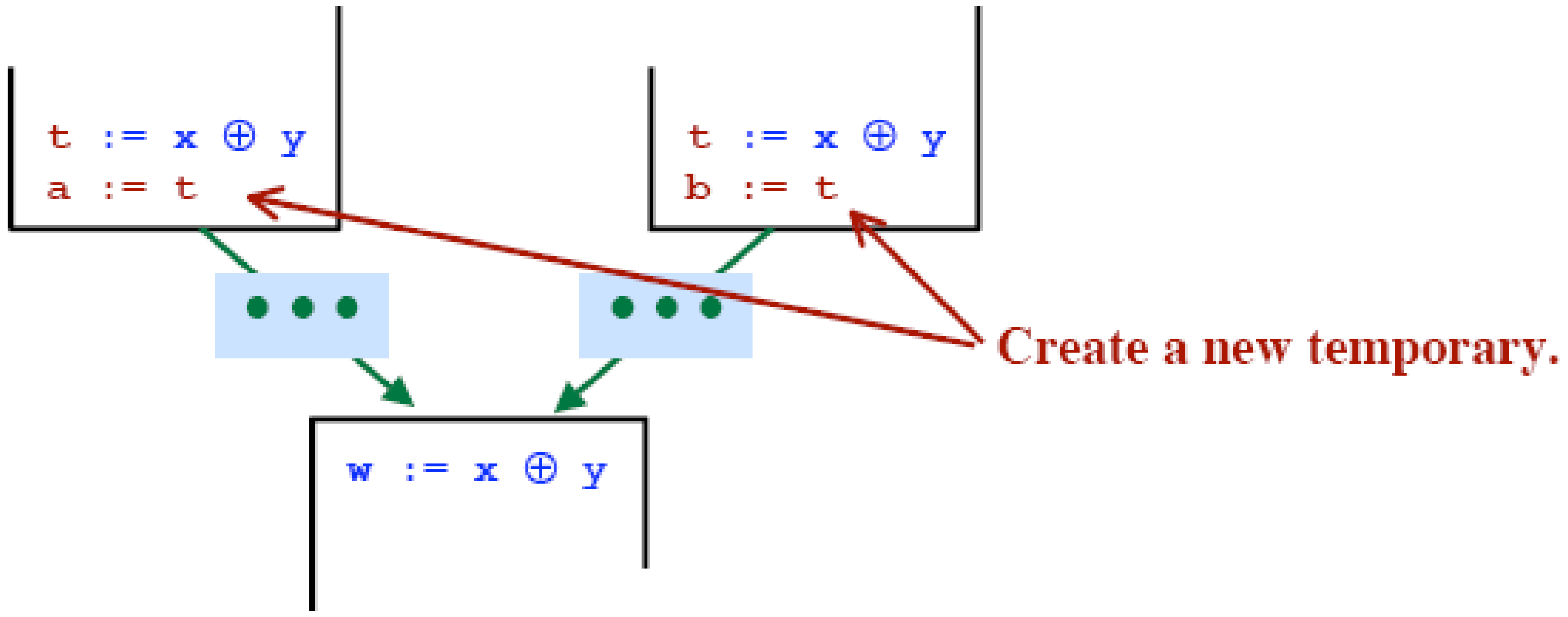
We can eliminate its recomputation!

Eliminating Common Global Subexpressions

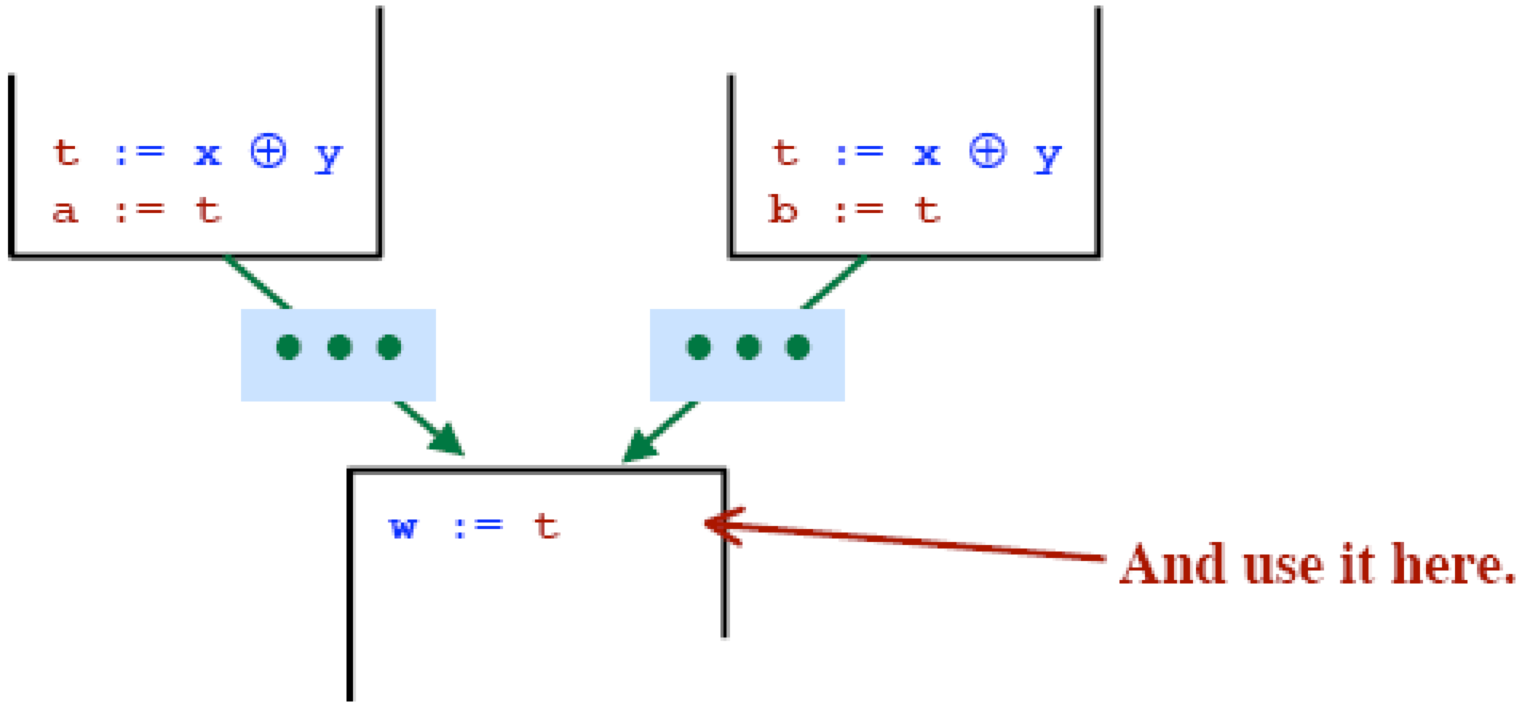
The Transformation



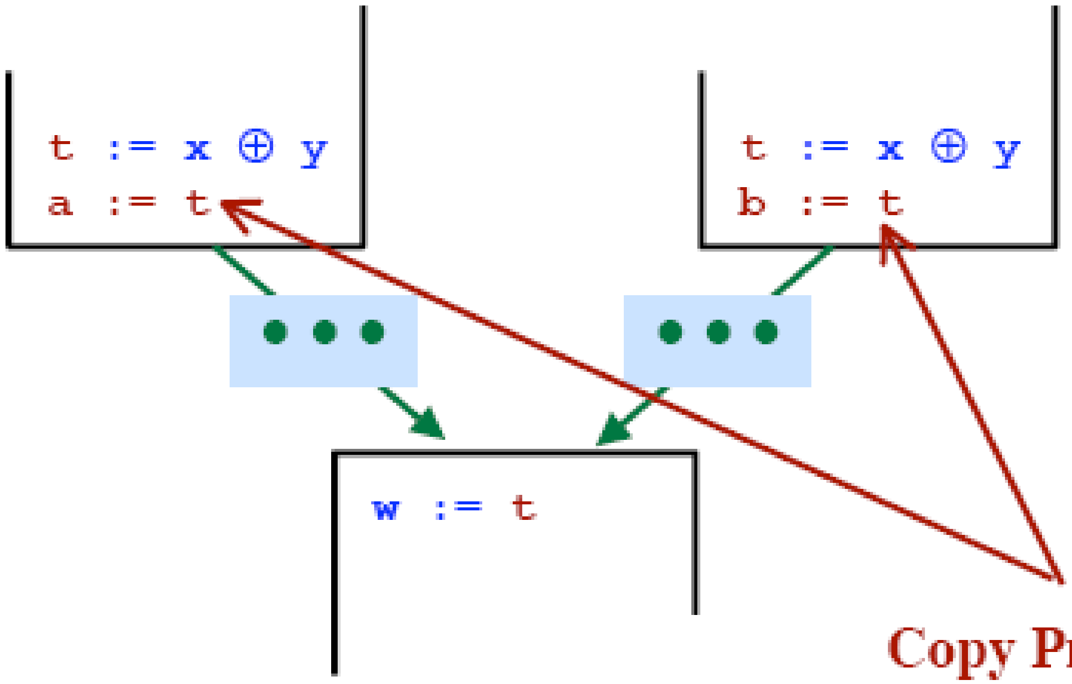
Eliminating Common Global Subexpressions



Eliminating Common Global Subexpressions



Eliminating Common Global Subexpressions



Copy Propagation may eliminate these statements.

Algorithm

Input: Flow Graph, Available Expression Information
Output: Revised Flow Graph

Step 1:

Find a statement such as

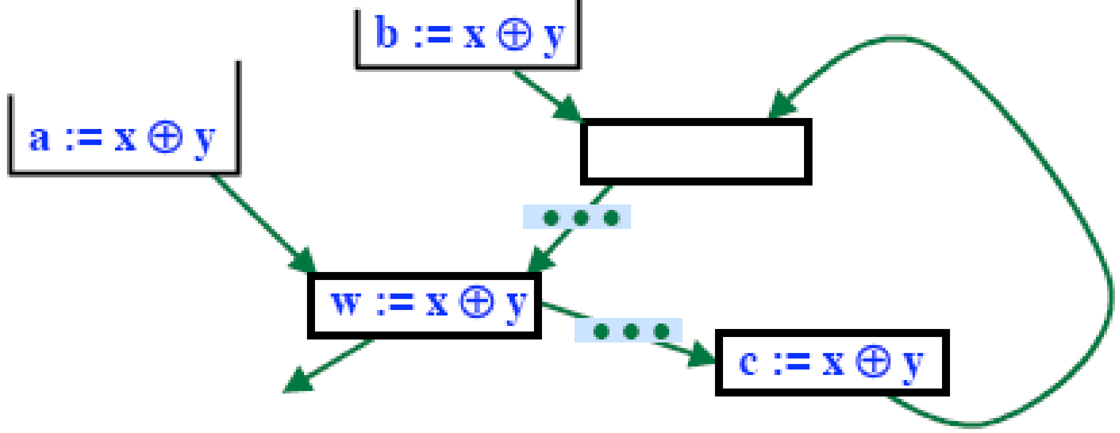
$w := x \oplus y$

such that expression $x \oplus y$ is available directly before it.

[Or: $x \oplus y$ is available in E_IN[B] for the block and there are no assignments to x or y before this statement.]

Step 2:

Follow the flow graph edges backward until you hit an evaluation of $x \oplus y$. Find all such evaluations.



Algorithm

Step 3:

Create a new temporary (say "t")

Step 4:

Replace all statements found in step 2.

$$\begin{tikzcd}
a := x \oplus y \arrow[d] & b := x \oplus y \arrow[d] & c := x \oplus y \arrow[d] \\
\begin{array}{l} t := x \oplus y \\ a := t \end{array} & \begin{array}{l} t := x \oplus y \\ b := t \end{array} & \begin{array}{l} t := x \oplus y \\ c := t \end{array}
\end{tikzcd}$$

Step 5:

Replace

$$\begin{tikzcd}
w := x \oplus y \arrow[d] \\
w := t
\end{tikzcd}$$

Notes:

- *Copy propagation may eliminate some of the extra assignments (but might not)*
- *Program size could grow*
- *Want to limit this effect...*

  If more than 1 statement found in step 2, just forget it.

Copy Propagation

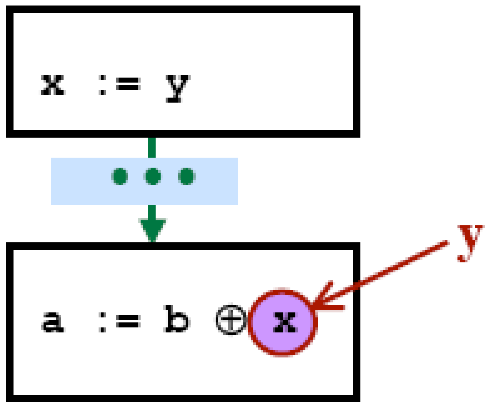
A copy statement

```
x := y
```

Where do the copies come from:

- IR code generation
- Common Sub-Expression Elimination
- Other Optimizations

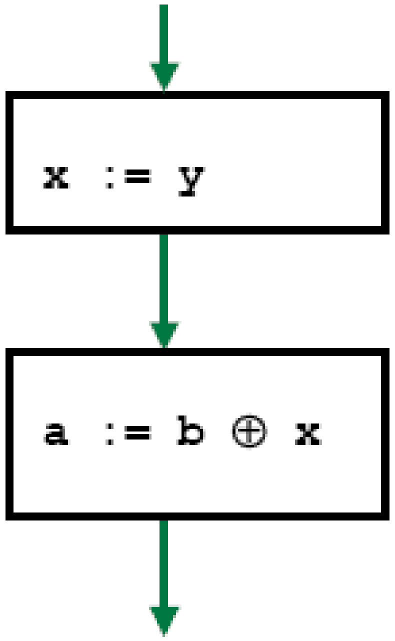
Copy Propagation



We can use y instead of x if...

- The only definition of x reaching a := b⊕x is x := y, and
- There is no assignment to y on any path
  from x := y to a := b⊕x.

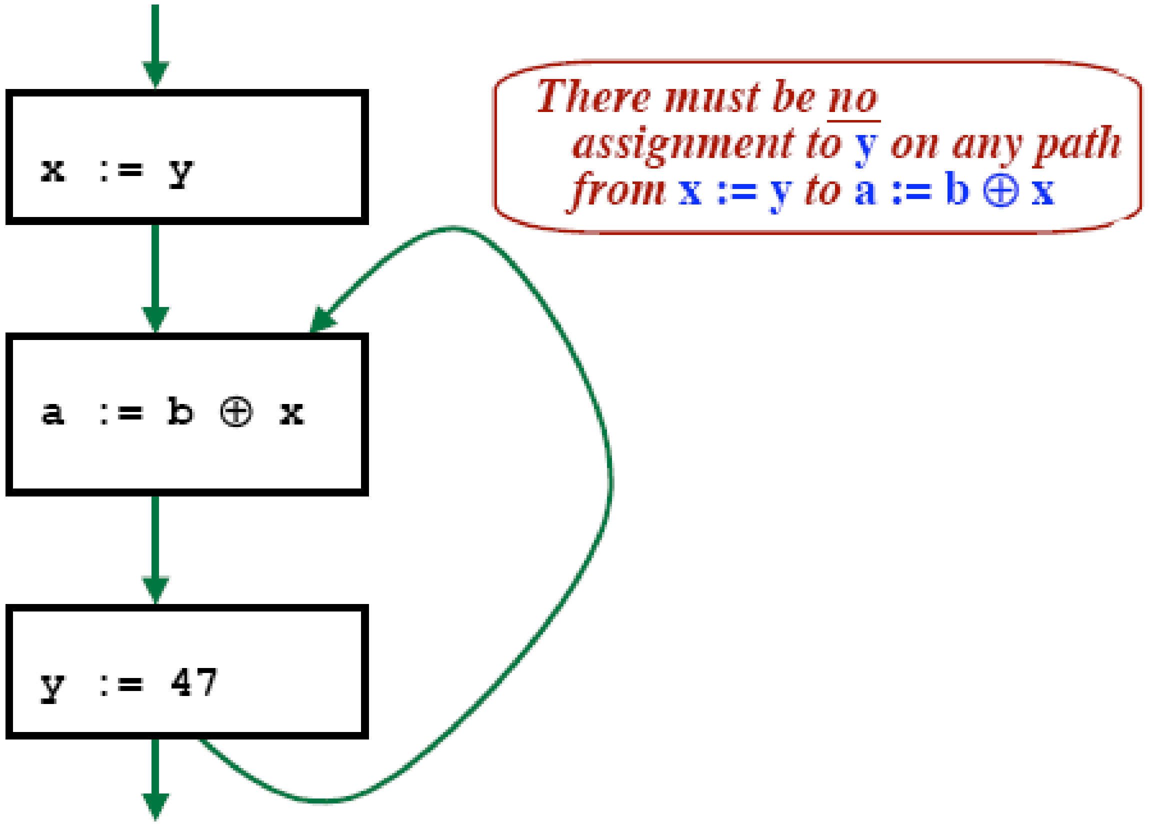
Copy Propagation



There must be no assignment to y *on any path from* x := y *to* a := b ⊕ x

Copy Propagation

We can not propagate the copy in this example:


x := y
a := b ⊕ x
y := 47
There must be no assignment to y on any path from x := y to a := b ⊕ x

Copy Propagation

We can use y instead of x if...

- The only definition of x reaching $a := b \oplus x$ is $x := y$, and

Compute the U-D Chains and use that info to determine this!

- There is no assignment to y on any path from $x := y$ to $a := b \oplus x$.

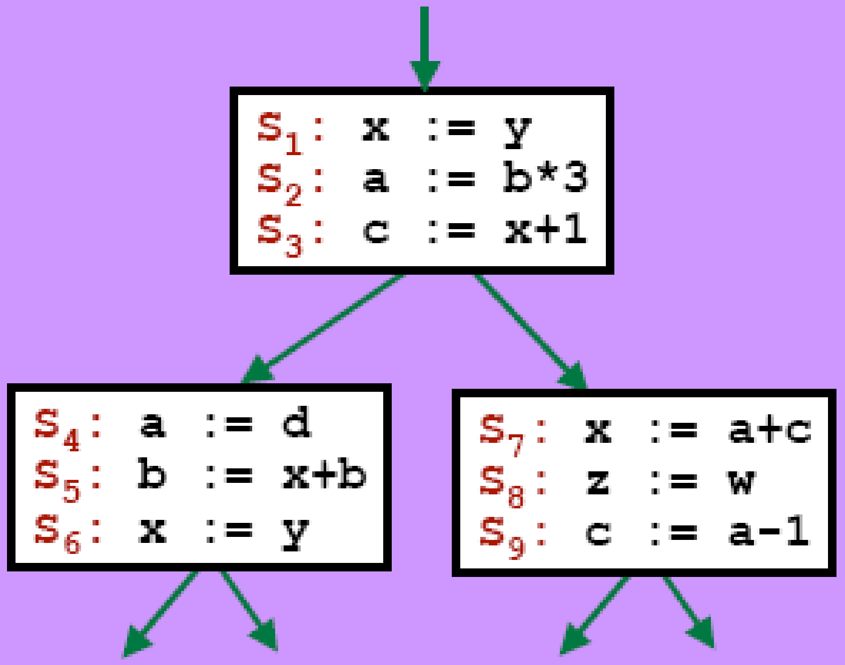
A new Data Flow problem!

Copy Propagation

Look at the entire Control Flow Graph
Identify all copy statements.
Two copy statements are different,
even if they have the same variables!

Example:

Universe = ???

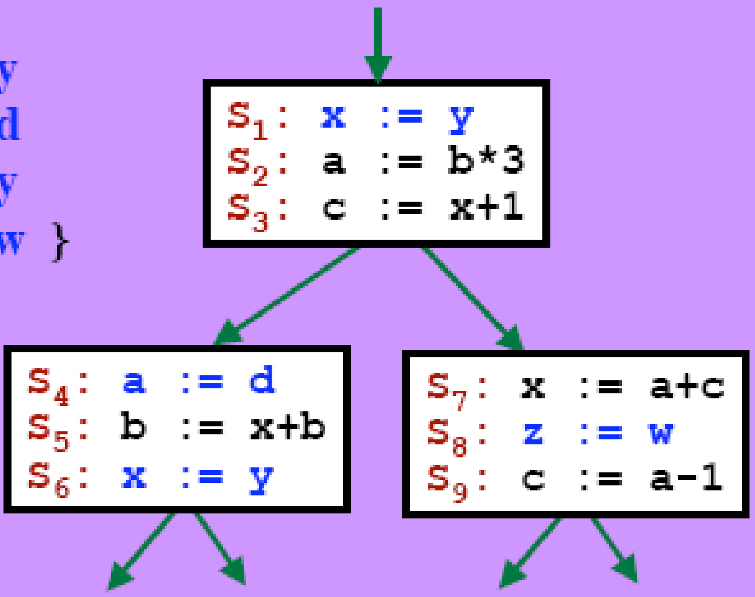


Copy Propagation

Look at the entire Control Flow Graph
Identify all copy statements.
Two copy statements are different,
even if they have the same variables!

Example:

Universe = { S_1: x := y
S_4: a := d
S_6: x := y
S_8: z := w }



```
S1: x := y
S2: a := b*3
S3: c := x+1
```

```
S4: a := d
S5: b := x+b
S6: x := y
```

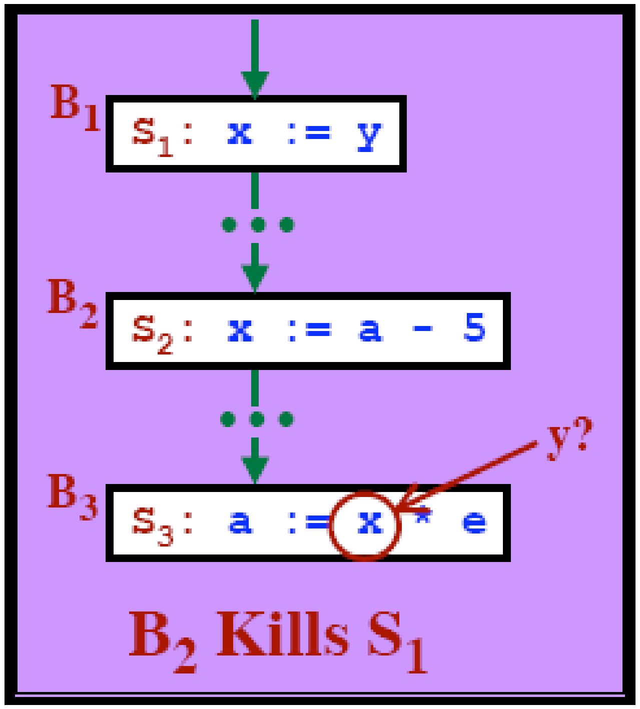
```
S7: x := a+c
S8: z := w
S9: c := a-1
```

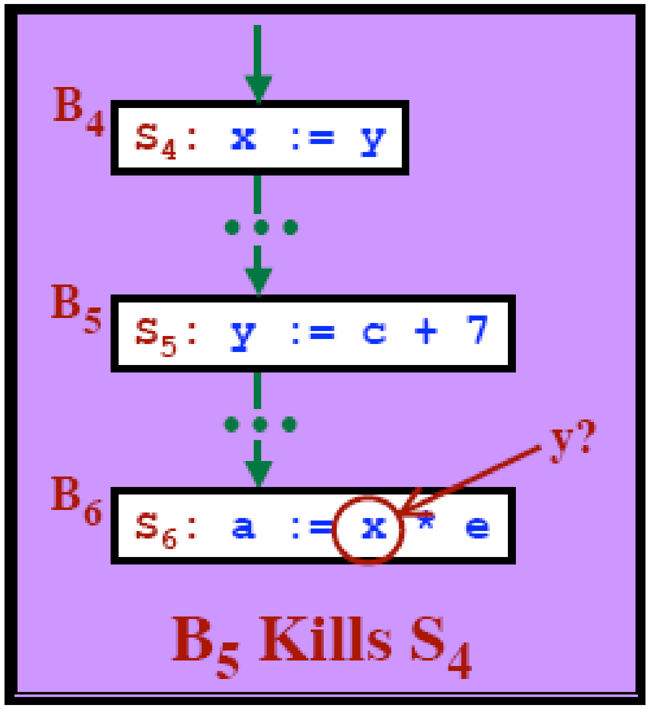
Copy Propagation

A block “kills” a copy

x := y

if it contains an assignment to x or y…





… unless the block contains the copy itself and does not assign to x or y after the copy.

Copy Propagation: Approach

For each basic block, we first compute...

C_GEN [B]

The set of all copy statements in basic block B, not killed before they reach the end of the block.

C_KILL [B]

The set of all copies in $\mathbb{U}$ that are killed by block B.

Copy Propagation: Approach

Then, Use Data Flow to Compute…

C_IN [B]

The set of all copy statements x := y such that every path from the initial block to the beginning of B contains the copy and there are no assignments to x or y on any path from the copy statement to the beginning of block B.

[Technically, there must be no assignments on the path between the last occurrence of the copy and the beginning of block B.]

C_OUT [B]

Same, at the end of the block.

The Data Flow Equations

$$\left.\begin{array}{l} \texttt{C\_OUT[B]} := \texttt{C\_GEN[B]} \cup (\texttt{C\_IN[B]} - \texttt{C\_KILL[B]}) \\ \texttt{C\_IN[B]} := \bigcap\limits_{\texttt{P is a predecessor of B}} \texttt{C\_OUT[P]} \end{array}\right\} \textit{For } B \neq B1 \textit{ (the initial block)}$$

$$\texttt{C\_IN}[\texttt{B}_1] = \{\}$$ *Nothing available before the initial block*

*These equations are identical to the **Available Expression** equations!*

Copy Deletion Algorithm

Input:

Control Flow Graph
U-D Chain info
D-U Chain info
Results of Data Flow Analysis; C_IN [B], for each block

Output:

Modified Flow Graph

Copy Deletion Algorithm

```
for each copy statement C: x:=y do
  Determine the set of all uses of x
                           that are reached by C.
  Call such stmts $U_1$, $U_2$, $U_3$, … $U_N$
  for each use $U_i$: … := … x… do
    Let B be the basic block containing $U_i$.
    if C ∈ C_IN[B] and there are no
                        definitions of x or y prior
                        to $U_i$ within B then
      It might be okay to delete C… Keep checking other uses.
    else
      We must not delete C!
      Skip to the next copy statement
    endif
  endfor
  delete C
  modify all uses $U_1$,$U_2$,…$U_N$
endfor
```

U_i: … := … x…

$\downarrow$

U_i: … := … y…

Loop Unrolling

Source:

```
for i := 1 to 100 by 1
    A[i] := A[i] + B[i];
endfor
```

Transformed Code:

```
for i := 1 to 100 by 4
    A[i  ] := A[i  ] + B[i  ];
    A[i+1] := A[i+1] + B[i+1];
    A[i+2] := A[i+2] + B[i+2];
    A[i+3] := A[i+3] + B[i+3];
endfor
```

Loop Unrolling

Source:

```
for i := 1 to 100 by 1
   A[i] := A[i] + B[i];
endfor
```

Transformed Code:

```
for i := 1 to 100 by 4
   A[i  ] := A[i  ] + B[i  ];
   A[i+1] := A[i+1] + B[i+1];
   A[i+2] := A[i+2] + B[i+2];
   A[i+3] := A[i+3] + B[i+3];
endfor
```

Larger Basic Blocks are Good!
More opportunities for optimizations such as scheduling

Benefits:

- The overhead of testing and branching is reduced.
- This optimization may "enable" other optimizations.

Loop Unrolling

Source:

```
for i := 1 to MAX by 1
   A[i] := A[i] + B[i];
endfor
```

Number of iterations is not known at compile-time.

Transformed Code:

```
i := 1;
 while (i+3 <= MAX) do
   A[i  ] := A[i  ] + B[i  ];
   A[i+1] := A[i+1] + B[i+1];
   A[i+2] := A[i+2] + B[i+2];
   A[i+3] := A[i+3] + B[i+3];
   i := i + 4;
endwhile
while (i <= MAX) do
   A[i] := A[i] + B[i];
   i := i + 1;
endwhile
```

Do 0 to 3 more iterations, as necessary, to finish

Loop-Invariant Computations

An assignment

```
        x  :=  y  ⊕  z
```

is "Loop-Invariant" if..

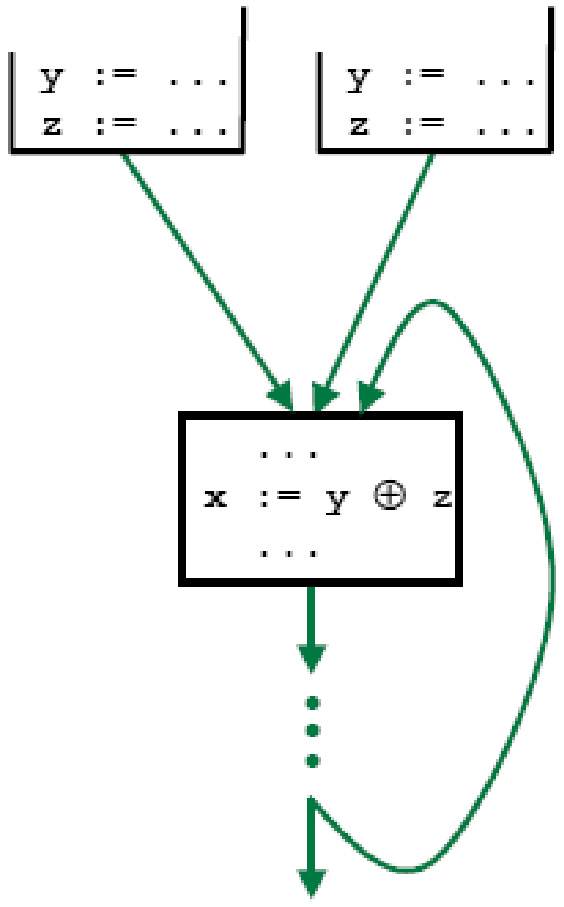
- It is in a loop, and
- All definitions of y and z that reach the statement are outside the loop.

We may be able to move the computation into the "preheader".

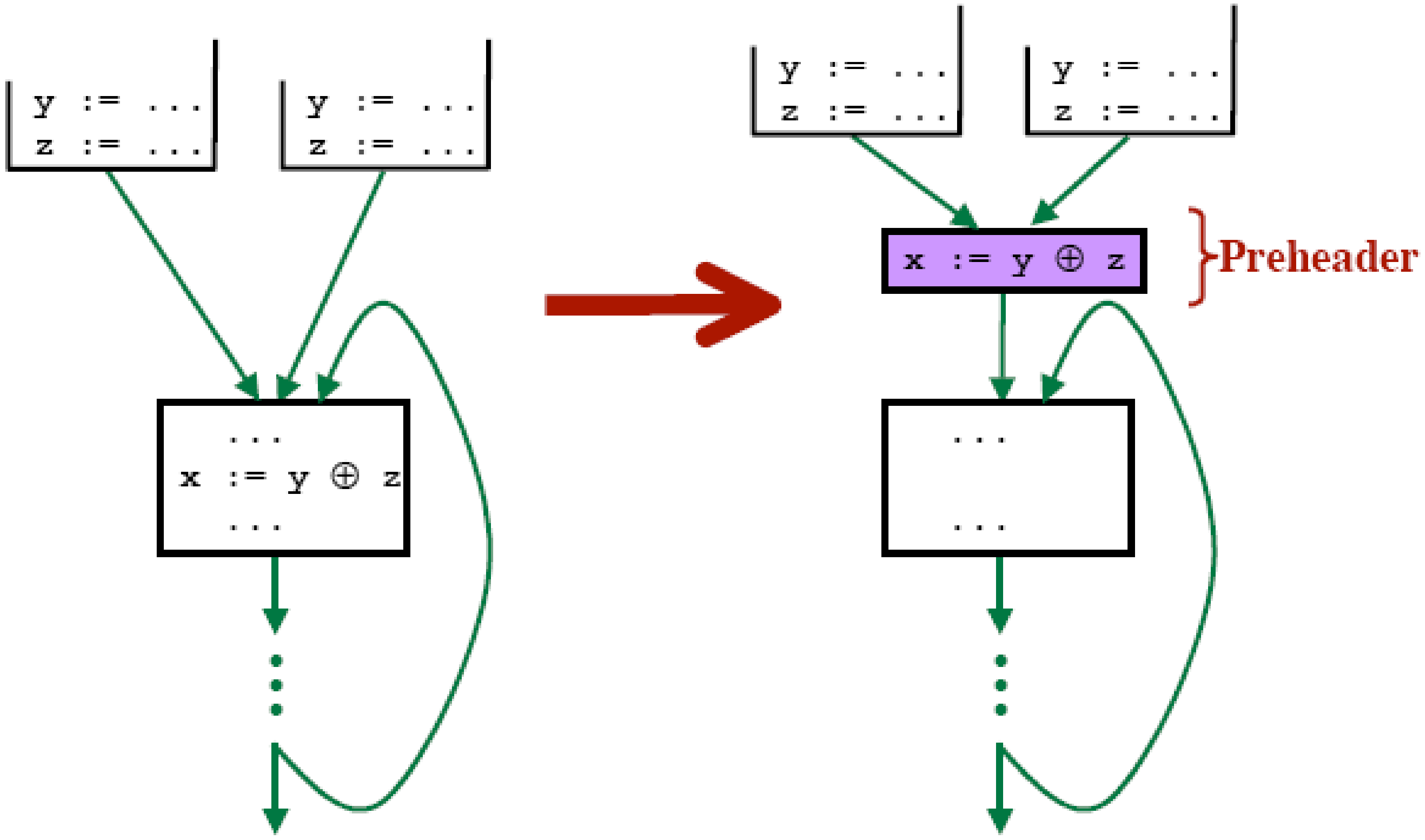
Step 1: Detect the Loop-Invariant Computations.

Step 2: See if it is okay to move the statement into the pre-header.

Loop-Invariant Computations: Example



Loop-Invariant Computations: Example



Detecting Loop-Invariant Computations

Input:

Loop L (= a set of basic blocks)
U-D Chain information

Output:

The set of loop-invariant statements.

Idea:

- Mark some of the statements as “loop-invariant”.
- This may allow us to mark even more statements as loop-invariant.
- Remember the order in which theses statements are marked.

Detecting Loop-Invariant Computations

```
repeat until no new statements are marked...
  Look at each statement in the loop.
  If all its operands are unchanging then
    mark the statement as "loop-invariant".
  An operand is "unchanging" if...
        • It is a constant
        • It has all reaching definitions
              outside of the loop
        • It has exactly one reaching definition
              and that definition has already
              been marked "loop-invariant".
end
```

Remember the order in which statements are marked "loop-invariant."

Moving Loop-Invariant Computations

Consider moving statement

S: $x := y \oplus z$

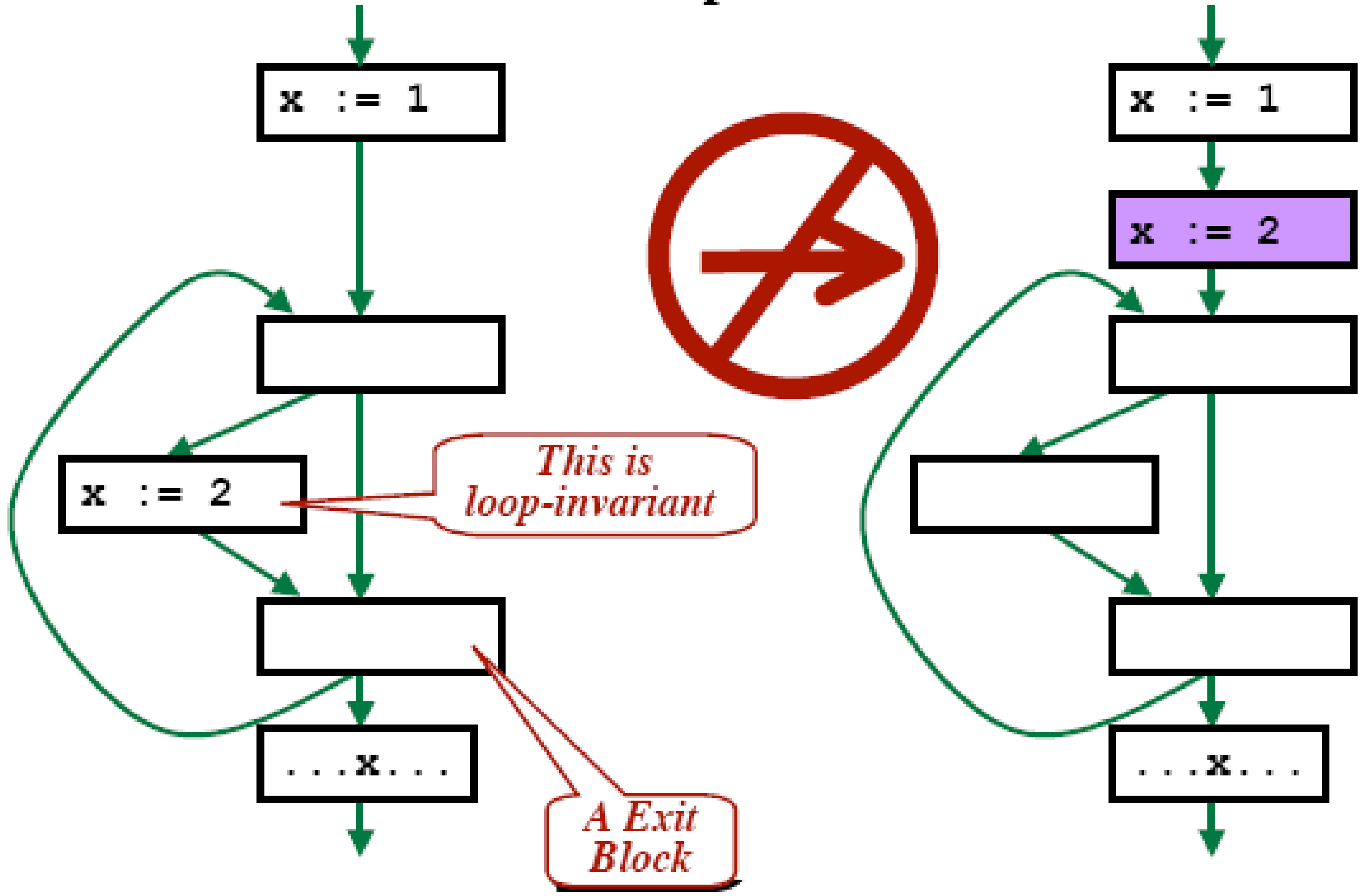
into the loop's preheader.

The statement must satisfy three conditions.

If it satisfies all conditions, then it can be moved.

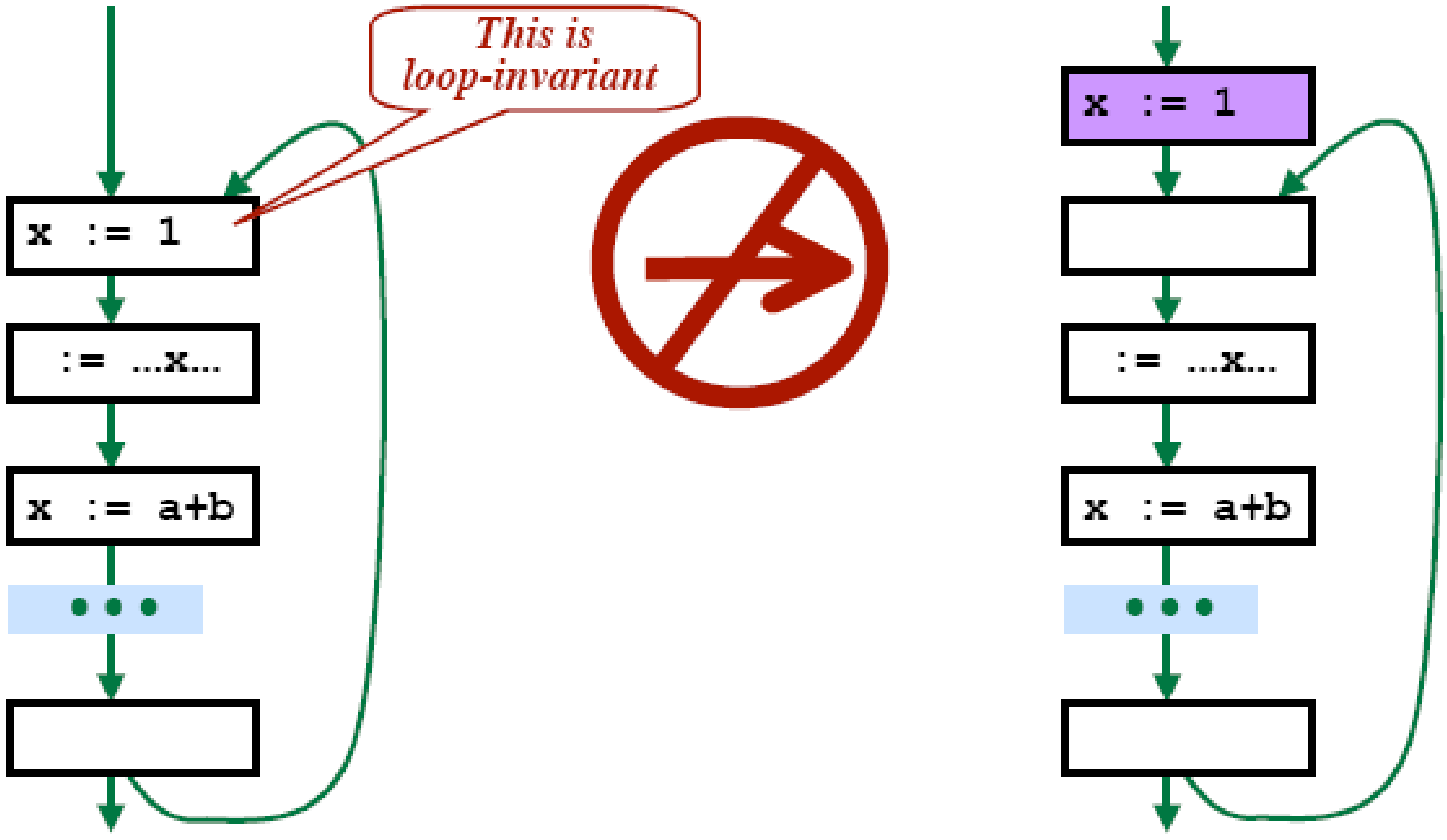
Moving Condition 1

The block containing S must dominate all exits from the loop.



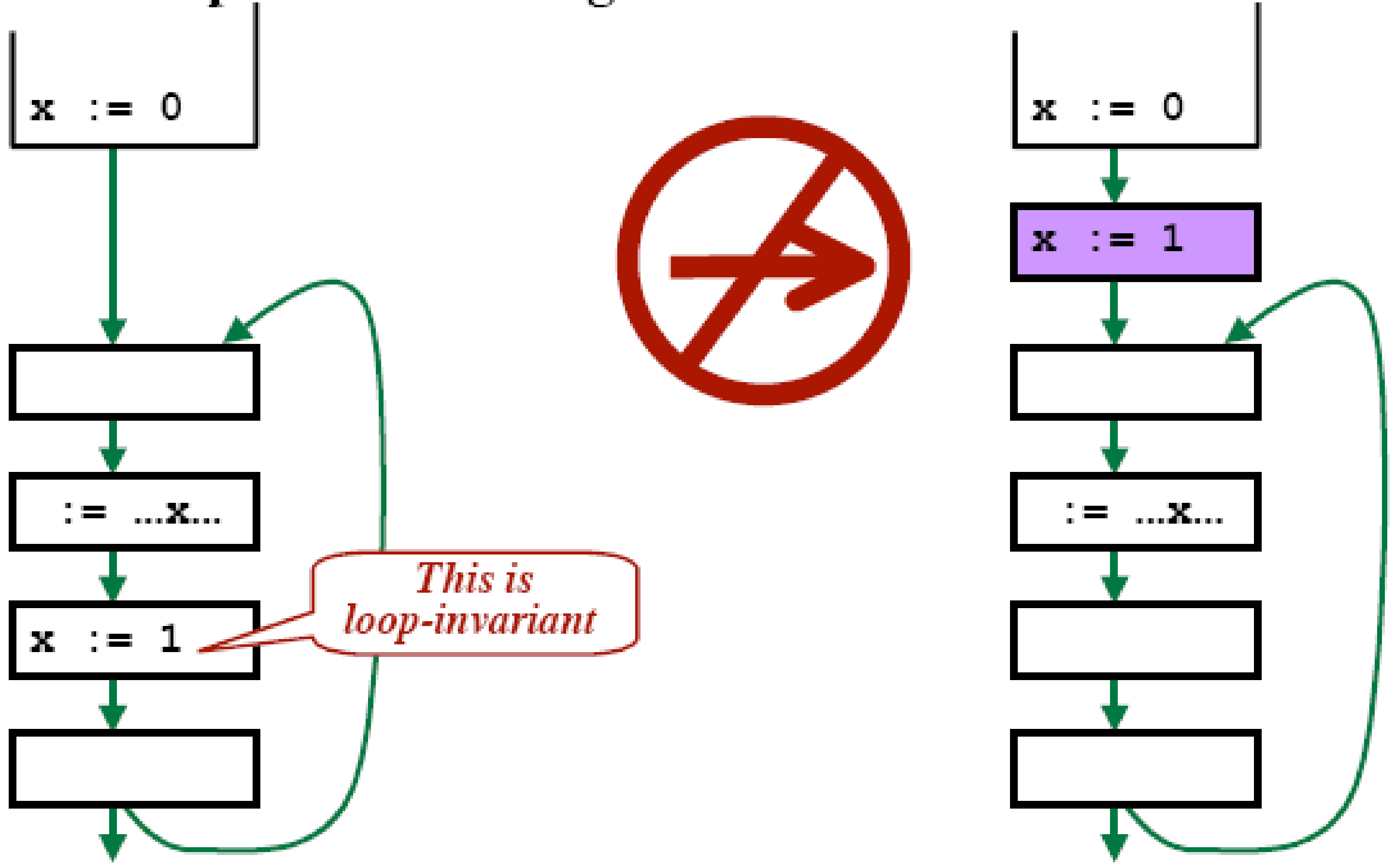
Moving Condition 2

There must be no other assignments to “x” in the loop.



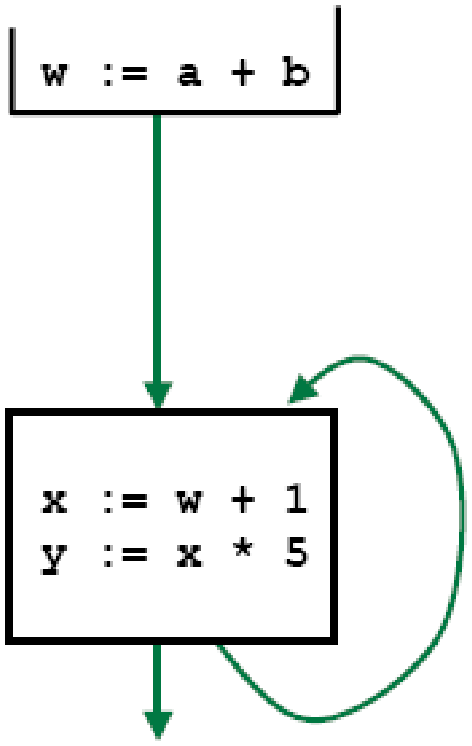
Moving Condition 3

All uses of “x” in the loop must be reached by ONLY the loop-invariant assignment.



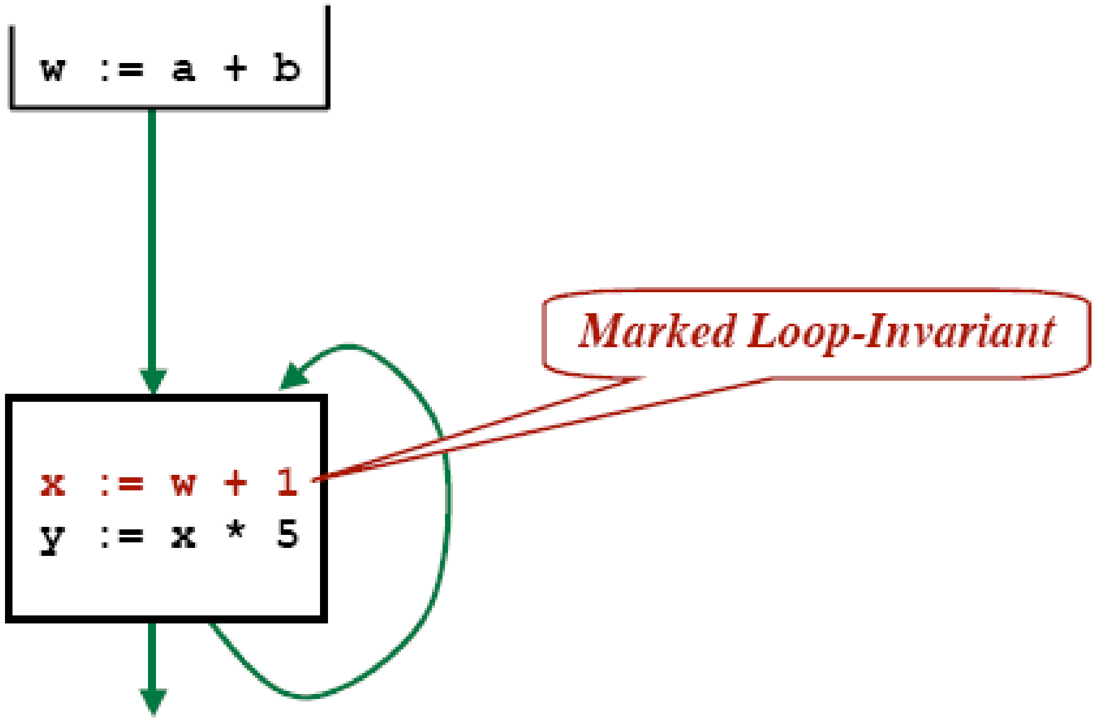
Loop Invariant Computation

If all three conditions are satisfied,
move the statements into the preheader
in the order they were marked Loop-Invariant.



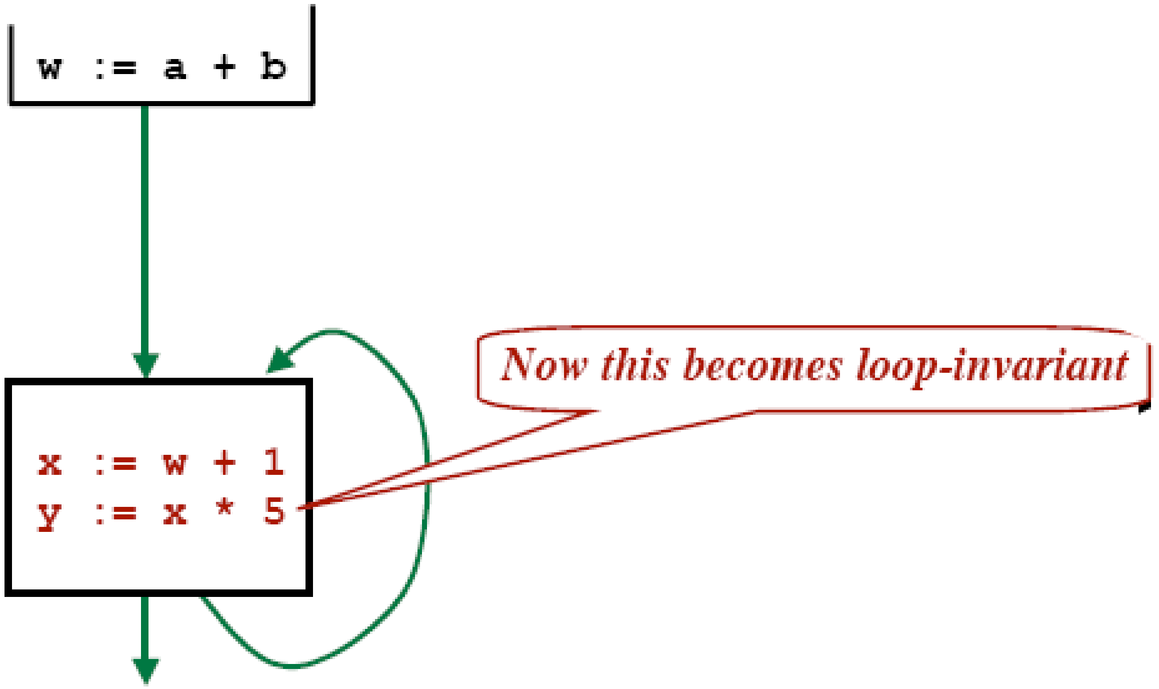
Loop Invariant Computation

If all three conditions are satisfied, move the statements into the preheader in the order they were marked Loop-Invariant.



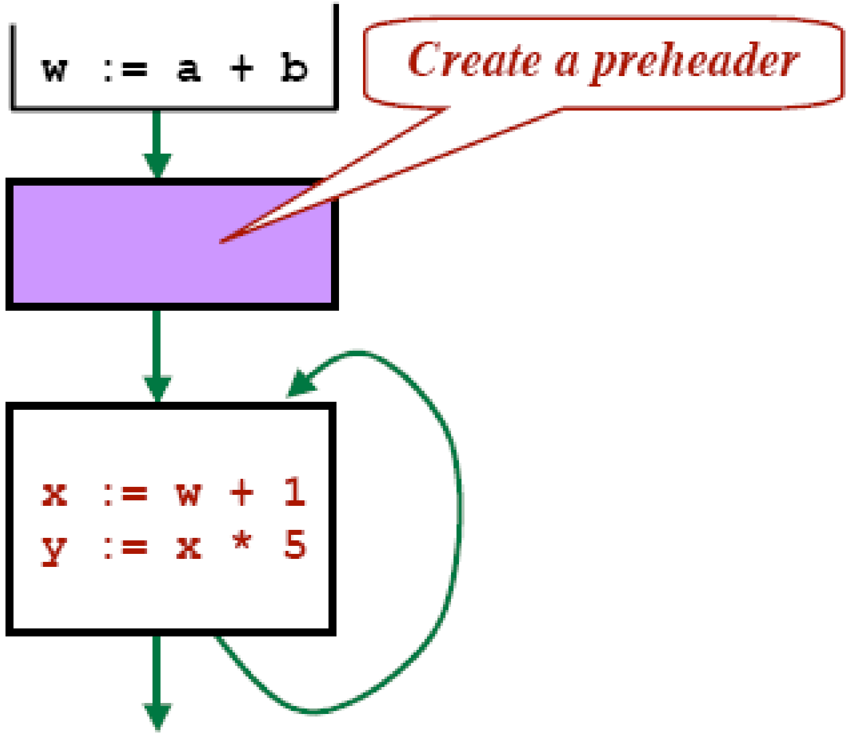
Loop Invariant Computation

If all three conditions are satisfied,
move the statements into the preheader
in the order they were marked Loop-Invariant.



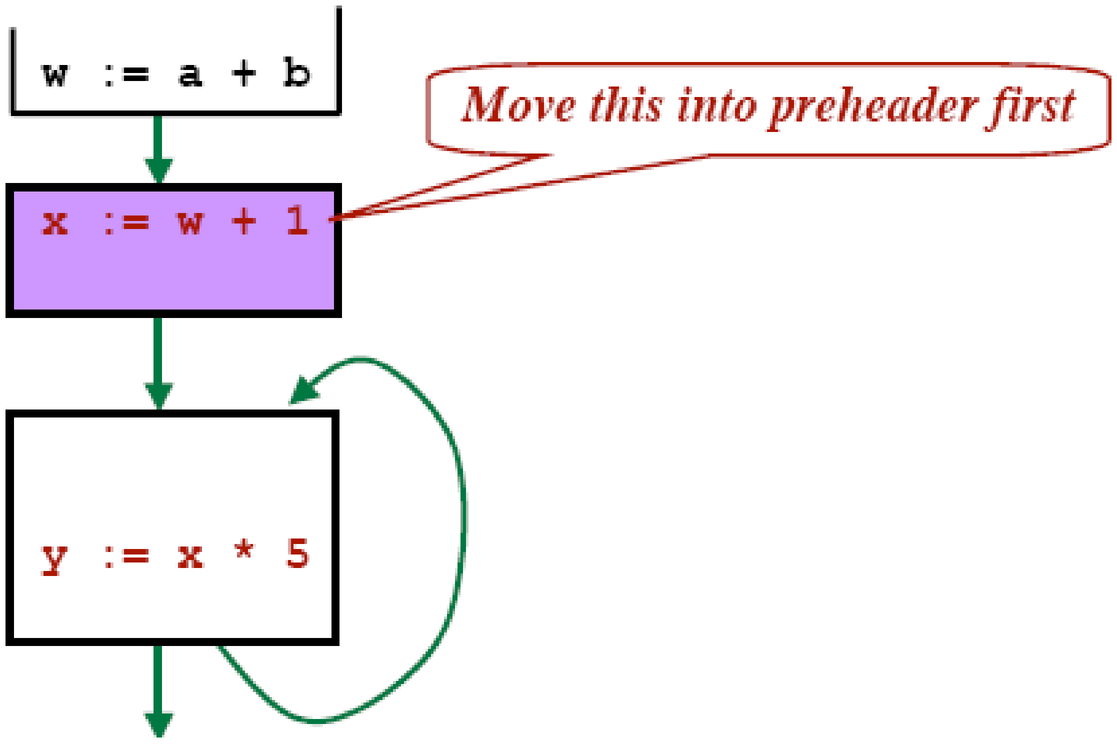
Loop Invariant Computation

**If all three conditions are satisfied,
move the statements into the preheader
in the order they were marked Loop-Invariant.**



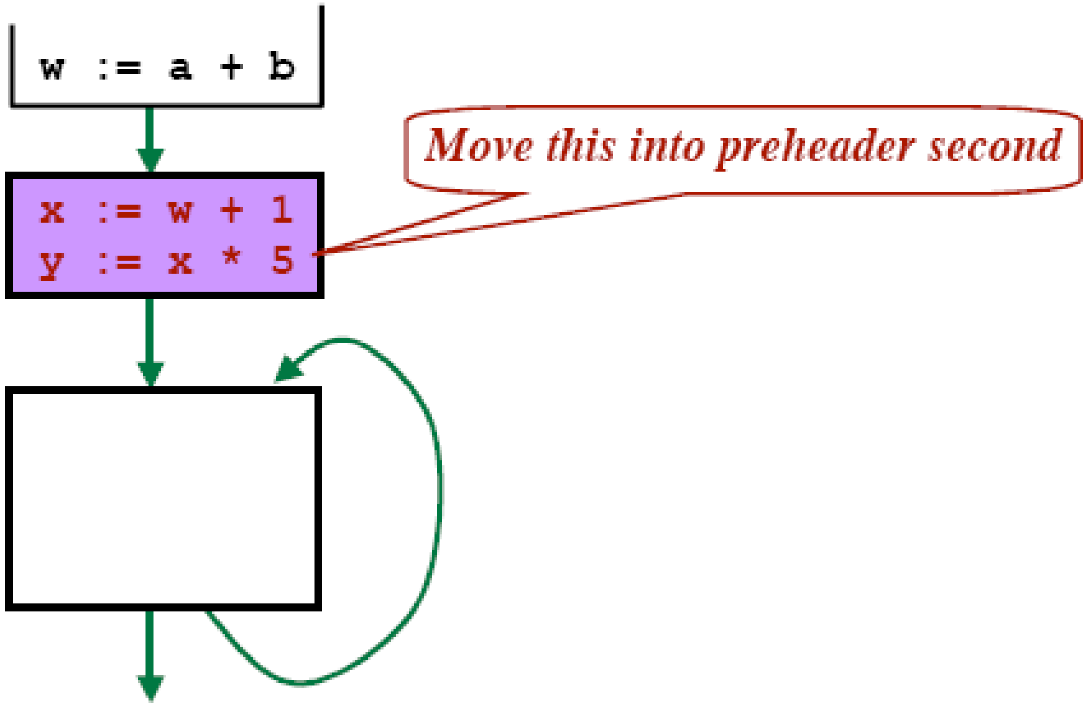
Loop Invariant Computation

If all three conditions are satisfied, move the statements into the preheader in the order they were marked Loop-Invariant.



Loop Invariant Computation

**If all three conditions are satisfied,
move the statements into the preheader
in the order they were marked Loop-Invariant.**


w := a + b
x := w + 1
y := x * 5
Move this into preheader second