CS620T: Compilers – Principles and Implementation

Autumn 2026

Week 12: Dataflow Analysis

Useful optimizations

- Common subexpression elimination (global)
  - Need to know which expressions are available at a point
- Dead code elimination
  - Need to know if the effects of a piece of code are never needed, or if code cannot be reached
- Constant folding
  - Need to know if variable has a constant value
- So how do we get this information?

Dataflow analysis

- Framework for doing compiler analyses to drive optimization
- Works across basic blocks
- Examples
  - Constant propagation: determine which variables are constant
  - Liveness analysis: determine which variables are live
  - Available expressions: determine which expressions have valid computed values
  - Reaching definitions: determine which definitions could "reach" a use

Dataflow Analysis - Common Traits

Common requirement among global optimizations:

- Know a particular **property X** at a *program point*

  (There is a program point one before a statement and one after a statement)

  - Say that property X definitely holds.

    OR

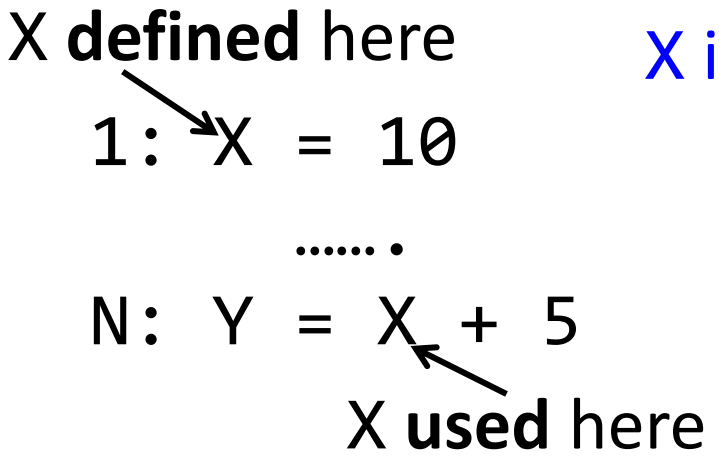
  - Don't know if property X holds or not (okay to be conservative)

  This requires the knowledge of entire program

Dataflow analysis

- Framework for doing compiler analyses to drive optimization
- Works across basic blocks
- Examples
  - Constant propagation: determine which variables are constant
  - Liveness analysis: determine which variables are live
  - Available expressions: determine which expressions have valid computed values
  - Reaching definitions: determine which definitions could "reach" a use

Liveness – Recap..



X is live at 1

..used in future

- A variable X is live at statement S if:
  - There is a statement S' that uses X
  - There is a path from S to S'
  - There are no intervening definitions of X

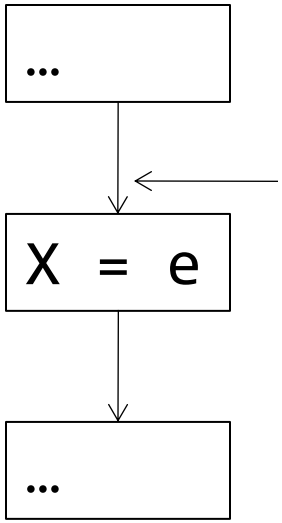
Liveness – Recap..

```
1: X = 10
2: X = Y + 2
   ….
N: Y = X + 5
```

X is dead at 1

- A variable X is dead at statement S if it is not live at S
  - What about …; `X = X + 1`?

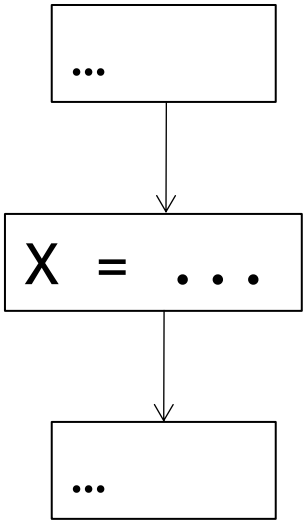
Liveness in a CFG



Given that e does not use X, X is *definitely dead* here (i.e. before the statement).

• Define a set `LiveIn(b)`, where b is a basic block, as: the set of all variables live at the entrance of a basic block

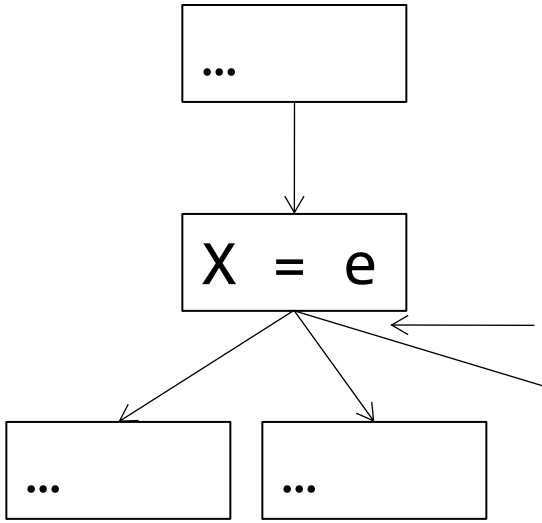
Liveness in a CFG



X is defined here

- Define a set `Def(b)`, where b is a basic block, as: the set of all variables that are defined in b

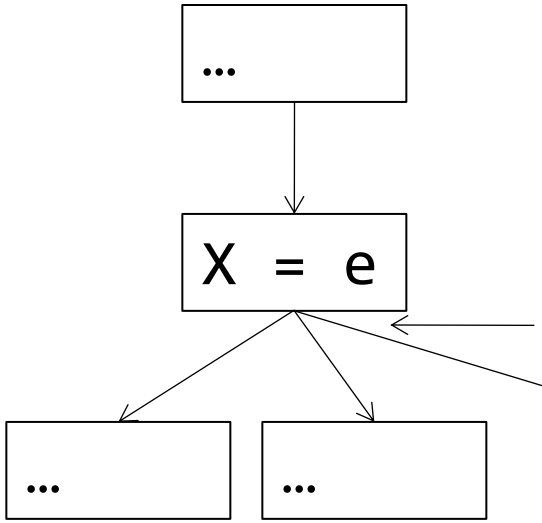
Liveness in a CFG



If X is live here (i.e. after the statement), X is used in *some* successor

• Define a set `LiveOut(b)`, where b is a basic block, as: the set of all variables live at the exit of a basic block

Liveness in a CFG

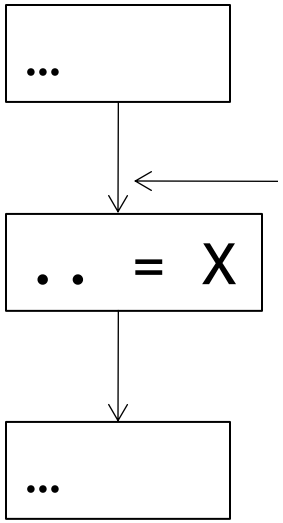


If X is live here (i.e. after the statement), X is used in *some* successor

- If $S(b)$ is the set of all successors of b, then

$$\text{LiveOut}(b) = \bigcup_{i \in S(b)} \text{LiveIn}(i)$$

Liveness in a CFG

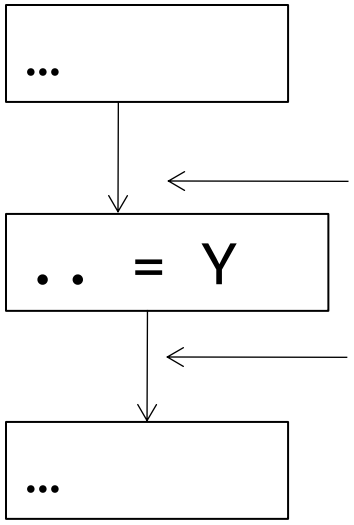


X *must be* live here (i.e. before the statement)

• Define a set `LiveUse(b)`, where b is a basic block, as the set of all variables that are used before they are defined within block b. `LiveIn(b)` ⊇ `LiveUse(b)`

Liveness in a CFG - Observation

- If a node neither uses nor defines X, the liveness property remains the same before and after executing the node



X not live here / X is live here

If X is not live here / X is live here

Liveness in a CFG

- If a variable is live on exit from b, it is either defined in b or live on entrance to b

$$\text{LiveIn}(b) \supseteq \text{LiveOut}(b) - \text{Def}(b)$$

- Under what scenarios can a variable be live at the entrance of a basic block?

Liveness in a CFG

- If a variable is live on exit from b, it is either defined in b or live on entrance to b

$$\text{LiveIn}(b) \supseteq \text{LiveOut}(b) - \text{Def}(b)$$

- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block

Liveness in a CFG

- If a variable is live on exit from b, it is either defined in b or live on entrance to b

$$\text{LiveIn}(b) \supseteq \text{LiveOut}(b) - \text{Def}(b)$$

- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block
  - OR the variable is live at exit and not defined within the block

Liveness in a CFG

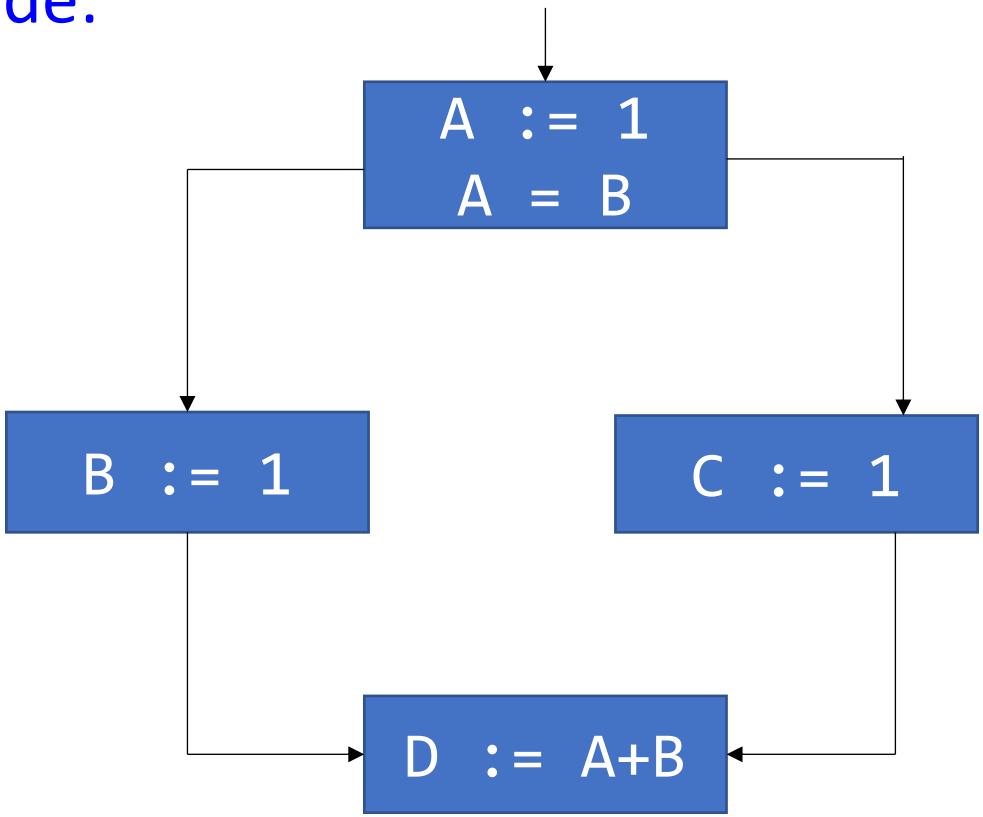
- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block
  - OR the variable is live at exit and not defined within the block

```
LiveIn(b) = LiveUse(b) U  (LiveOut(b) – Def(b))
```

Liveness in a CFG - Example

- Draw CFG for the code:

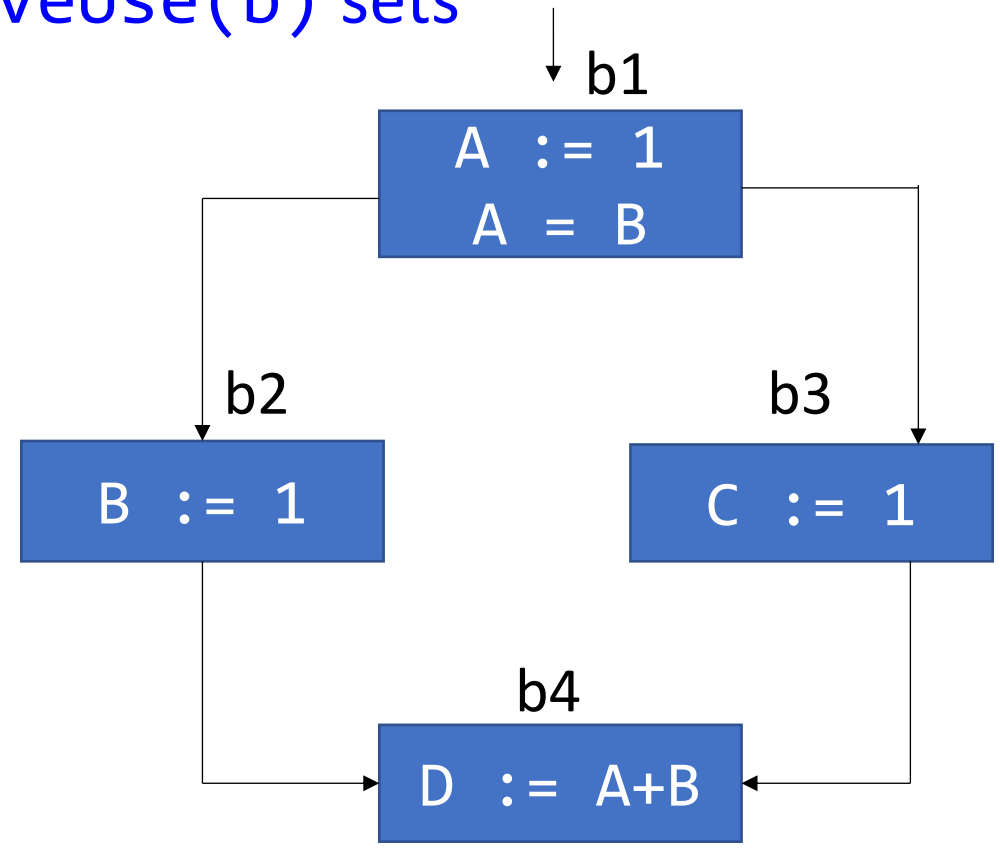
```
A:=1
if A=B then
   B:=1
else
   C:=1
endif
D:=A+B
```



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

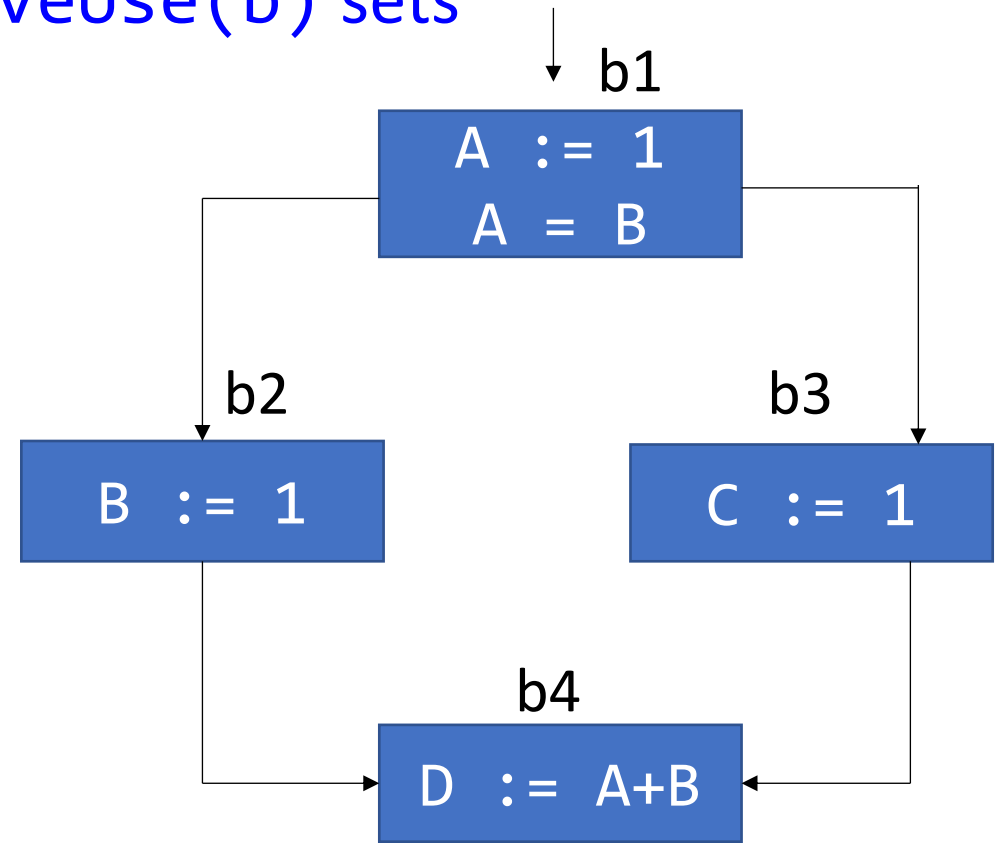
Block	Def	LiveUse
b1		
b2		
b3		
b4		



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

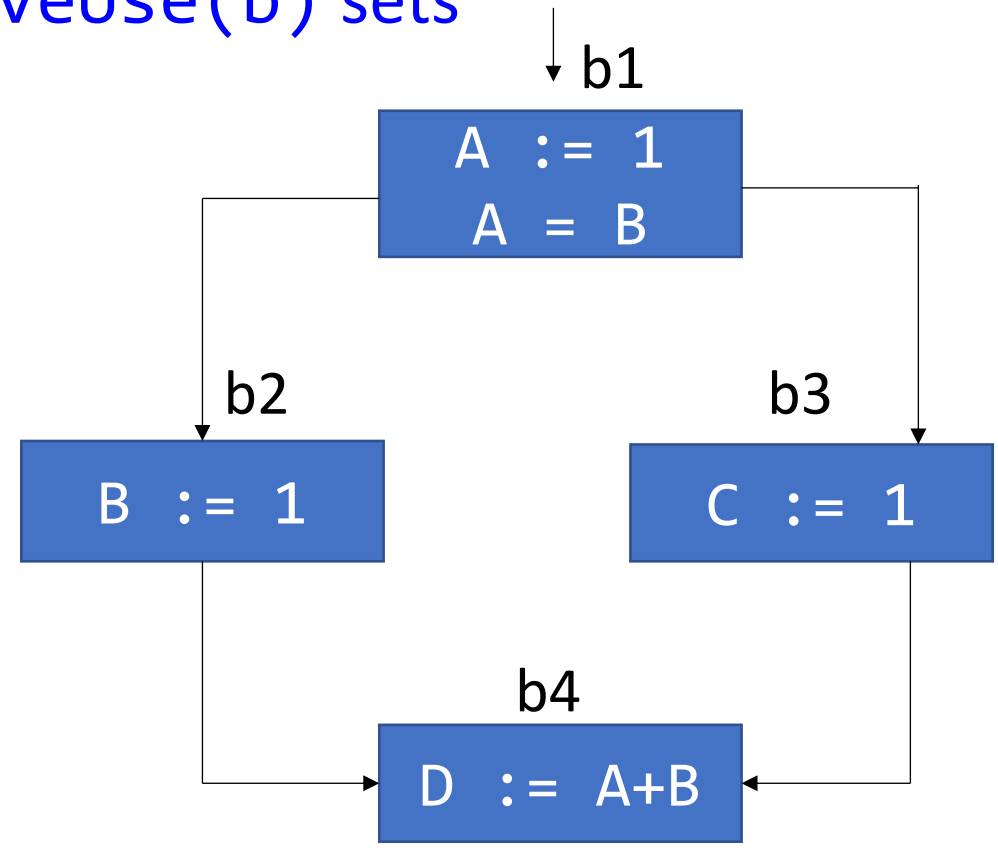
Block	Def	LiveUse
b1	{A}	{B}
b2		
b3		
b4		



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

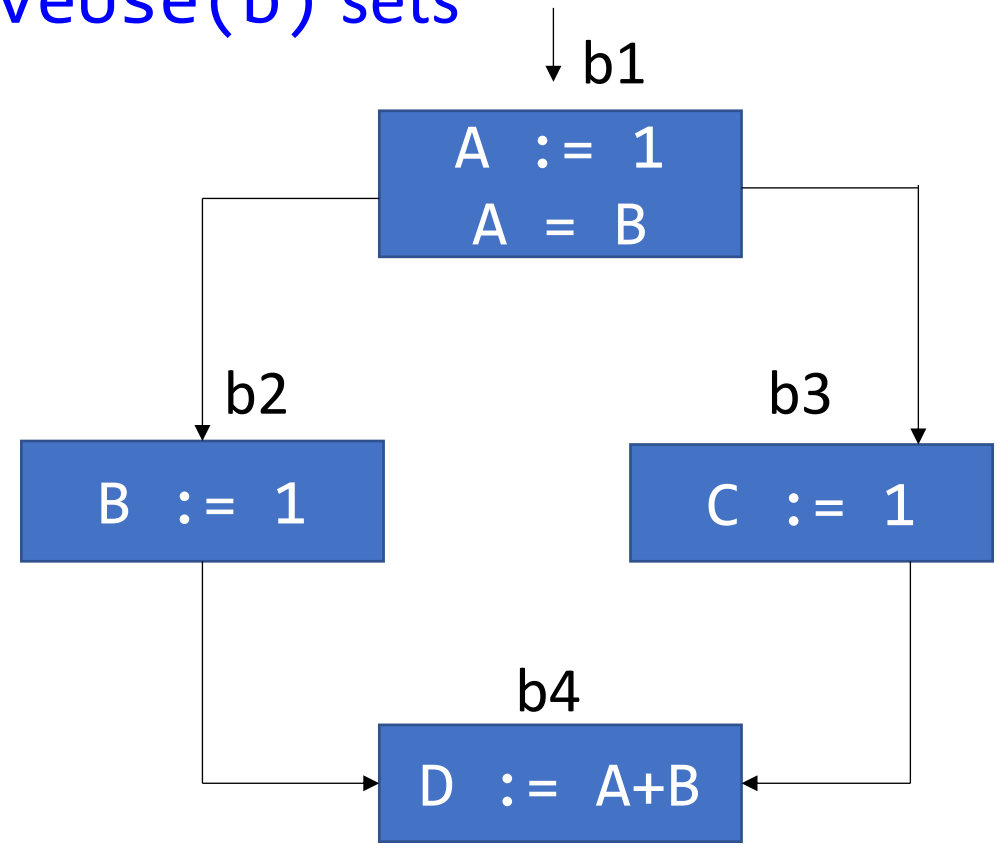
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3		
b4		



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

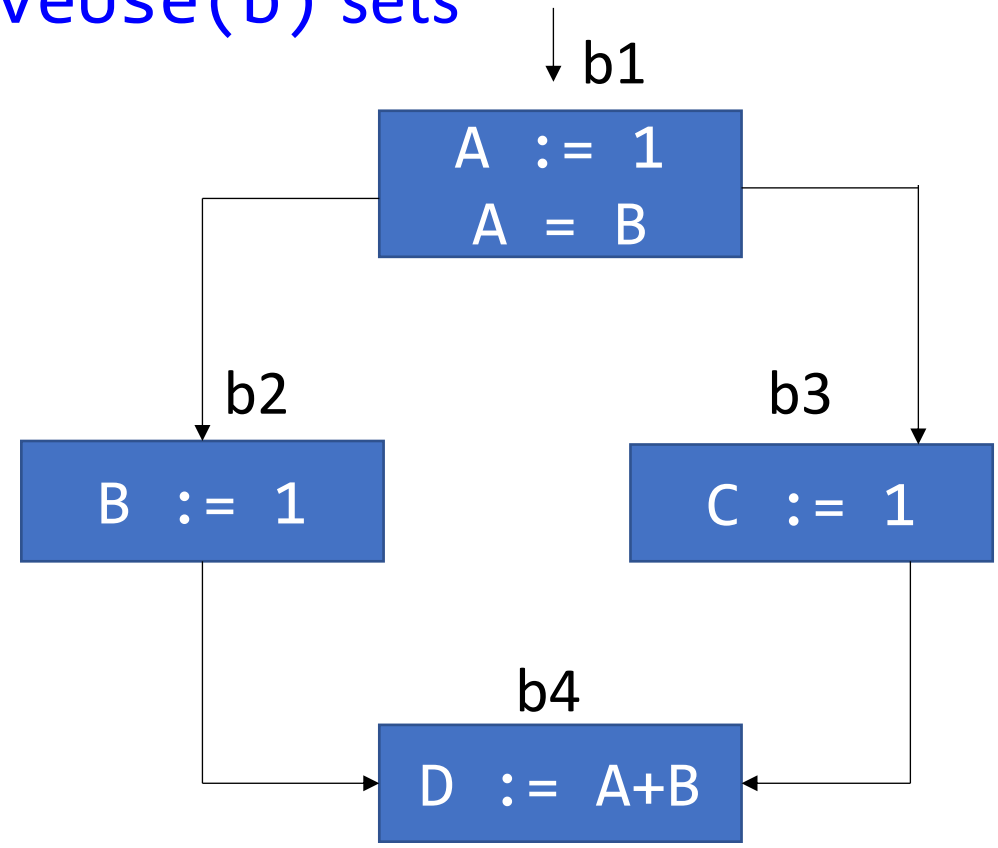
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4		



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

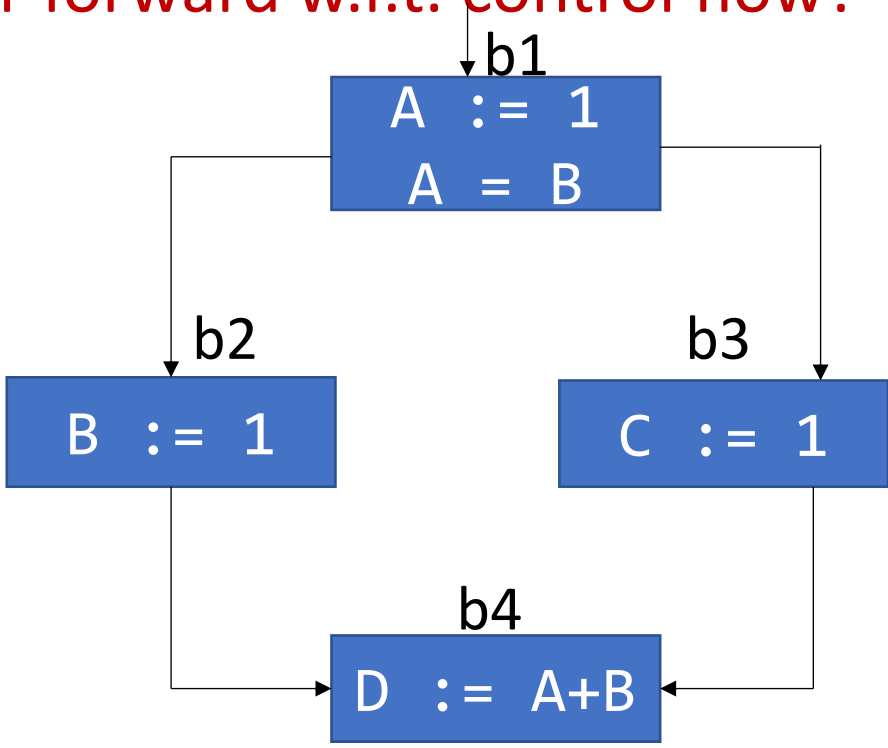


Liveness in a CFG - Example

- start from use of a variable to its definition.

Is this analysis going backward or forward w.r.t. control flow?

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

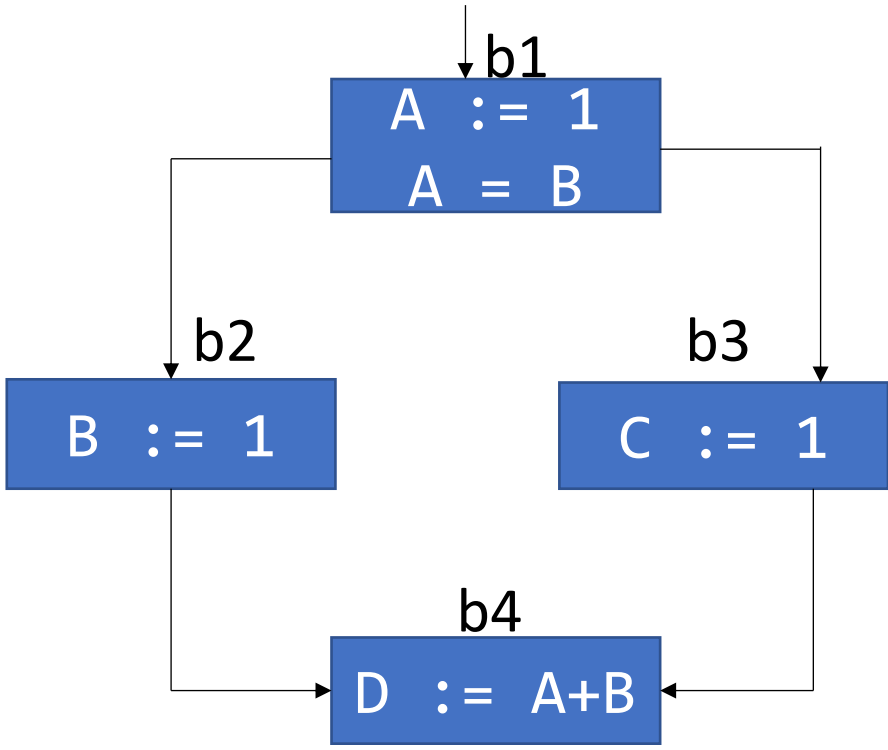


Liveness in a CFG - Example

- start from use of a variable to its definition.

Backward-flow problem

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

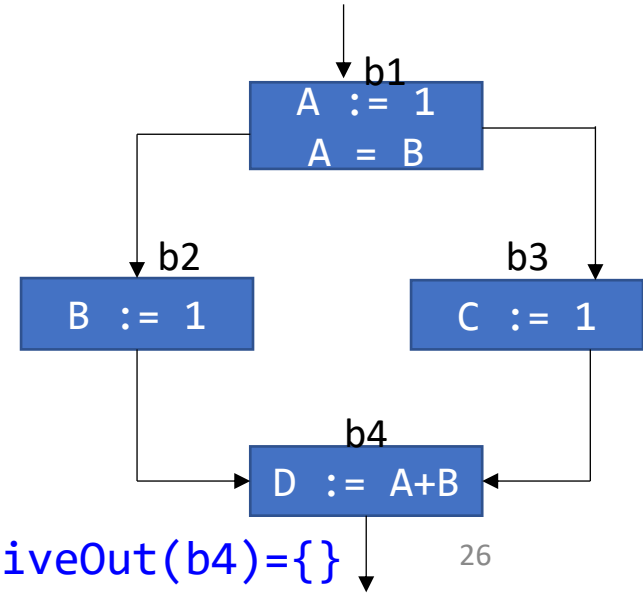


Liveness in a CFG - Example

- Start from use of a variable to its definition.
- Compute LiveOut and LiveIn sets:

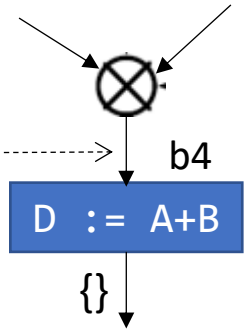
$$LiveIn(b) = LiveUse(b) \cup (LiveOut(b) - Def(b))$$

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}



Liveness in a CFG - Example

```
LiveIn(b4) = LiveUse(b4) U  (LiveOut(b4) – Def(b4))
           = {A,B} U  ({} – {D})
```



Program point

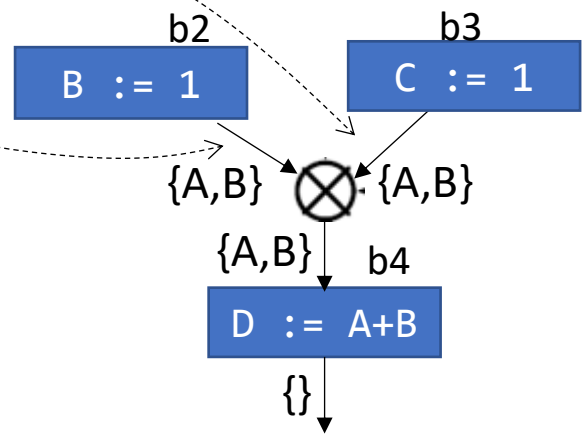
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Liveness in a CFG - Example

$LiveOut(b) = \bigcup_{i \in S(b)} LiveIn(i)$

LiveOut(b3) = LiveIn(b4) = {A,B}

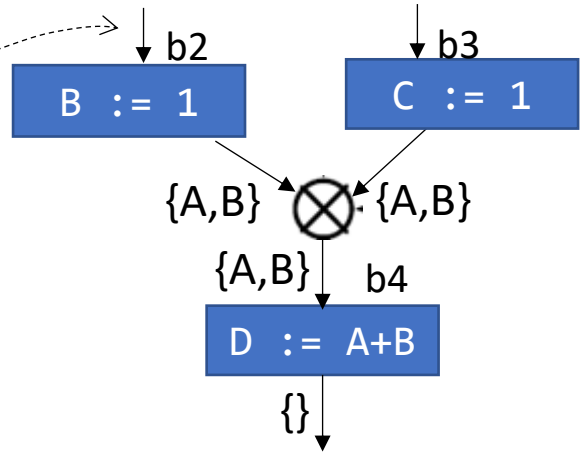
LiveOut(b2) = LiveIn(b4) = {A,B}



Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Liveness in a CFG - Example

```
LiveIn(b3) = LiveUse(b3) ∪  (LiveOut(b3) – Def(b3))
           = {} ∪  ({A,B} – {C}) = {A,B}
LiveIn(b2) = LiveUse(b2) ∪  (LiveOut(b2) – Def(b2))
           = {} ∪  ({A,B} – {B}) = {A}
```



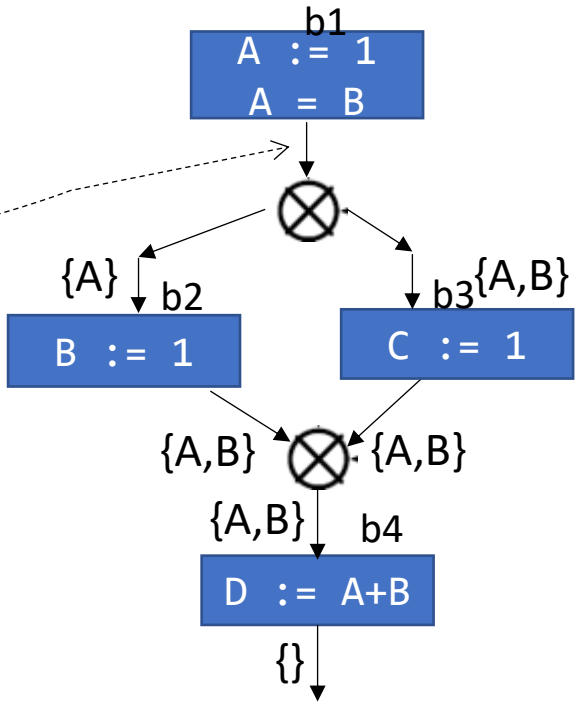
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Liveness in a CFG - Example

$\texttt{LiveOut(b)} = \bigcup_{i \in S(b)} \text{LiveIn(i)}$

$\texttt{LiveOut(b1)} = \text{LiveIn(b2)} \cup \text{LiveIn(b3)}$

$= \{A\} \cup \{A,B\} = \{A,B\}$

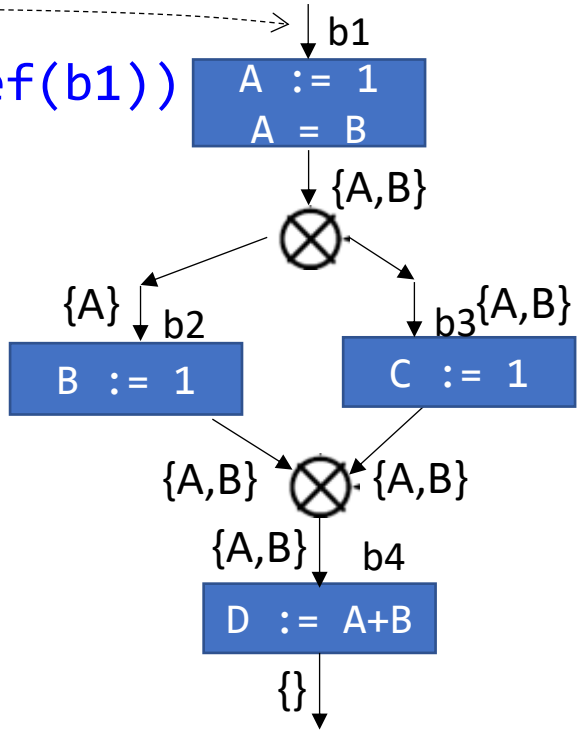


Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Liveness in a CFG - Example

LiveIn(b1) = LiveUse(b1) ∪ (LiveOut(b1) – Def(b1))

= {B} ∪ ({A,B} – {A}) = {B}



Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Liveness in a CFG - Example

- Summary: Compute `LiveIn(b)` and `LiveOut(b)`

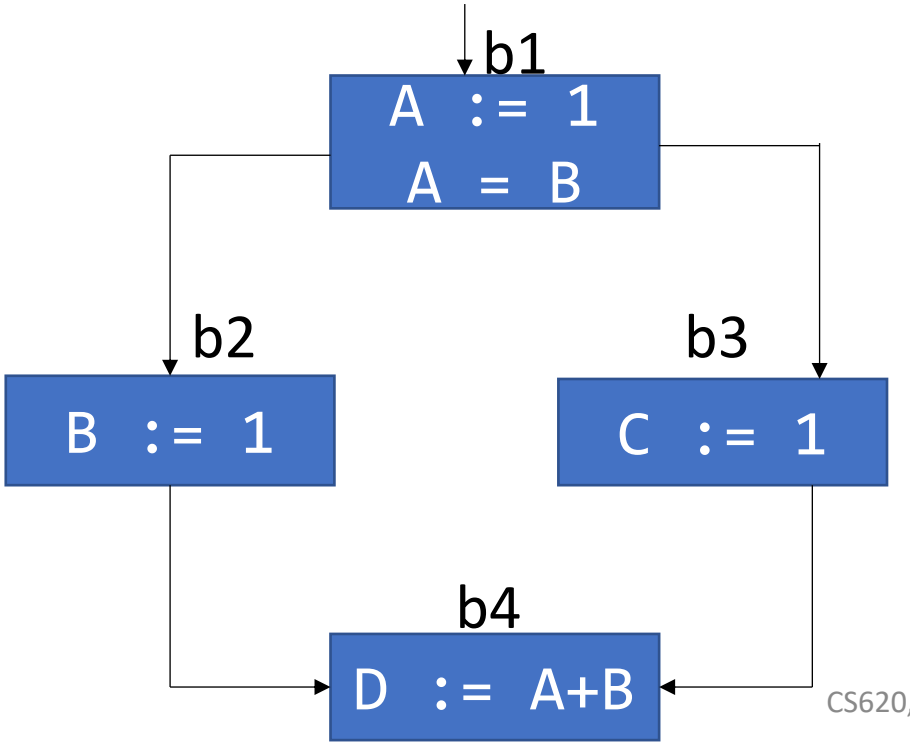
`LiveIn(b) = LiveUse(b) ∪ (LiveOut(b) – Def(b))`

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Block	LiveIn	LiveOut
b1	{B}	{A,B}
b2	{A}	{A,B}
b3	{A,B}	{A,B}
b4	{A,B}	{}

Liveness in a CFG – Use Case

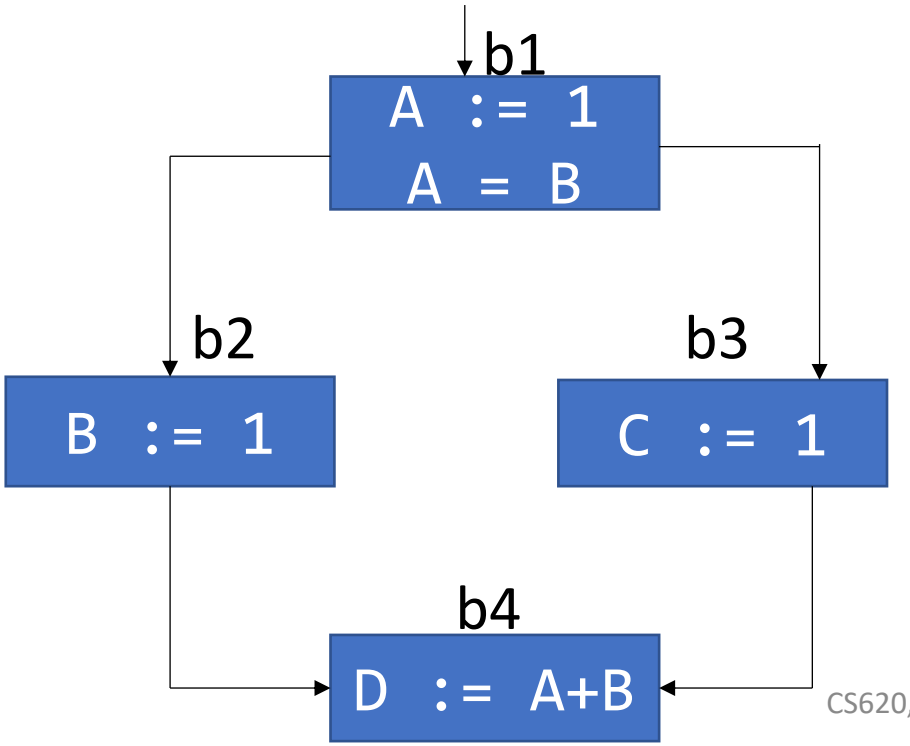
- Assume that the CFG below represents *your entire program* (b1 is the entry to program and b4 is the exit)
  - What can you infer from the table?



Block	LiveIn	LiveOut
b1	{B}	{A,B}
b2	{A}	{A,B}
b3	{A,B}	{A,B}
b4	{A,B}	{}

Liveness in a CFG – Use Case

- Assume that the CFG below represents *your entire program*
  - Variable B is live at the entrance of b1, the entry basic block of CFG. This implies that B is used before it is defined. An error!



Block	LiveIn	LiveOut
b1	{B}	{A,B}
b2	{A}	{A,B}
b3	{A,B}	{A,B}
b4	{A,B}	{}

Liveness in a CFG – Use Case

- Liveness information tells us what variable is dead. Can remove statements that assign to dead variables.

X is dead here implies that we can remove this statement.

```
X = 1
Y = X + 2
Z = Y + A
```



```
X = 1
Y = 1 + 2
Z = Y + A
```



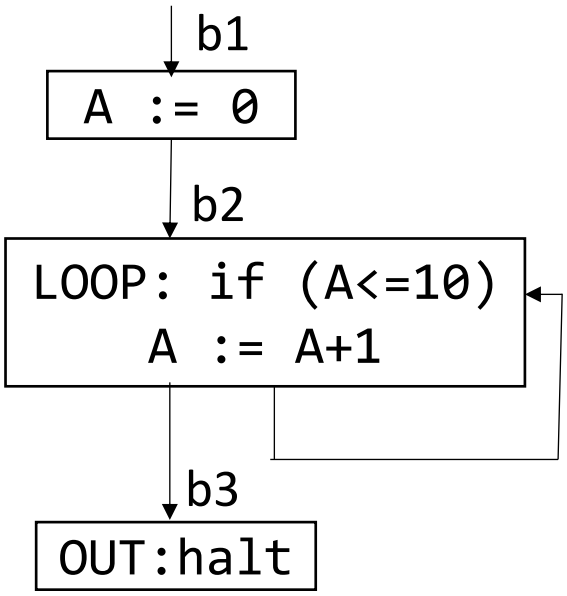
```
X = 1
Y = 1 + 2
Z = Y + A
```

Constant Propagation

Dead Code Elimination

Liveness in a CFG – Example (Loop)

- How do we compute liveness information when a loop is present?



Block	Def	LiveUse
b1	{A}	{}
b2	{A}	{A}
b3	{}	{}

Block	LiveIn	LiveOut
b1	{}	{A}
b2	{A}	{A}
B3	{}	{}

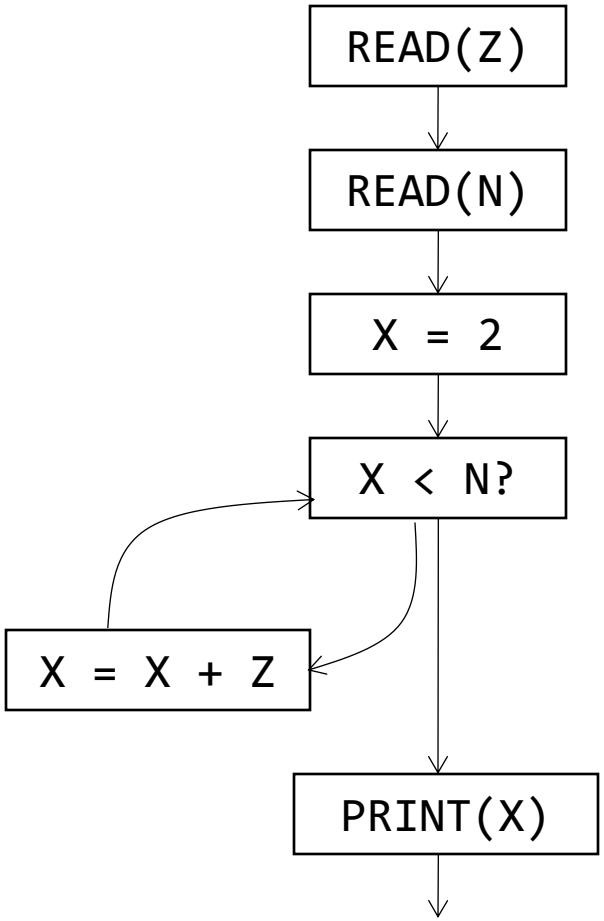
Liveness in a CFG - Observations

- Liveness is computed as information is *transferred* between adjacent statements
- At a program point, a variable can be live or not live (property: true or false)
  - To begin with we did not have any information=property is false

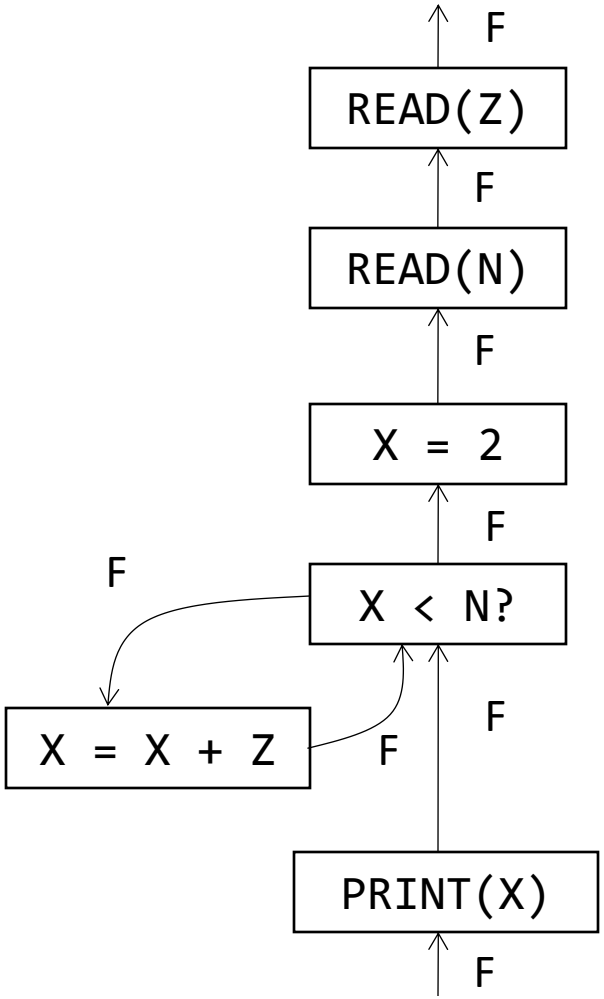
At a program point can the liveness information change?

- Yes, Liveness information changes from false to true and not otherwise.

Recap: Liveness

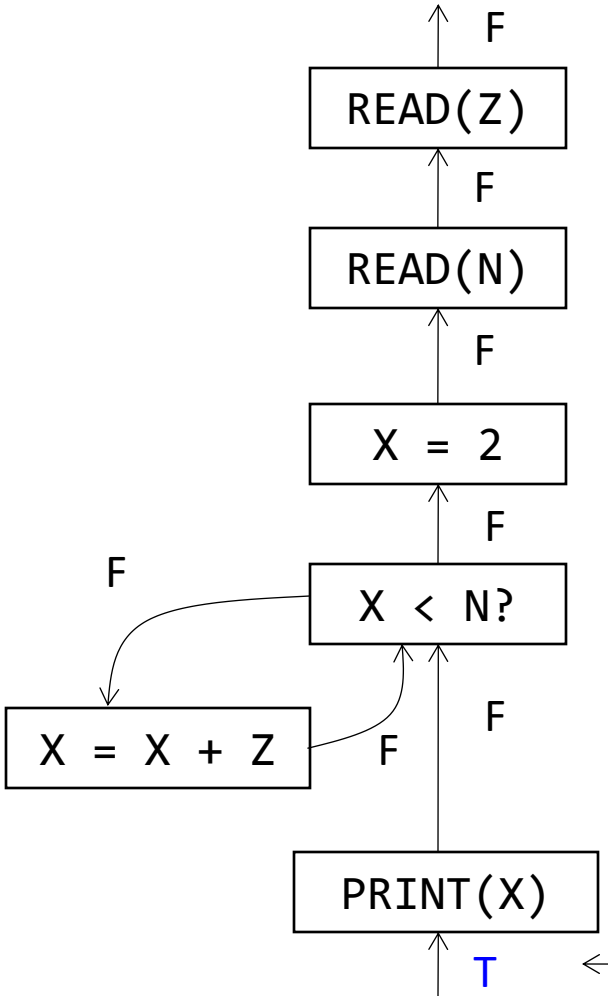


Original CFG



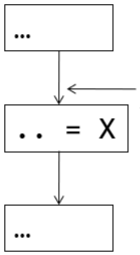
CFG with edges reversed (and initialized) for backwards analysis: is X live? (F=false, T=true)

Recap: Liveness



X must be live here (why?)

Liveness in a CFG



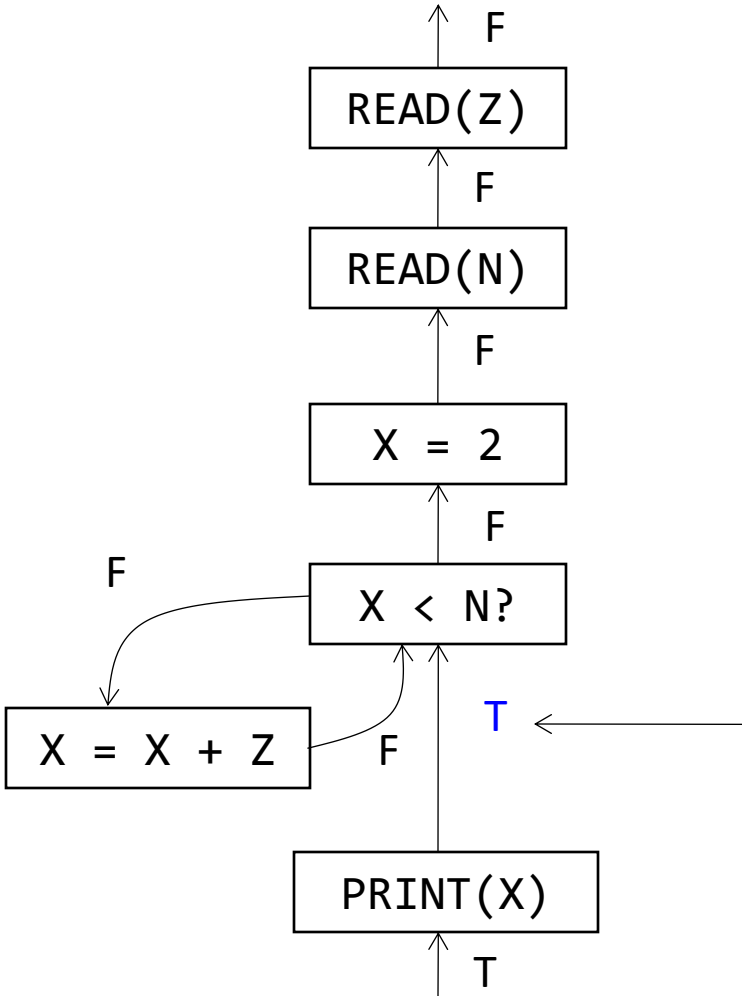
X *must be* live here (i.e. before the statement)

• Define a set `LiveUse(b)`, where b is a basic block, as the set of all variables that are used within block b. `LiveIn(b)` ⊇ `LiveUse(b)`

CS406, IIT Dharwad 40

Recall: LiveIn(b) = LiveUse(b) ∪ (LiveOut(b) – Def(b))

Recap: Liveness



X must be live here (why?)

Liveness in a CFG

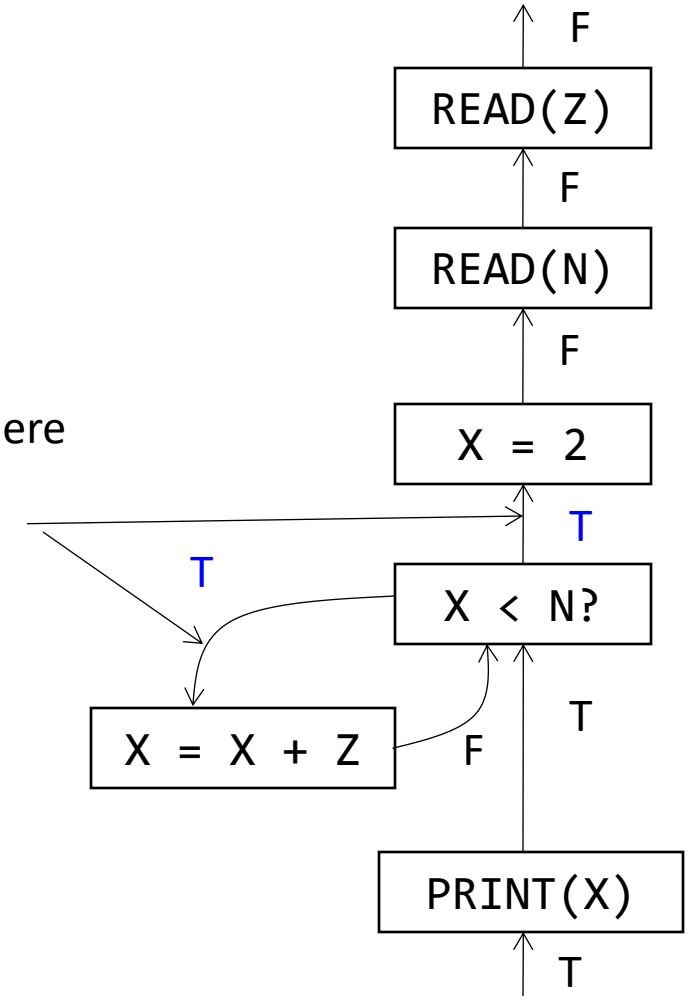
- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block
  - OR the variable is live at exit and not defined within the block

LiveIn(b) = LiveUse(b) U (LiveOut(b) – Def(b))

CS406, IIT Dharwad 45

Recap: Liveness

X must be live here (why?)



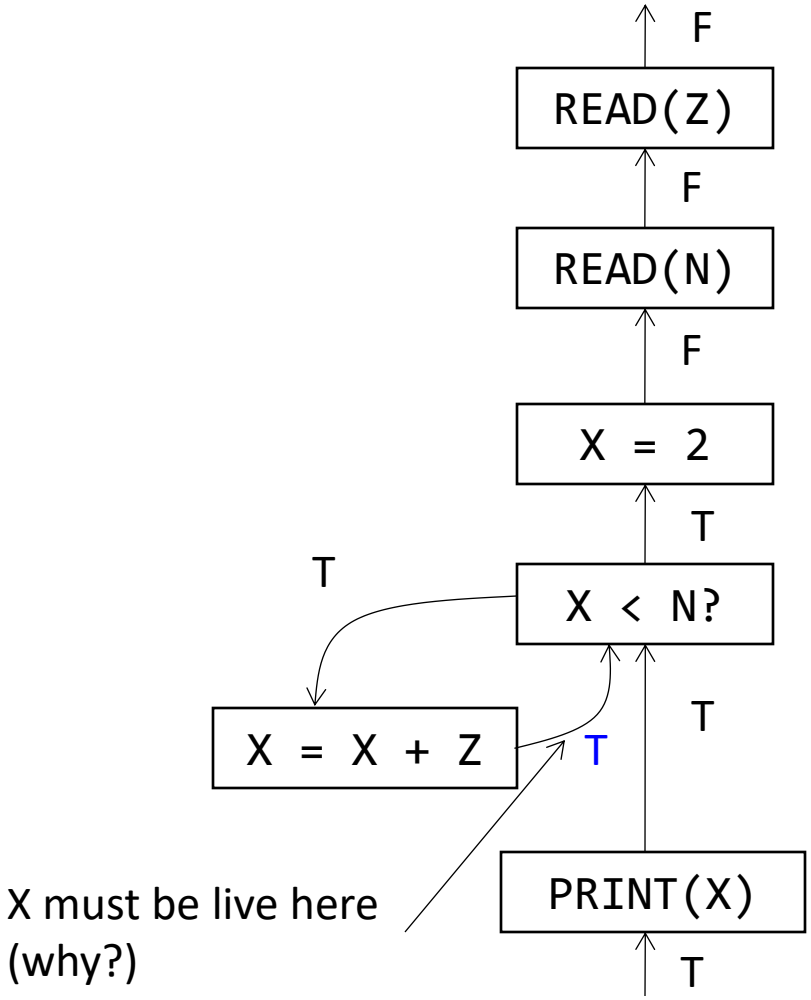
Liveness in a CFG

- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block
  - OR the variable is live at exit and not defined within the block

LiveIn(b) = LiveUse(b) U (LiveOut(b) – Def(b))

CS406, IIT Dharwad 45

Recap: Liveness

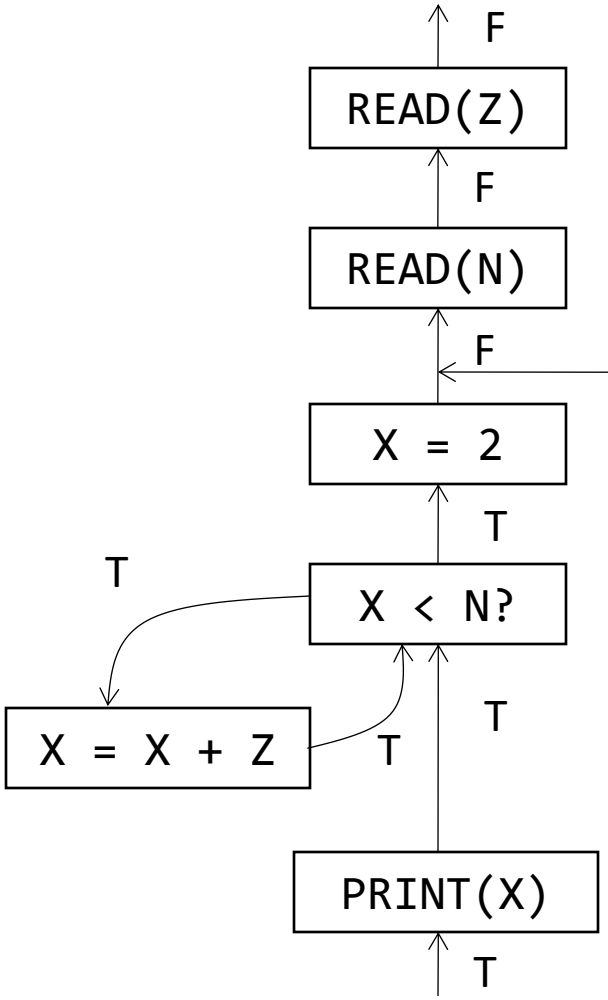


Liveness in a CFG

- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block
  - OR the variable is live at exit and not defined within the block

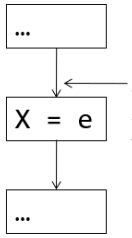
LiveIn(b) = LiveUse(b) U (LiveOut(b) – Def(b))

Recap: Liveness



X dead here (why?). No change in information.

Liveness in a CFG

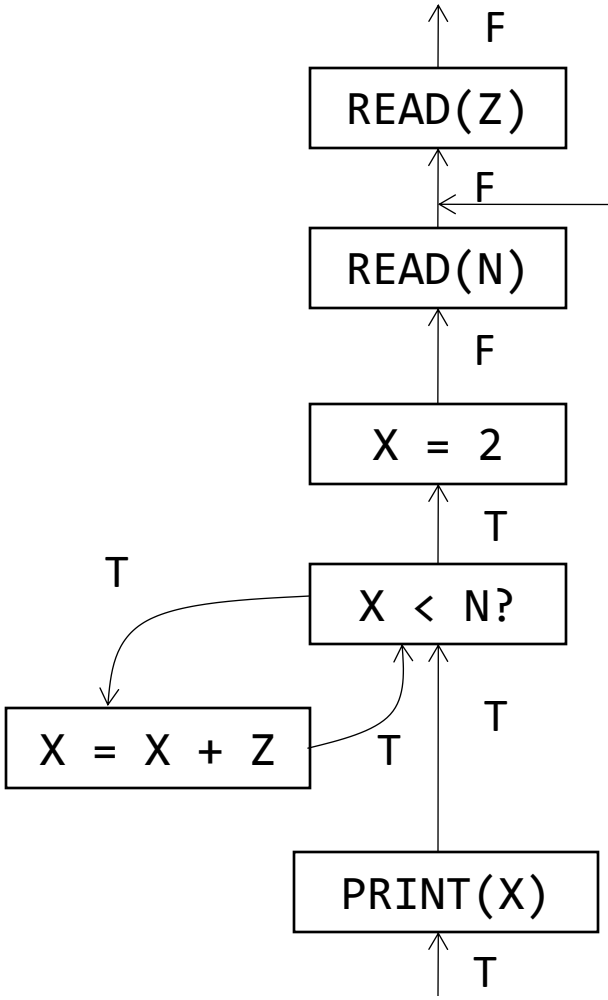


Given that e does not use X, X is *definitely dead* here (i.e. before the statement).

- Define a set `LiveIn(b)`, where b is a basic block, as: the set of all variables live at the entrance of a basic block

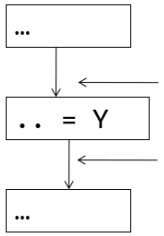
CS406, IIT Dharwad 36

Recap: Liveness



X dead here (why?). No change in information.

Liveness in a CFG - Observation



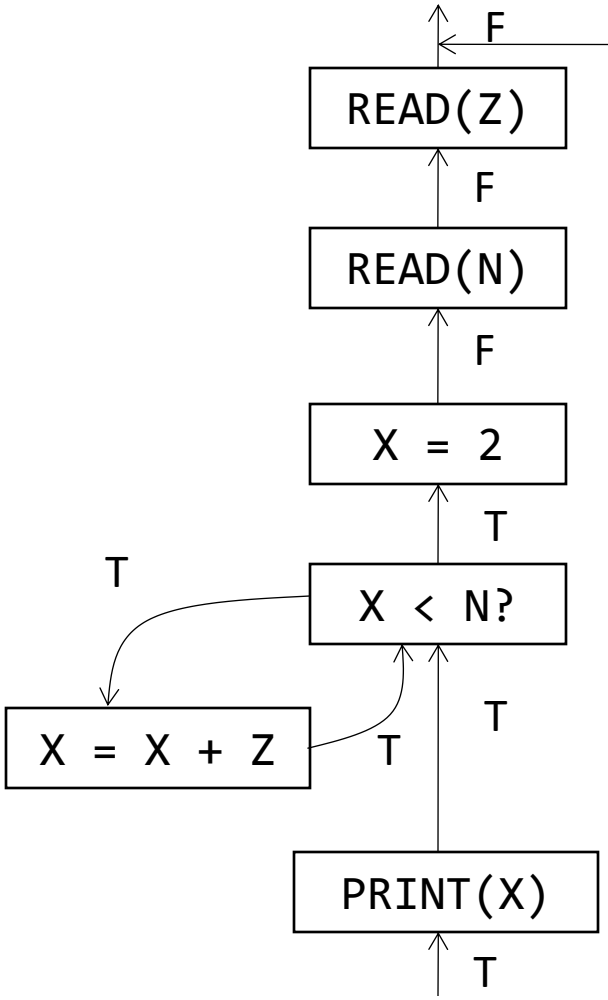
X not live here / X is live here

If X is not live here / X is live here

•If a node neither uses nor defines X, the liveness property remains the same before and after executing the node

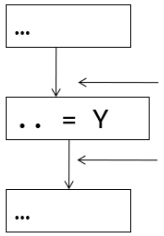
CS406, IIT Dharwad 41

Recap: Liveness



X dead here (why?). No change in information.

Liveness in a CFG - Observation



X not live here / X is live here

If X is not live here / X is live here

•If a node neither uses nor defines X, the liveness property remains the same before and after executing the node

CS406, IIT Dharwad 41

Exercise: Repeat for Z and N

Constant Propagation

- Bigger problem size:
  - Which lines using X could be replaced with a constant value? (apply only constant propagation)
  - How can we automate to find an answer to this question?

```
1. X := 2
2. Label1:
3. Y := X + 1
4. if Z > 8 goto Label2
5. X := 3
6. X := X + 5
7. Y := X + 5
8. X := 2
9. if Z > 10 goto Label1
10.X := 3
11.Label2:
12.Y := X + 2
13.X := 0
14.goto Label3
15.X := 10
16.X := X + X
17.Label3:
18.Y := X + 1
```

Constant Propagation

- Problem statement:
  - Replace use of a variable X by a constant K

- Requirement:
  - **property**: on every path to the use of X, the last assignment to X is: X=K

    Same as: “is X=K at a program point?”

    At any program point where the above property holds, we can apply constant propagation.

How can we find constants?

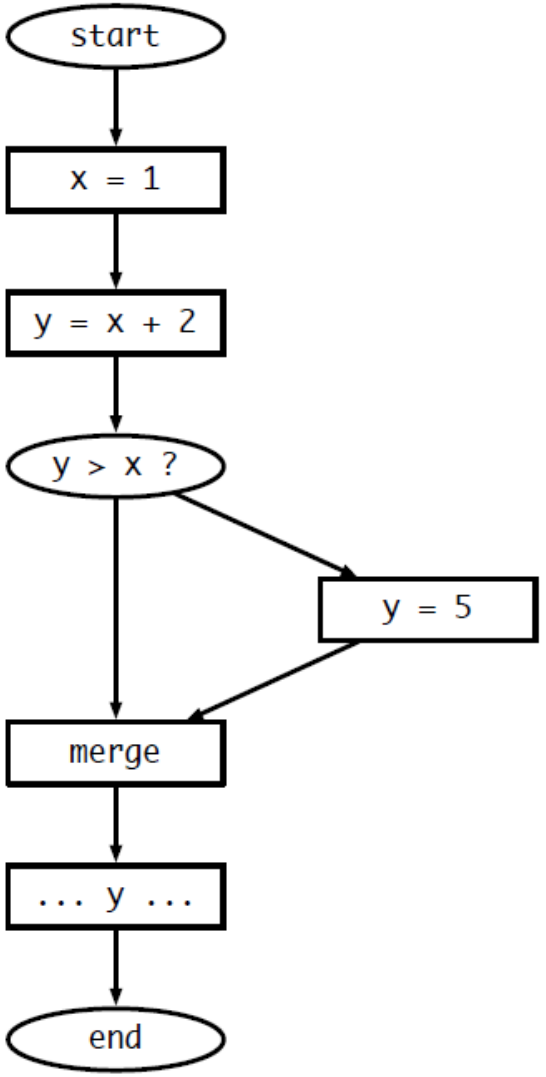
- Ideal: run program and see which variables are constant
  - Problem: variables can be constant with some inputs, not others – need an approach that works for all inputs!
  - Problem: program can run forever (infinite loops?) – need an approach that we know will finish
- Idea: run program *symbolically*
  - Essentially, keep track of whether a variable is constant or not constant (but nothing else)

Overview of algorithm

- Build control flow graph
  - We'll use statement-level CFG (with merge nodes) for this
- Perform symbolic evaluation
  - Keep track of whether variables are constant or not
- Replace constant-valued variable uses with their values, try to simplify expressions and control flow

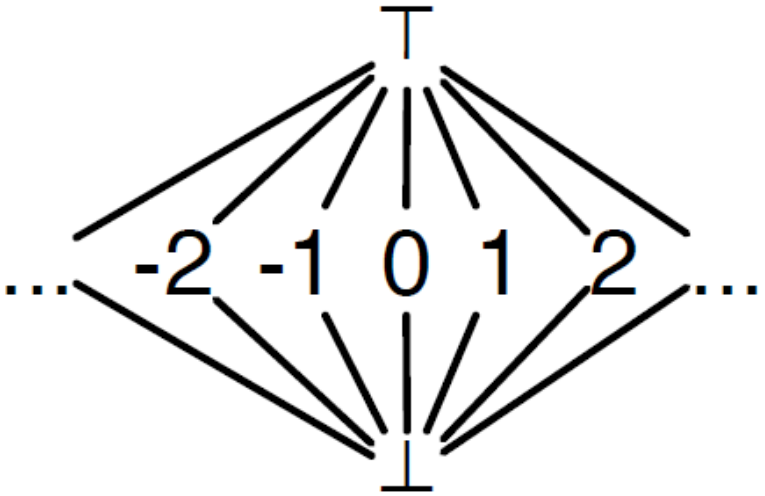
Build CFG

```
x = 1;
y = x + 2;
if (y > x) then y = 5;
... y ...
```



Symbolic evaluation

- Idea: replace each value with a symbol
  - constant (specify which), no information, definitely not constant
- Can organize these possible values in a *lattice*
  - Set of possible values, arranged from least information to most information



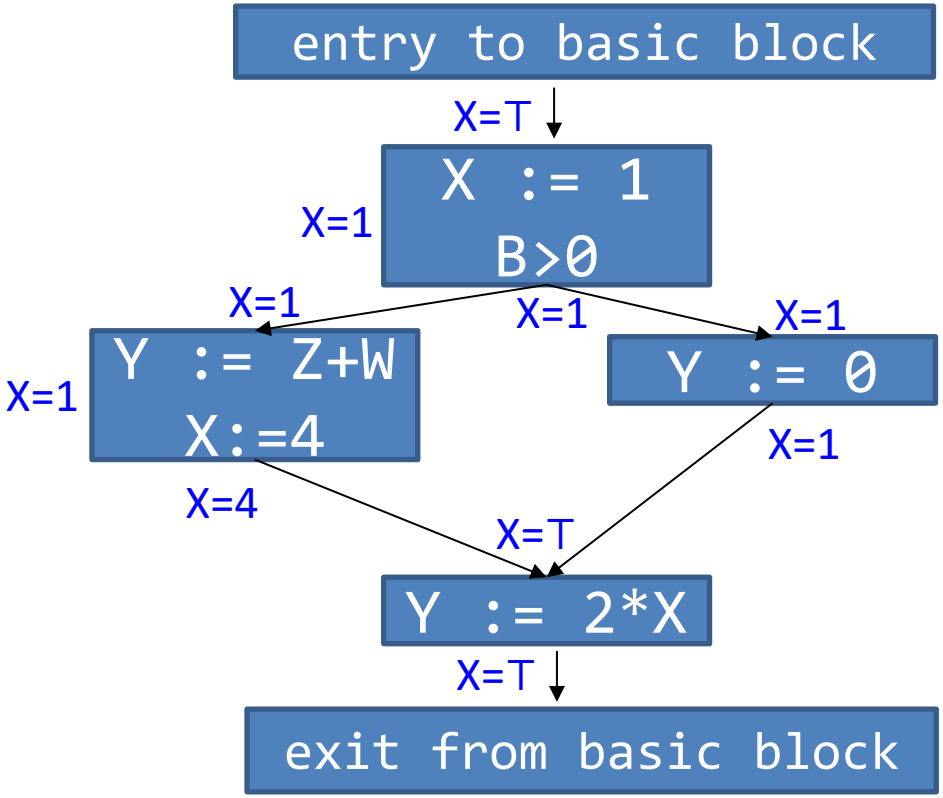
Symbolic Evaluation

- Associate with X one of the following values:

Value	Meaning
⊥ (“bottom”)	This statement never executes
K (“constant”)	X = K
⊤ (“top”)	X is not a constant

- Idea of symbolic execution: at all program points, determine the value of X

Constant Propagation



If X=K at some program point, we can apply constant propagation (replace the use of X with value of K at that program point)

Constant Propagation

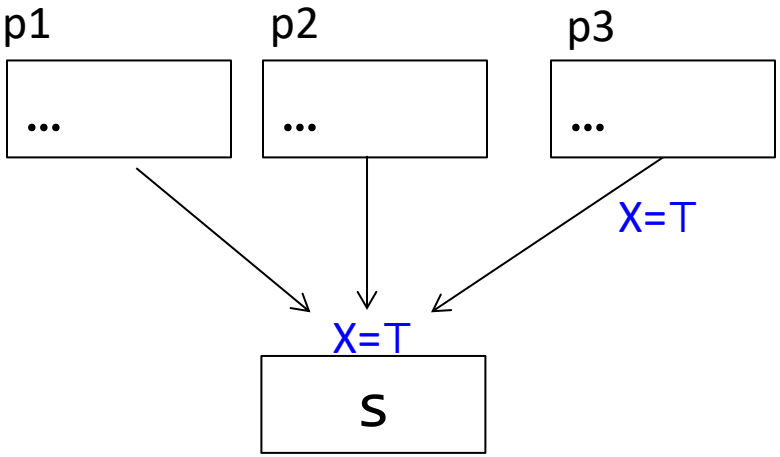
- Determining the value of X at program points:
  - Just like in Liveness Computation in a CFG, the information required for constant propagation flows from one statement to adjacent statement
  - For each statement `s`, compute the information just before and after `s`. `C` is the function that computes the information:

```
C(X,s,flag)
//if flag=IN, before s what is the value of X
//if flag=OUT, after s what is the value of X
```

    - **Transfer function** (pushes / transfers information from one statement to another)

Constant Propagation

- Determining the value of X at program points (Rule 1):

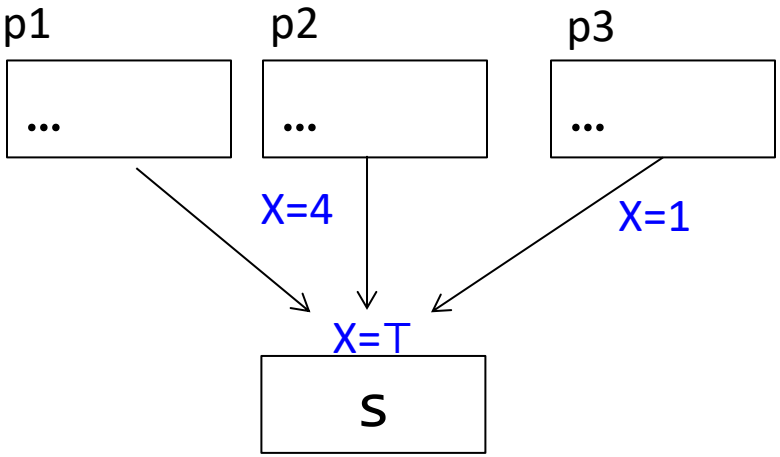


If X=⊤ at exit of *any* of the predecessors, X=⊤ at the entrance of S

if $C(p_i, s, OUT) = \top$
for any i, then $C(X, s, IN) = \top$

Constant Propagation

- Determining the value of X at program points (Rule 2):

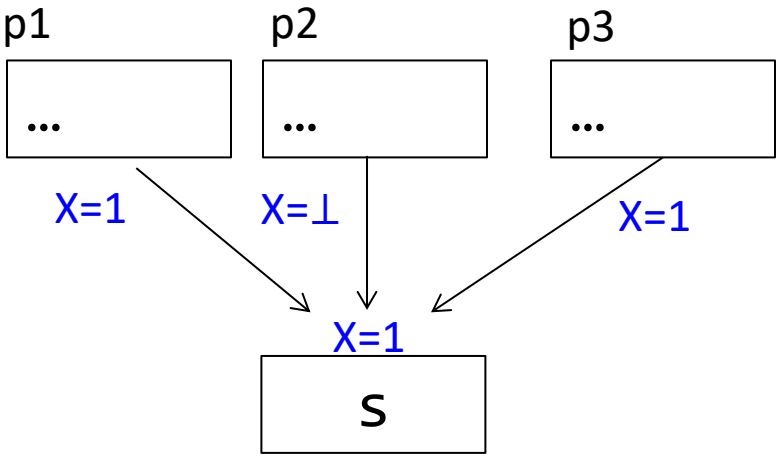


If X=K1 at one predecessor and X=K2 at another predecessor and K1 ≠ K2, then X=T at the entrance of S

if $C(p_i,s,OUT)=K1$ and $C(p_j,s,OUT)=K2$ and $K1 \neq K2$ then $C(X,s,IN)=T$

Constant Propagation

- Determining the value of X at program points (Rule 3):

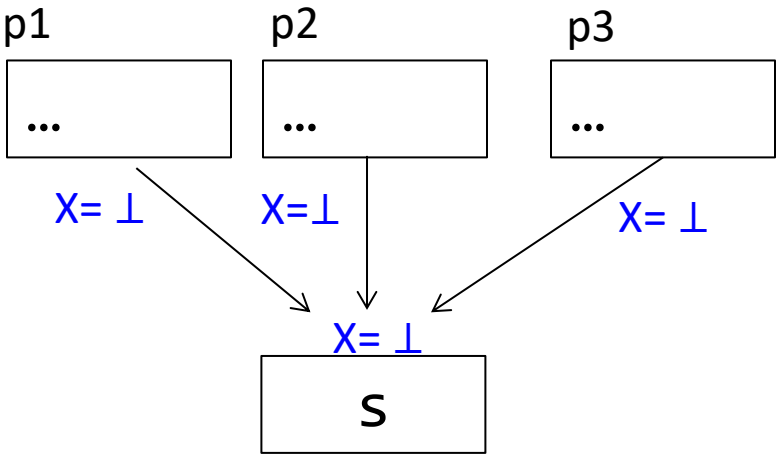


If X=K at some of the predecessors and X= ⊥ at all other predecessors, then X=K at the entrance of S

if C(p_i,s,OUT)=K or ⊥ for all i then C(X,s,IN)= K

Constant Propagation

- Determining the value of X at program points (Rule 4):

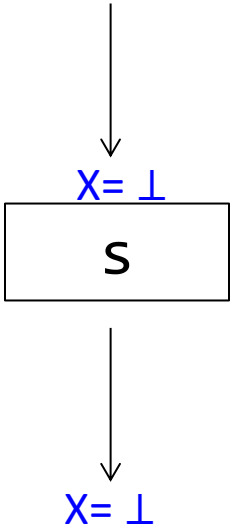


If X= ⊥ at all predecessors, then X= ⊥ at the entrance of S

if $C(p_i,s,OUT)=\perp$ for all i then $C(X,s,IN)= \perp$

Constant Propagation

- Determining the value of X at program points (Rule 5):

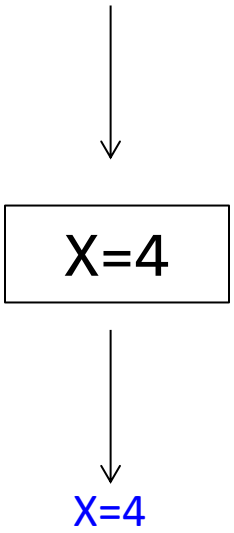


If X= ⊥ at entrance of s, then X= ⊥ at the exit of S

`if C(X,s,IN)=⊥ then C(X,s,OUT)= ⊥`

Constant Propagation

- Determining the value of X at program points (Rule 6):



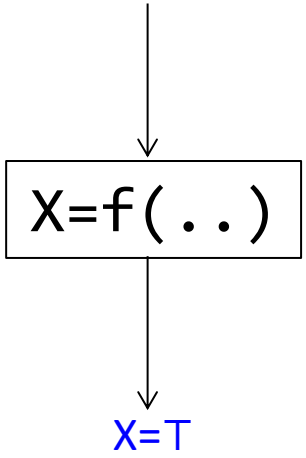
No matter what the value of X is at entrance of `s(X:=K)`, X=K at the exit of s

`C(X,s(X:=K),OUT)=K`

But previous slide said `if C(X,s,IN)=⊥` then `C(X,s,OUT)= ⊥`. *So, we give priority to this.*

Constant Propagation

- Determining the value of X at program points (Rule 7):



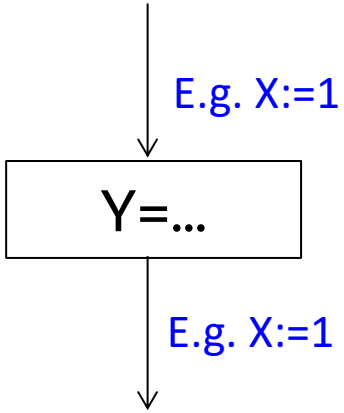
In s, assignment to X is any complicated expression (not a constant assignment).

$$C(X,s(X:=f()),OUT)=\top$$

But earlier slide said if $C(X,s,IN)=\perp$ then $C(X,s,OUT)=\perp$. *So, we give priority to this.*

Constant Propagation

- Determining the value of X at program points (Rule 8):



Value of X remains unchanged before and after s(Y:=..) when s doesn't assign to X and X ≠ Y

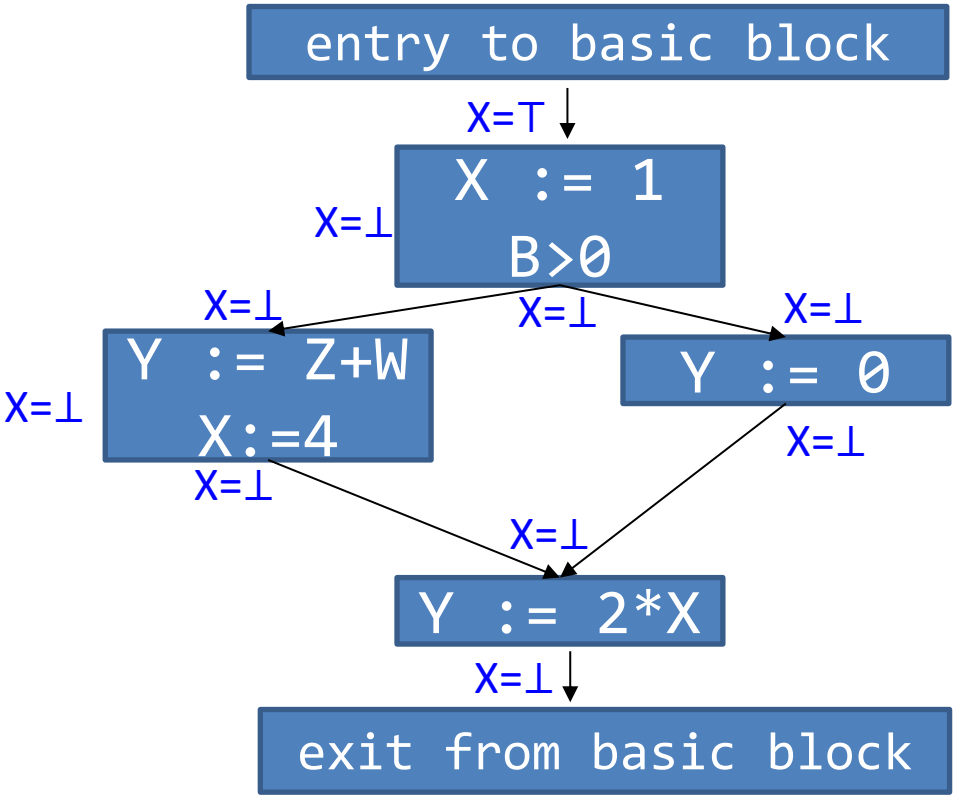
```
C(X,s(Y:=..),OUT)=C(X,s(Y:=..),IN)
```

Constant Propagation

- Putting it all together
  1. For entry s in the program, initialize C(X,s,IN)=⊤ and initialize C(X,s,IN)=C(X,s,OUT)=⊥ everywhere else
  2. Repeat until all program points (i.e. any s) satisfy rules 1-8
     1. Pick s in the CFG that doesn't satisfy any one of rules 1-8 and update information.

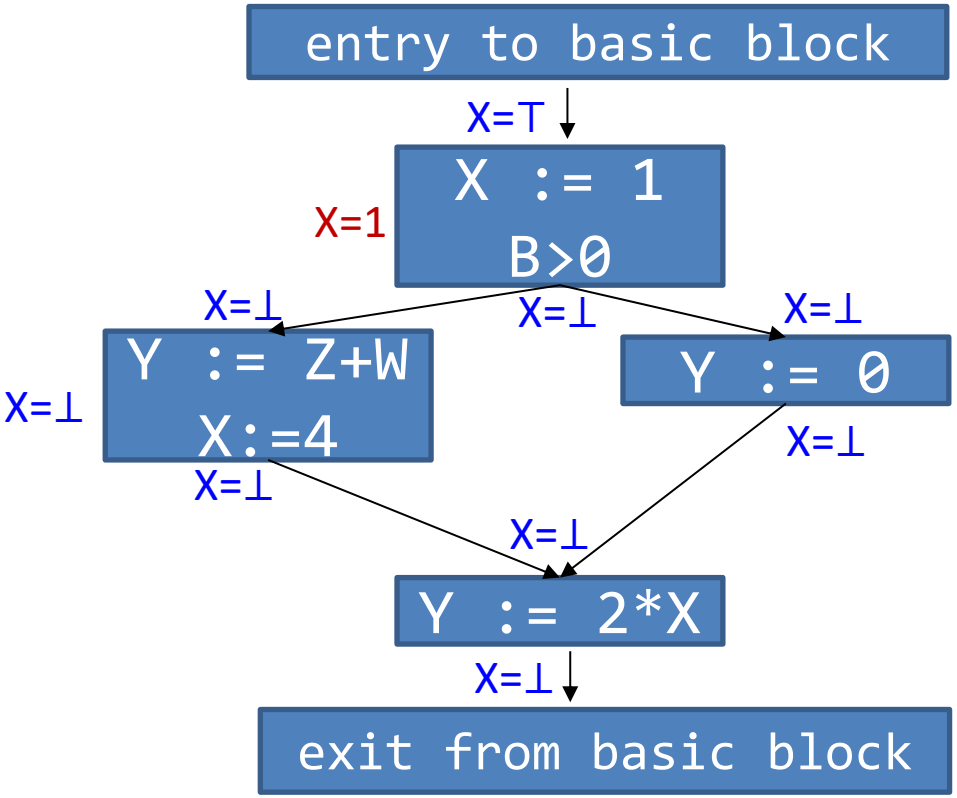
Constant Propagation

- Putting it all together



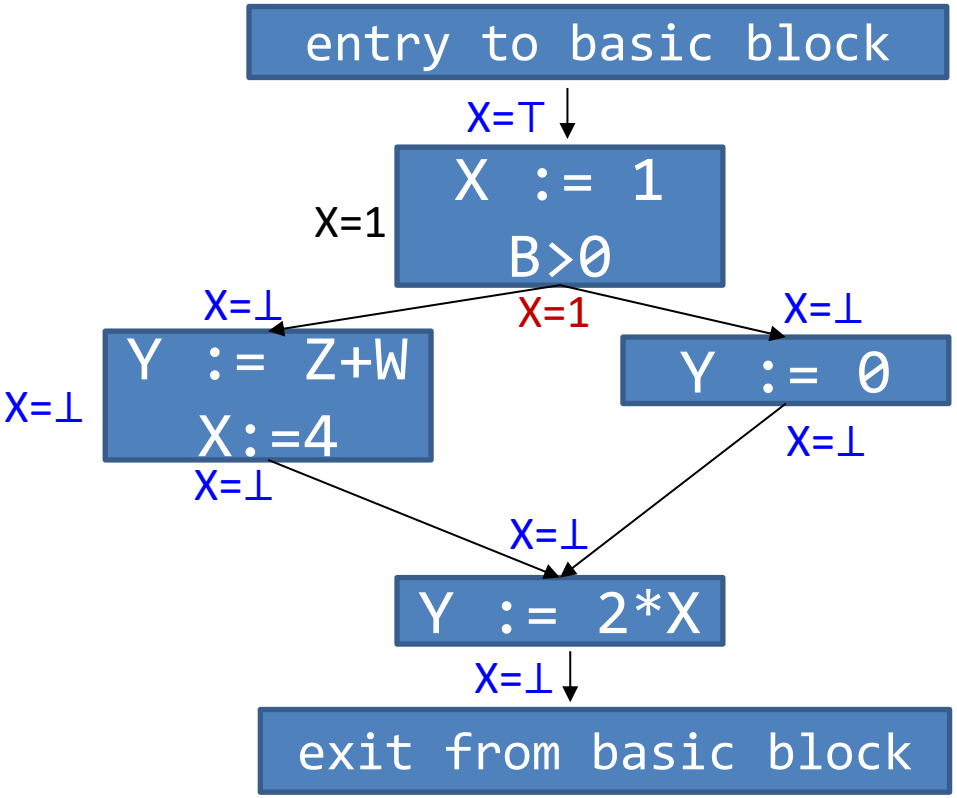
Constant Propagation

- Putting it all together



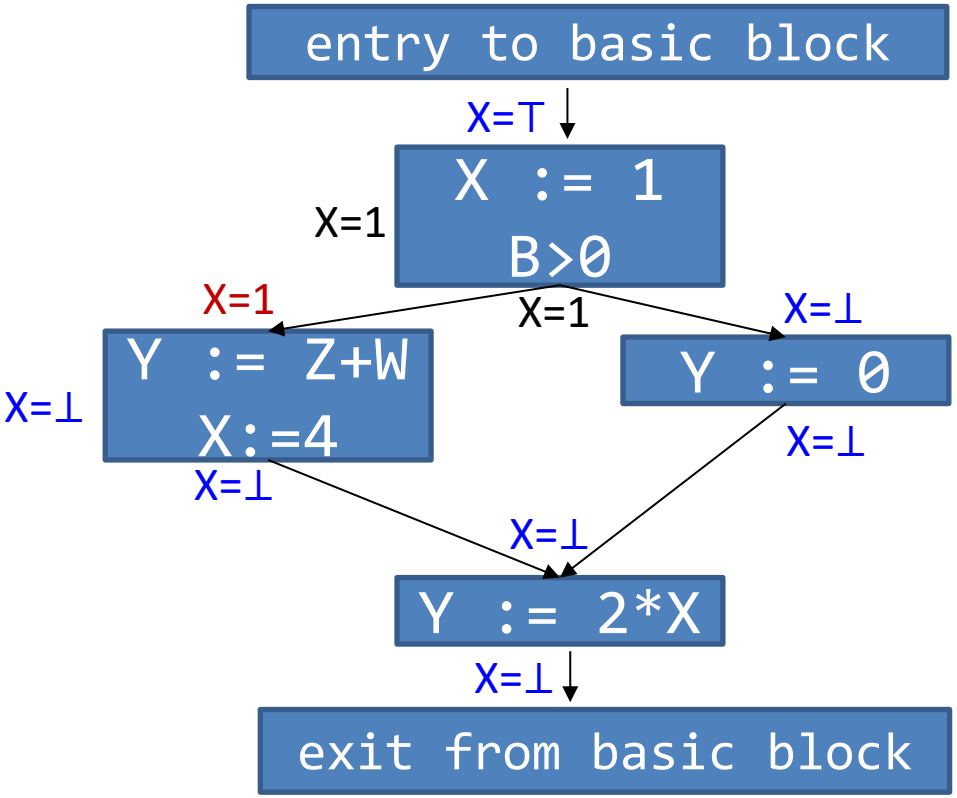
Constant Propagation

- Putting it all together



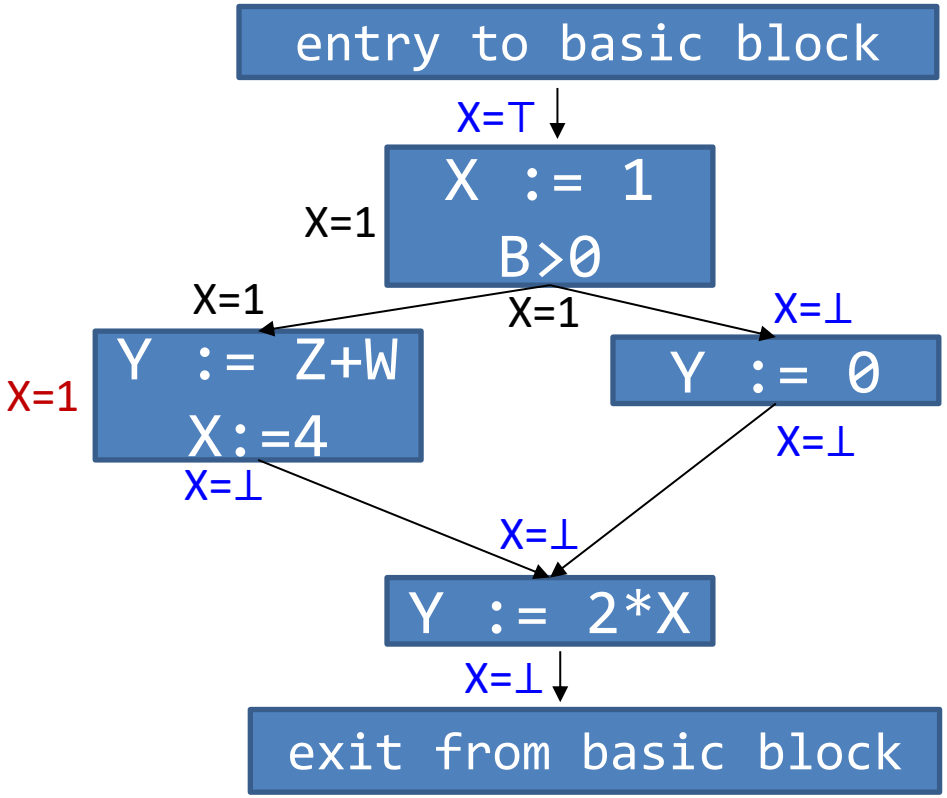
Constant Propagation

- Putting it all together



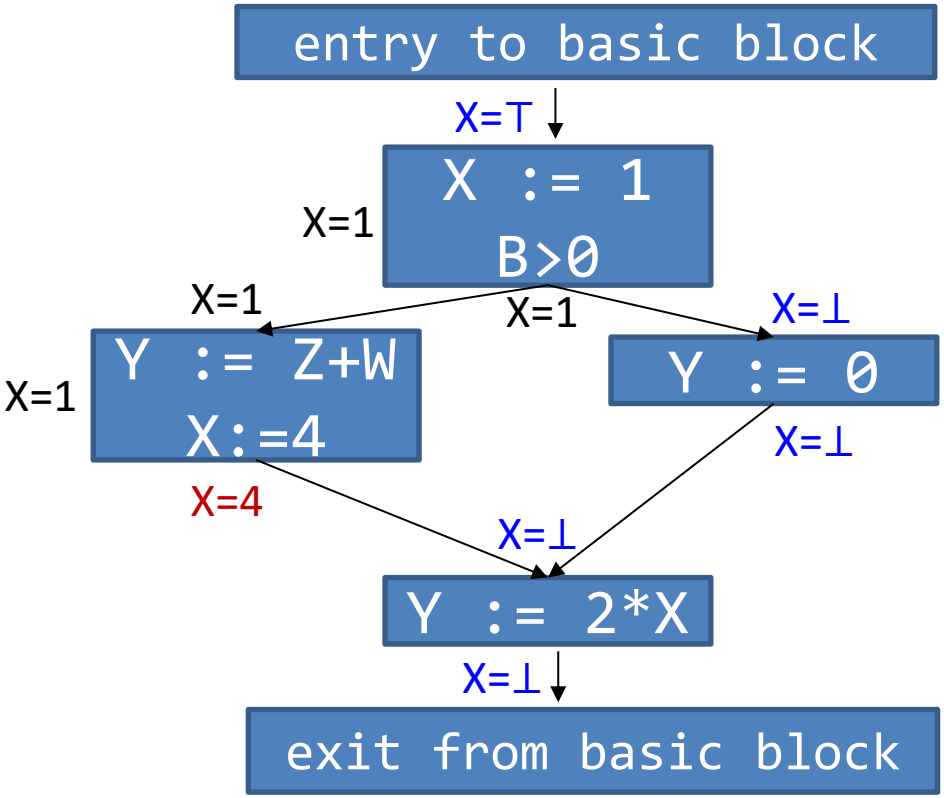
Constant Propagation

- Putting it all together



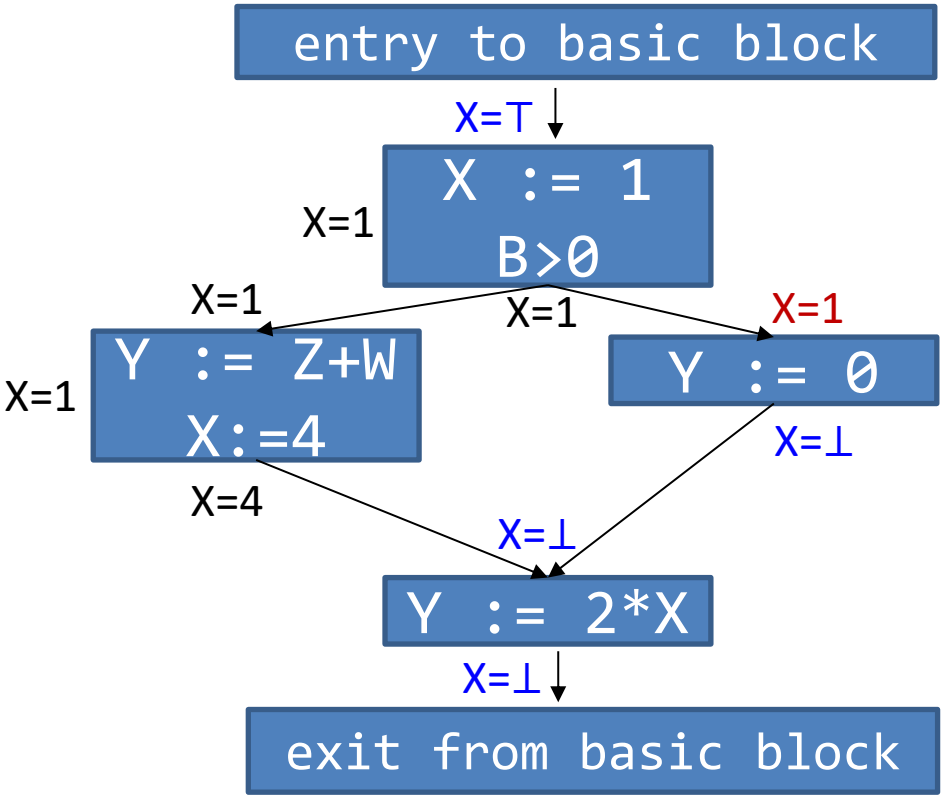
Constant Propagation

- Putting it all together



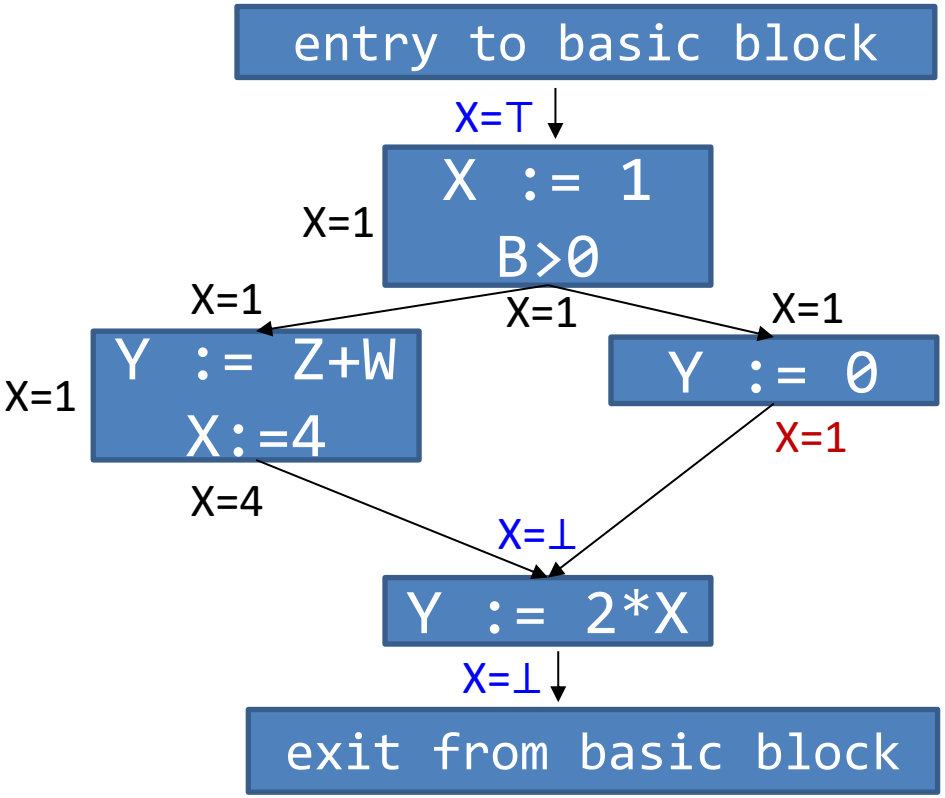
Constant Propagation

- Putting it all together



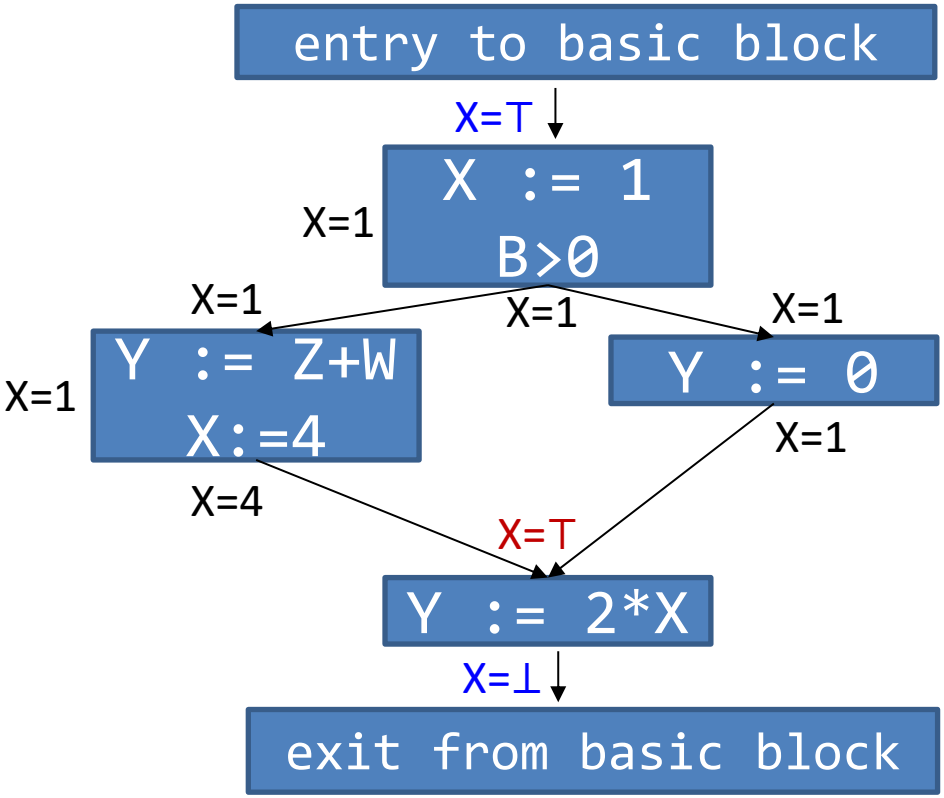
Constant Propagation

- Putting it all together



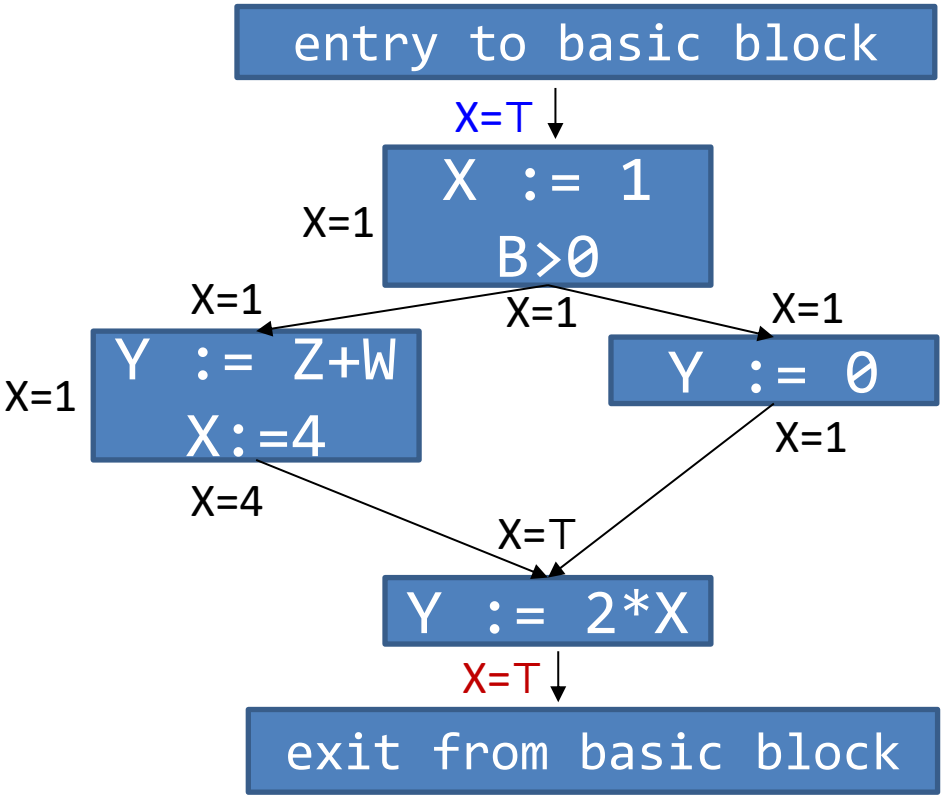
Constant Propagation

- Putting it all together

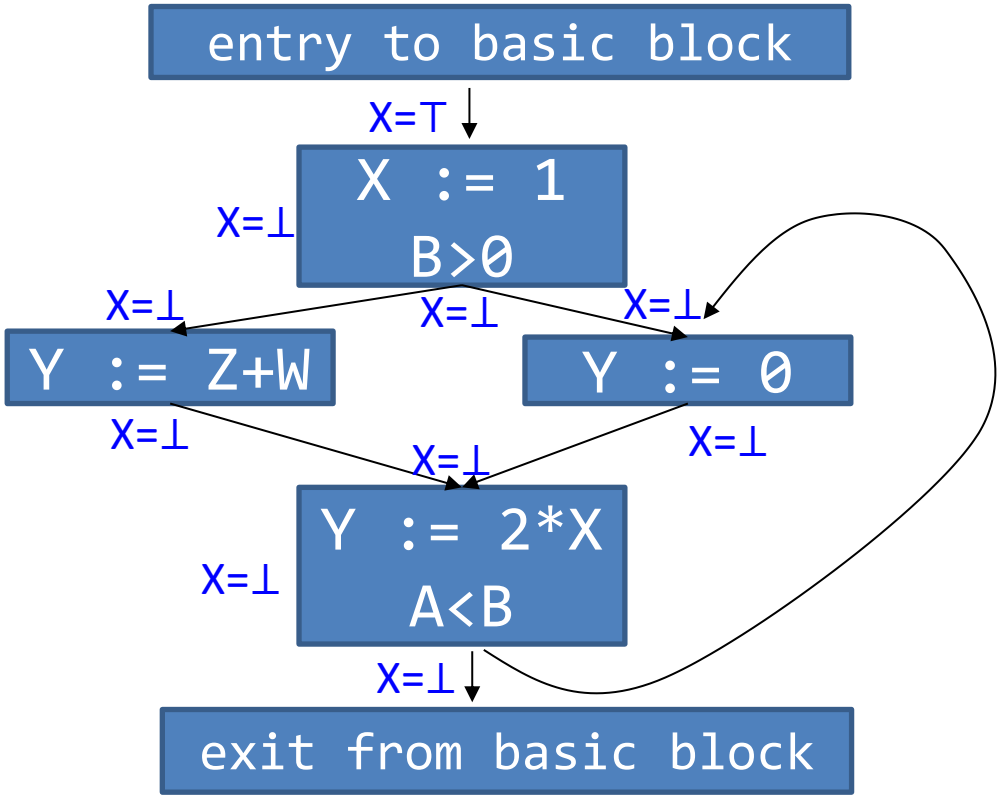


Constant Propagation

- Putting it all together



Constant Propagation - Loops



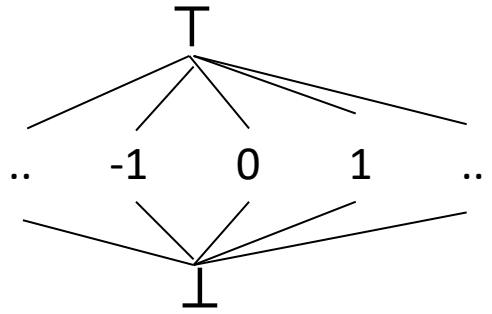
Ordering of information: Generalizing

- We have been executing with symbols $\bot$, $\top$, and K. These are called *abstract values*
- Order these values as:

$$\bot < K < \top$$

Can also be thought of as an ordering from least information to most information

Pictorially:



Ordering of information: Generalizing

- Least Upper Bound (lub) : smallest element (abstract value) that is greater than or equal to values in the input
  - E.g. $\text{lub}(\bot, \bot) = \bot$, $\text{lub}(\top, \bot) = \top$, $\text{lub}(-1, 1) = \top$, $\text{lub}(1\ \ \bot) = ?$
  - Rewriting rules 1-4: `C(X,s,IN)=lub{C(p_i,s,OUT) for all predecessors i)}`
  - Also called as join operator. Written as: A ⊔ B

Ordering of information: Generalizing

- Recall that in determining information at all program points:

  "2. Repeat until all program points (i.e. any s) satisfy rules 1-8

  - Pick s in the CFG that doesn't satisfy any one of rules 1-8 and update information. "

  – How do we know that this terminates?

    - lub ensures that the information changes from lower value to higher value
    - In the constant propagation algorithm:
      – $\perp$ can change to constant and then to $\top$
      – $\perp$ can change to $\top$
      – `C(X, s, flag)` can change at most twice

Constant Propagation

- Exercise: what is the complexity of our constant propagation algorithm?

= NumS* 4 (NumS = number of statements in the program).

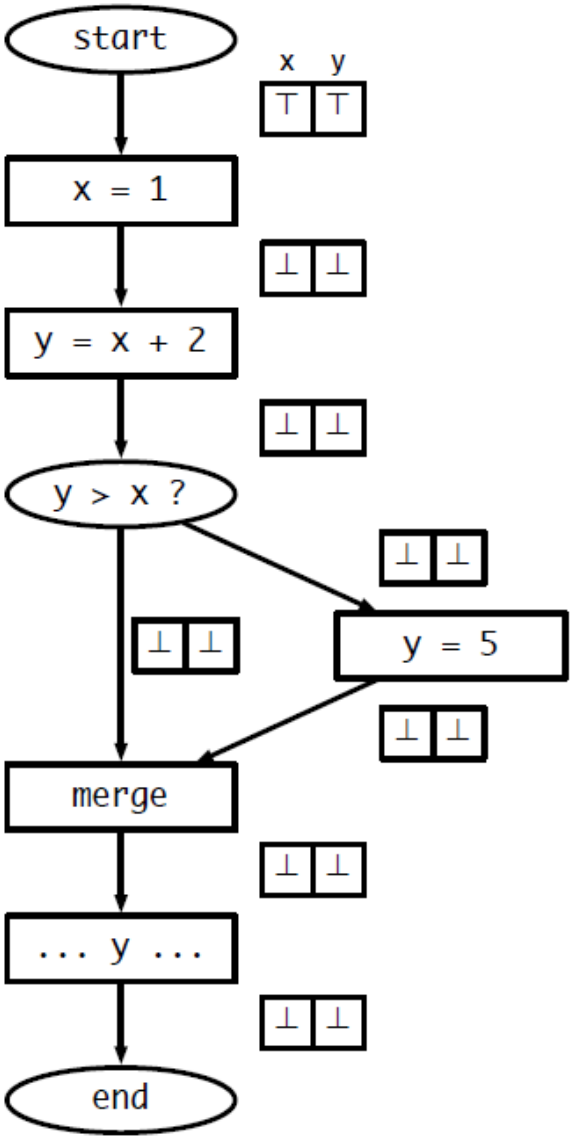
- Per program point, we evaluate the C function.
- The C function changes value at most two times (initialized to $\perp$ first and then could change to K and then to $\top$).
- There are two program points (entry/IN and exit/OUT) for every statement.

This is the complexity of the analysis per variable

How do we do the analysis considering all variables that exist in the program?

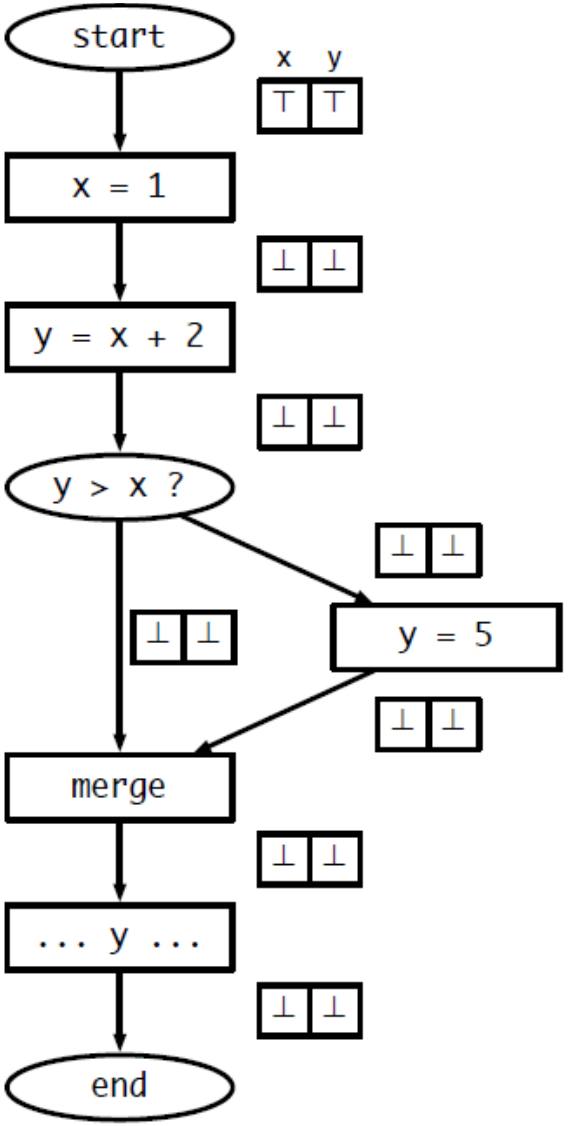
Constant Propagation (Multiple Variables)

- Keep track of the symbolic value of a variable at every program point (on every CFG edge)
  - State vector V
- What should our initial value be?
  - Starting state vector is all $\top$
    - Can't make any assumptions about inputs – must assume not constant
  - Everything else starts as $\bot$, since we have no information about the variable at that point



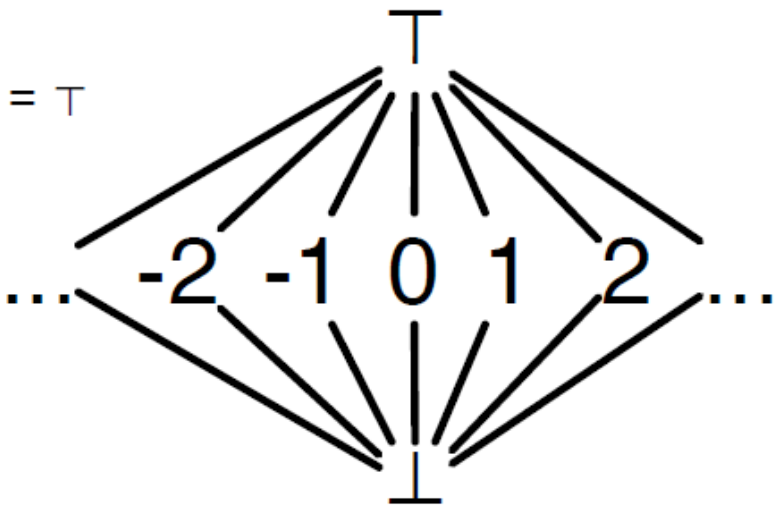
Constant Propagation (Multiple Variables)

- For each statement `t = e` evaluate `e` using V_{in}, update value for `t` and propagate state vector to next statement
- What about switches?
  - If `e` is true or false, propagate V_{in} to appropriate branch
  - What if we can't tell?
    - Propagate V_{in} to both branches, and symbolically execute both sides
- What do we do at merges?



Handling merges

- Have two different V_{in}s coming from two different paths
- Goal: want new value for V_{in} to be *safe* (shouldn't generate wrong information), and we don't know which path we actually took
- Consider a single variable. Several situations:
  - $V_1 = \bot, V_2 = * \rightarrow V_{out} = *$
  - V_1 = constant x, V_2 = x $\rightarrow V_{out}$ = x
  - V_1 = constant x, V_2 = constant y $\rightarrow V_{out} = \top$
  - $V_1 = \top, V_2 = * \rightarrow V_{out} = \top$
- Generalization:
  - $V_{out} = V_1 \sqcup V_2$



Result: worklist algorithm

- Associate state vector with each edge of CFG, initialize all values to $\perp$, worklist has just start edge
  - While worklist not empty, do:

Process the next edge from worklist

Symbolically evaluate target node of edge using input state vector

If target node is assignment (x = e), propagate V_{in}[eval(e)/x] to output edge

If target node is branch (e?)

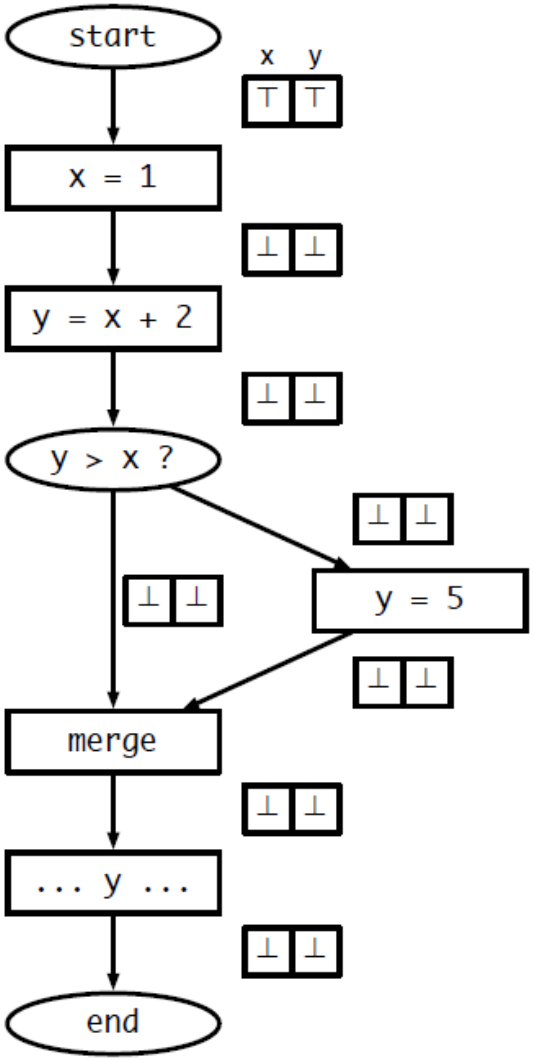
If eval(e) is true or false, propagate V_{in} to appropriate output edge

Else, propagate V_{in} along both output edges

If target node is merge, propagate join(all V_{in}) to output edge

If any output edge state vector has changed, add it to worklist

Running example



Worklist

Running example

