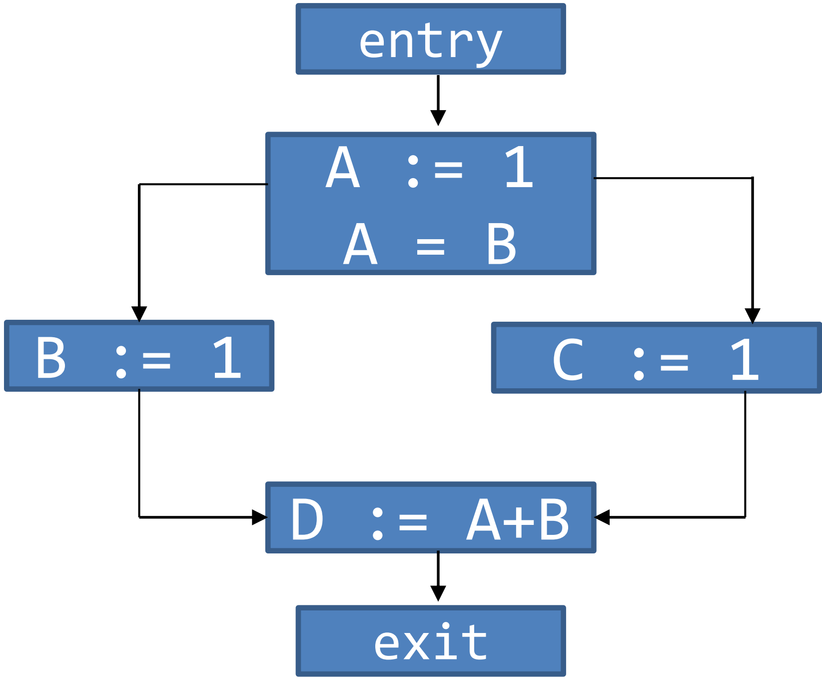
# CS406: Compilers

Spring 2022

## Week 12: Dataflow Analysis – Constant Propagation, Exercises

# Recap: Liveness

- Variables are live if there exists *some path* leading to its use
- Start from exit block and proceed *backwards* against the control flow to compute



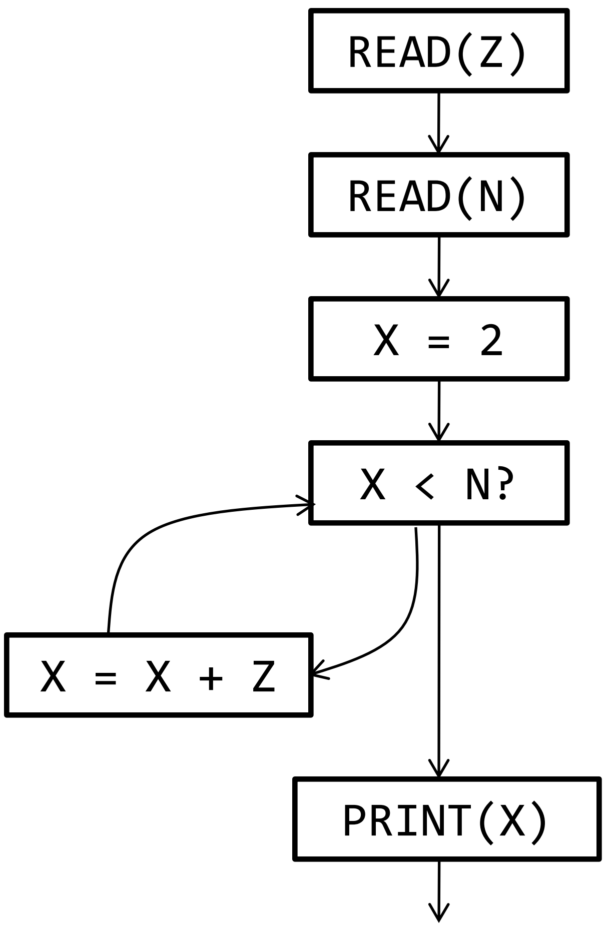

$$\text{LiveOut(b)} = \bigcup_{i \in Succ(b)} \text{LiveIn}(i)$$

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

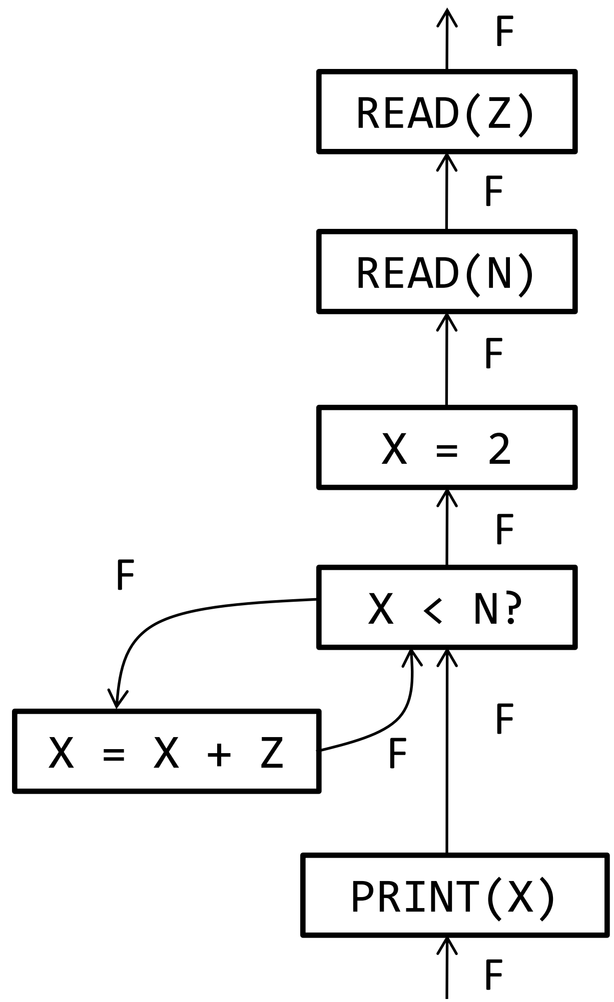
```
//set that contains all variables
used by block b
```

```
//set that contains all
variables defined by block b
```

# Recap: Liveness

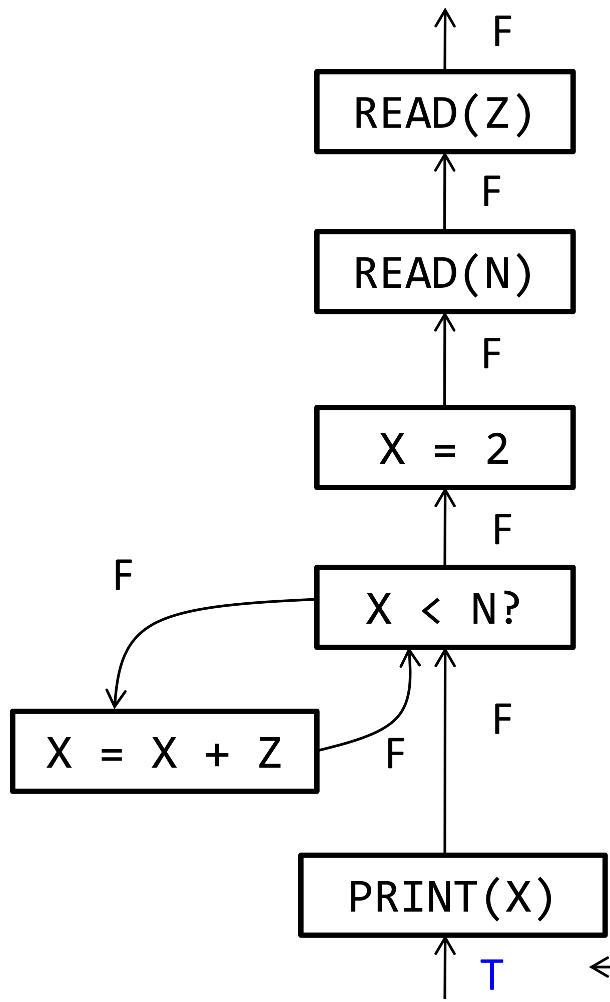



Original CFG



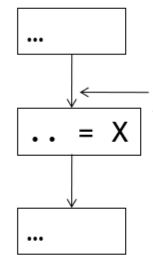

CFG with edges reversed (and initialized) for backwards analysis: is X live? (F=`false`, T=`true`)

# Recap: Liveness




X must be live here
(refer week11 slide)

Liveness in a CFG

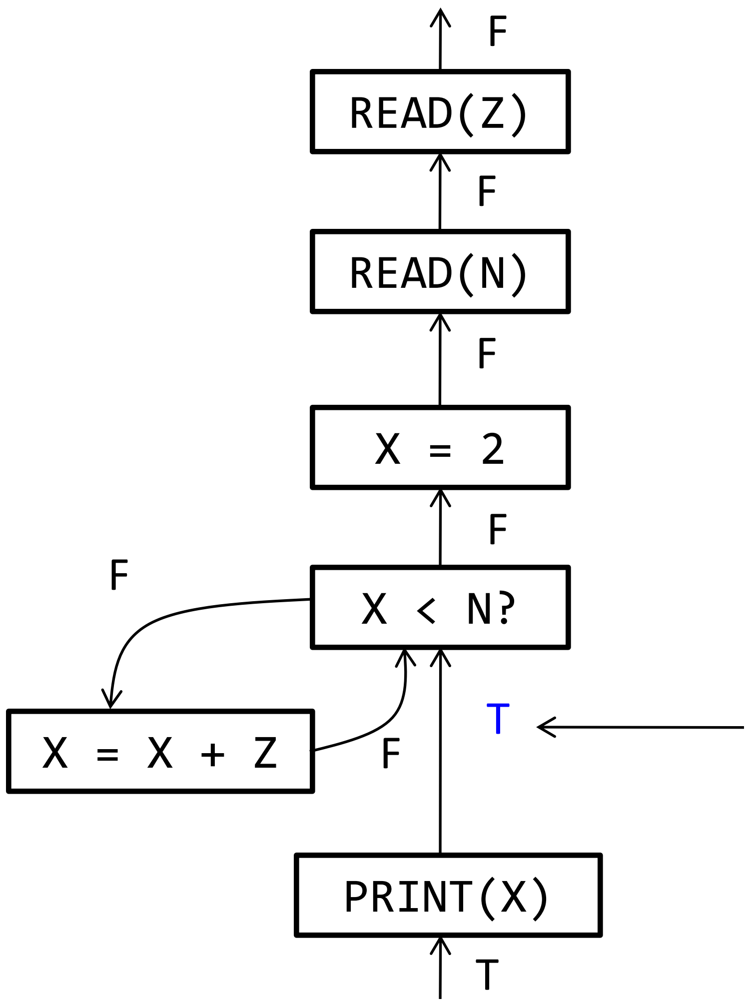



X *must be* live here (i.e. before the statement)

- 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

# Recap: Liveness




X must be live here
(refer week11 slide)

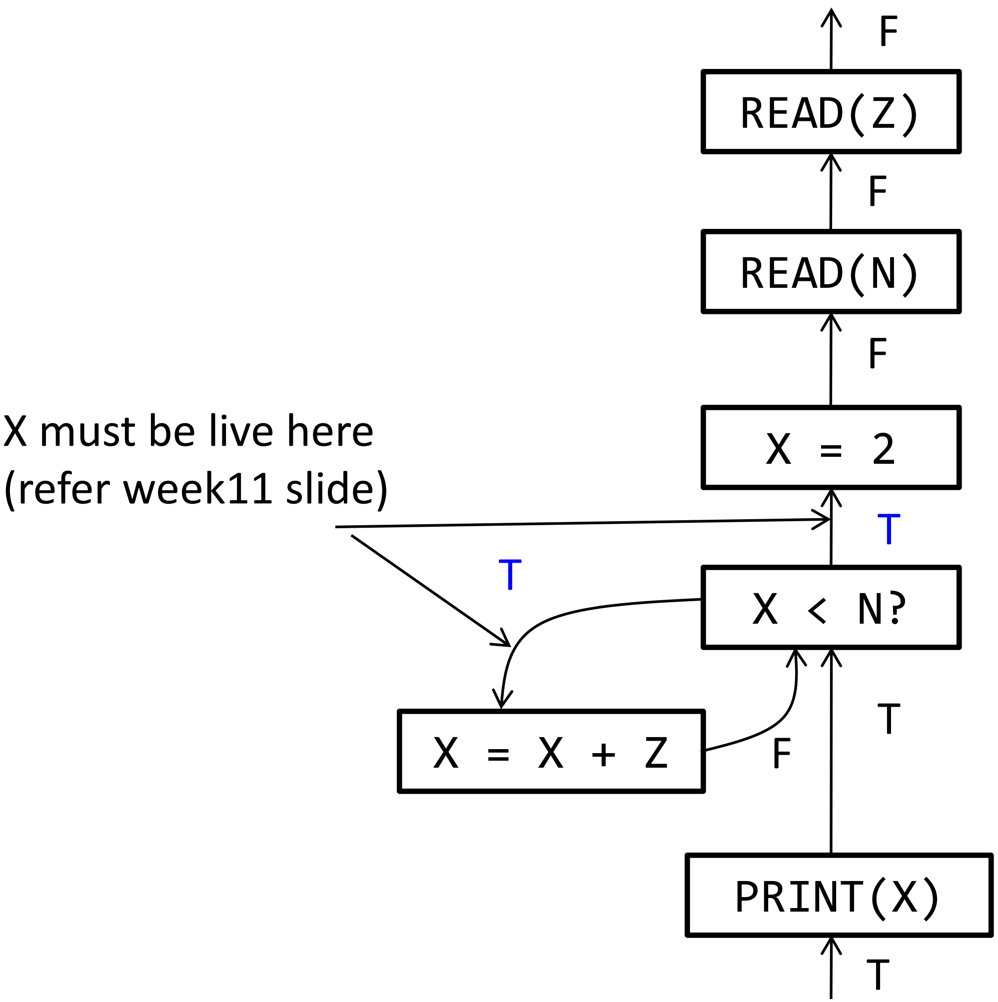
Liveness in a CFG

- 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) ∪ (LiveOut(b) – Def(b))`

CS406, IIT Dharwad 45

# Recap: Liveness



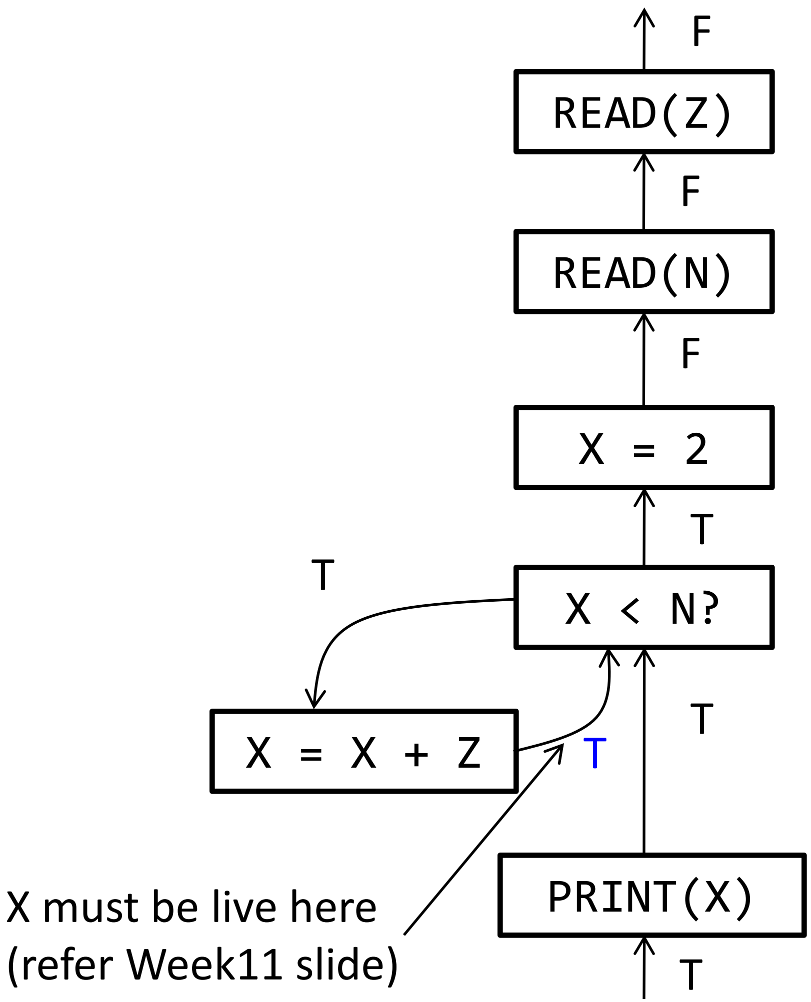

Liveness in a CFG

- 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) ∪ (LiveOut(b) – Def(b))
```

CS406, IIT Dharwad 45

# Recap: Liveness



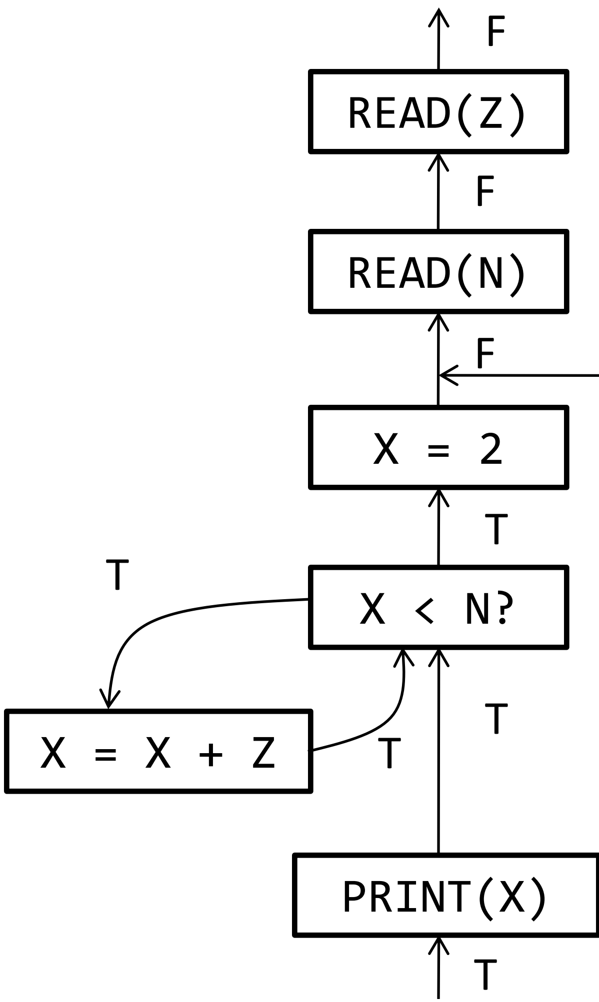

## Liveness in a CFG

- 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

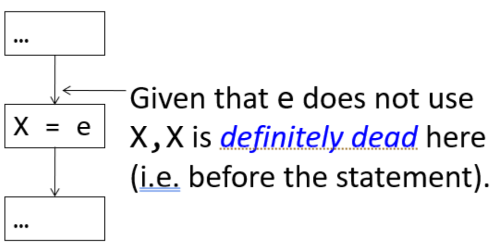
$$LiveIn(b) = LiveUse(b) \cup (LiveOut(b) - Def(b))$$

CS406, IIT Dharwad 45

# Recap: Liveness




X dead here (refer Week11 slide).
No change in information.

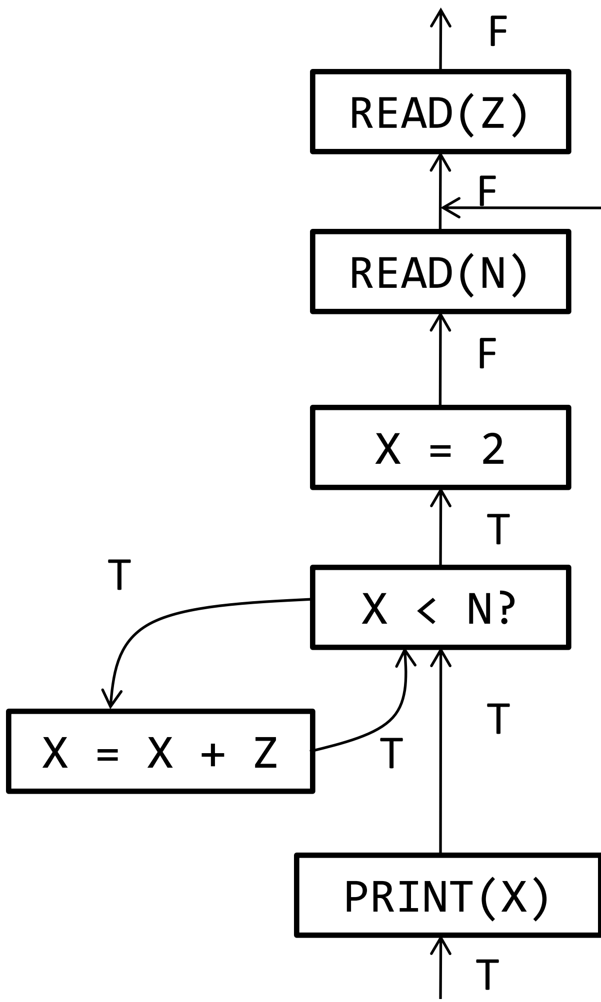

Liveness in a CFG



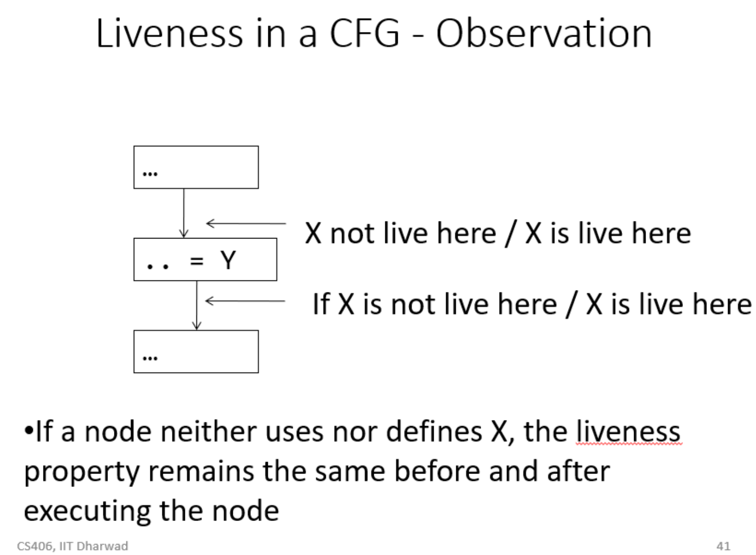
- Define a set `LiveIn(b)`, where b is a basic block, as: the set of all variables live at the entrance of a basic block

CS406, IIT Dharwad 36

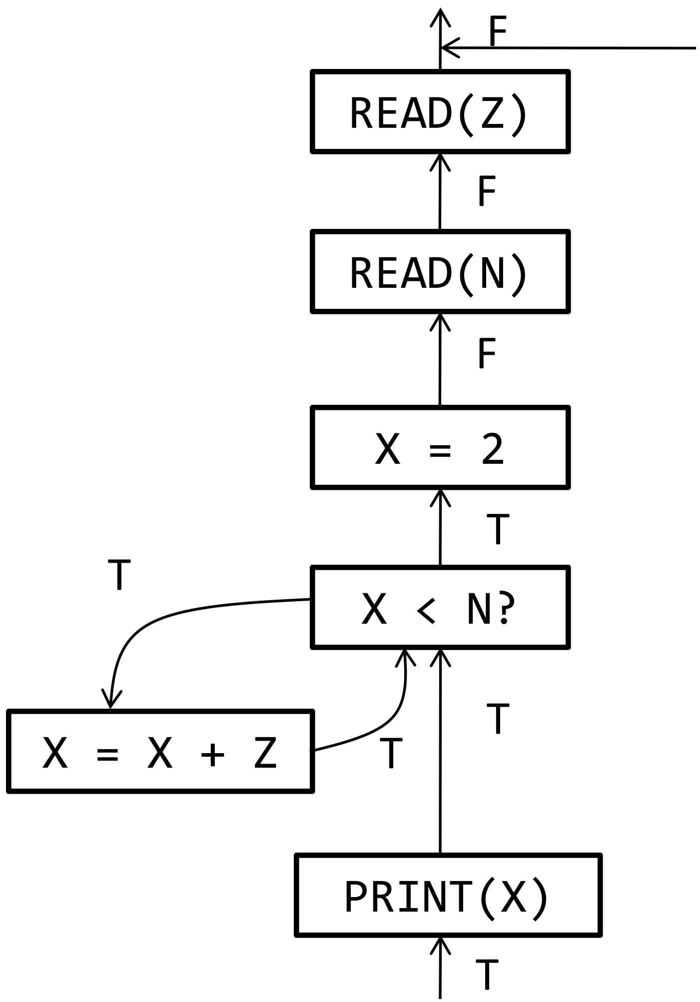
# Recap: Liveness




X dead here (refer Week11 slide).
No change in information.

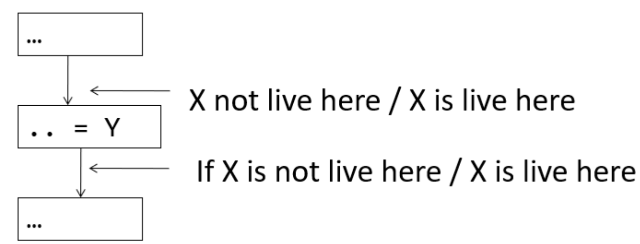


# Recap: Liveness




X dead here (refer Week11 slide). No change in information.

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

CS406, IIT Dharwad 41

*Exercise: Repeat for Z and N*

# Constant Propagation

```
X = 1
Y = X + 2
if( Y> X)
   Y = 5
..
```

⇨

```
X = 1
Y = 1 + 2
if( Y> X)
   Y = 5
..
```

Constant Propagation

⇨

```
X = 1
Y = 3
if( Y> X)
   Y = 5
..
```

Using Constant Propagation, we can optimize further: do constant folding

# Constant Propagation

```
X = 1
Y = X + 2
if( Y> X)
   Y = 5
..
```

⇨

```
X = 1
Y = 3
if( Y> X)
   Y = 5
..
```

Constant Propagation

⇨

~~X = 1~~<br>
~~Y = 3~~ //dead code<br>
if(true)<br>
   Y = 5

Using Liveness information leads to further optimizations: Dead Code Elimination

# 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.

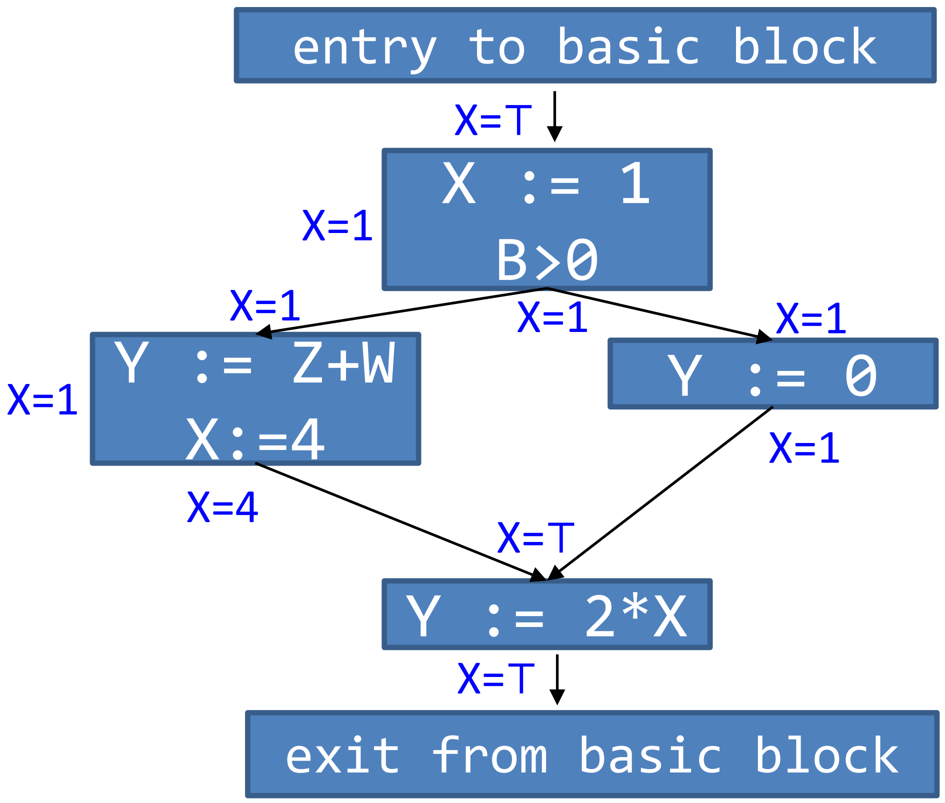
# Constant Propagation

- Associate with X one of the following values:

| Value | Meaning |
|---|---|
| ⊥ ("bottom") | This statement never executes |
| K ("constant") | X = K |
| T ("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

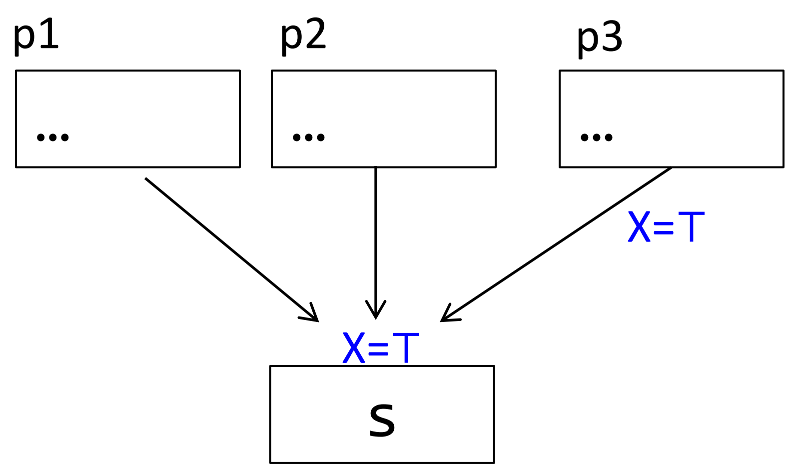
- Determining the value of X at program points:
  - Just like in Liveness Computation in a CFG, the information required for constant propagation flows from one statement to adjacent statement
  - For each statement `s`, compute the information just before and after `s`. C is the function that computes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

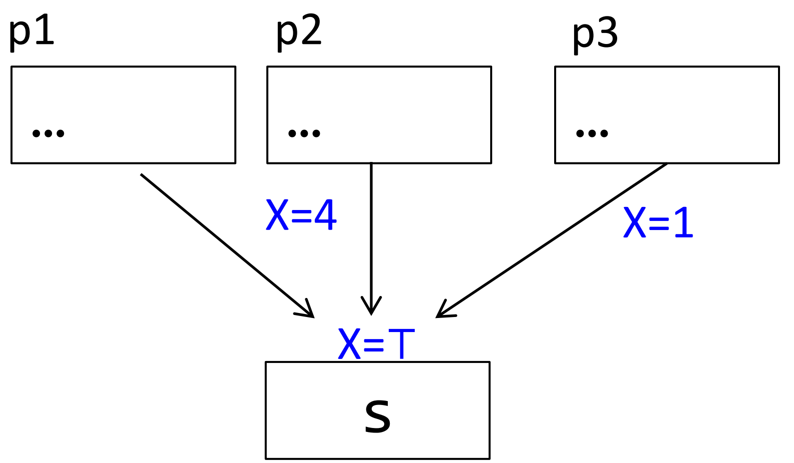



If X=T at exit of *any* of the predecessors, X=T at the entrance of S

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

# Constant Propagation

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

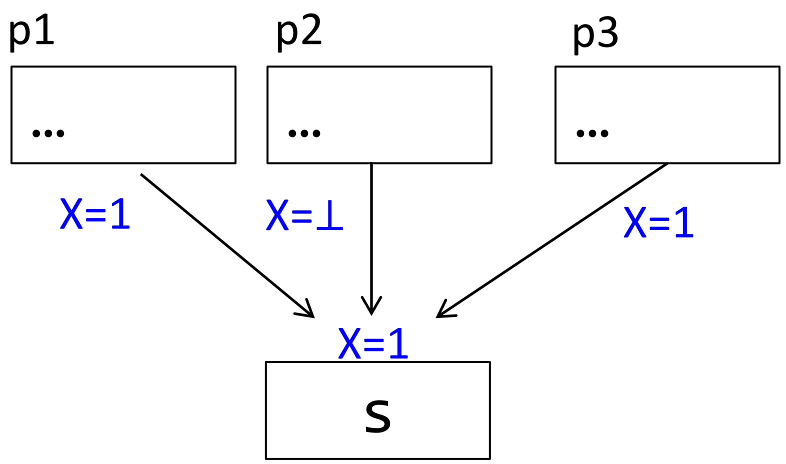



If X=K1 at one predecessor and X=K2 at another predecessor and K1 $\neq$ K2, then X=T at the entrance of S

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):

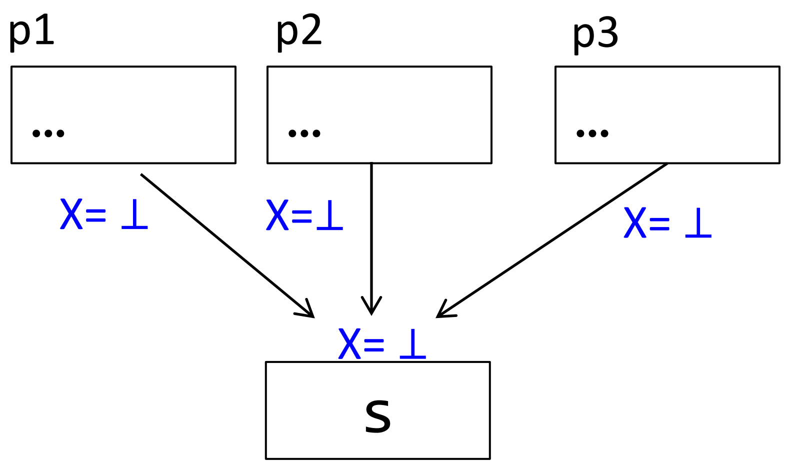



If X=K at some of the predecessors and X= ⊥ at all other predecessors, then X=K at the entrance of S

`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):

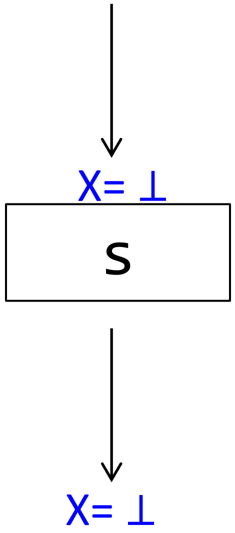



If X= ⊥ at all predecessors, then X= ⊥ at the entrance of S

`if C(p`$_i$`,s,OUT)=⊥` for all `i` then `C(X,s,IN)= ⊥`

# Constant Propagation

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

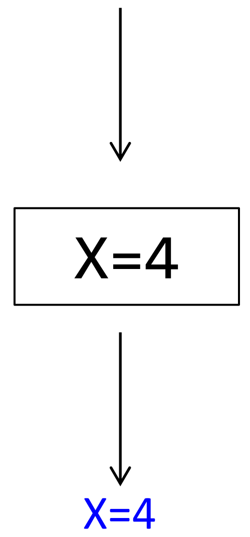



If X= ⊥ at entrance of s, then X= ⊥ at the exit of S

```
if C(X,s,IN)=⊥ then C(X,s,OUT)= ⊥
```

# Constant Propagation

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



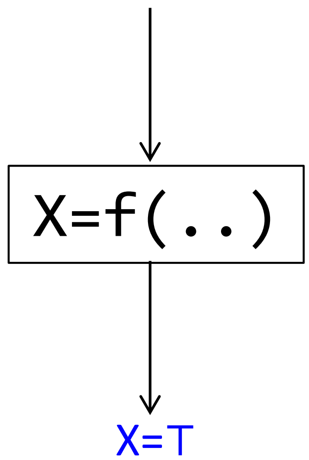

No matter what the value of X is at entrance of `s(X:=K)`, X=K at the exit of s

`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):



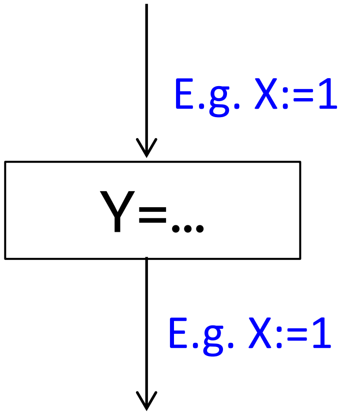

In `s`, assignment to X is any complicated expression (not a constant assignment).

`C(X,s(X:=f()),OUT)=⊤`

But earlier 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 8):




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

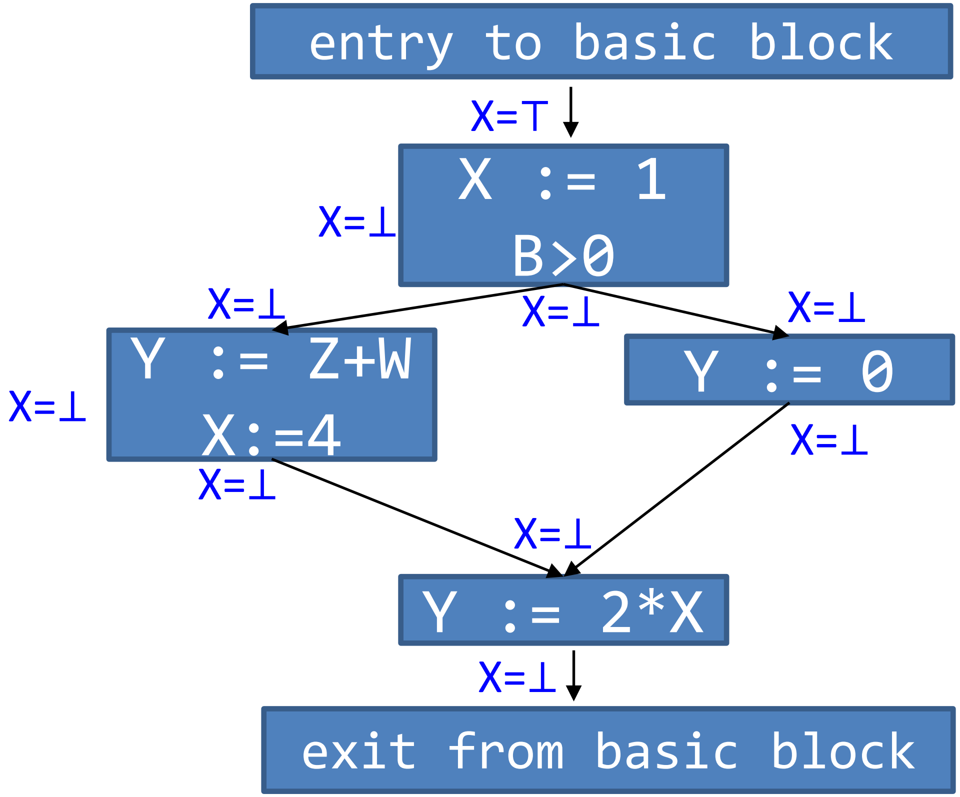
```
C(X,s(Y:=..),OUT)=C(X,s(Y:=..),IN)
```

# 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,IN)=⊥` 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.

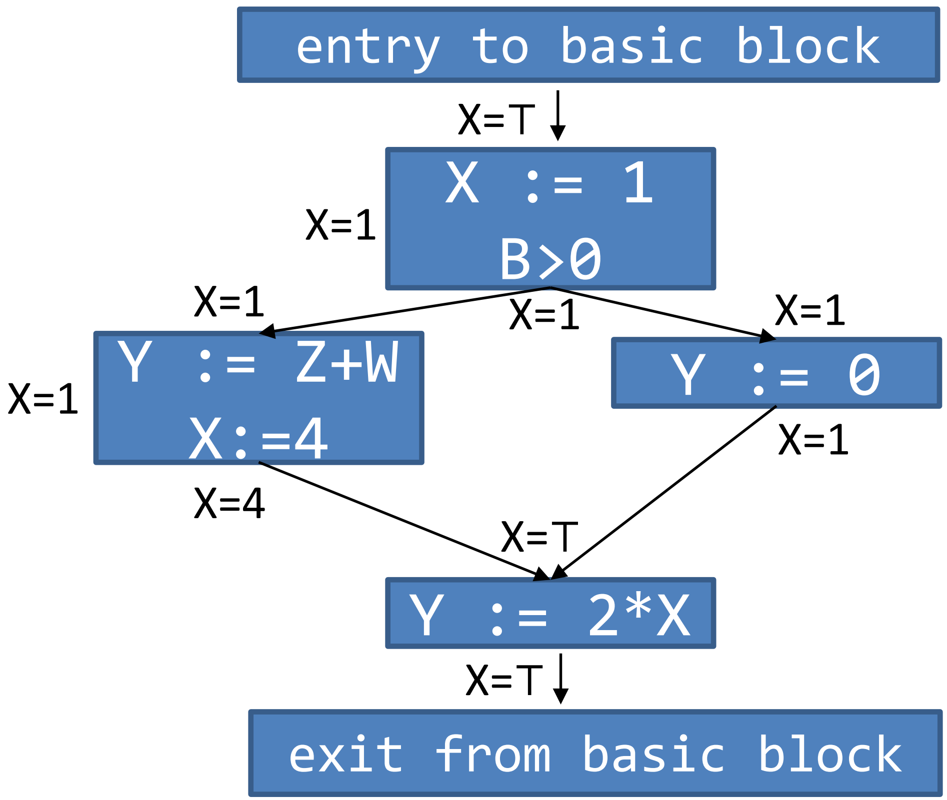
# Constant Propagation

- Putting it all together

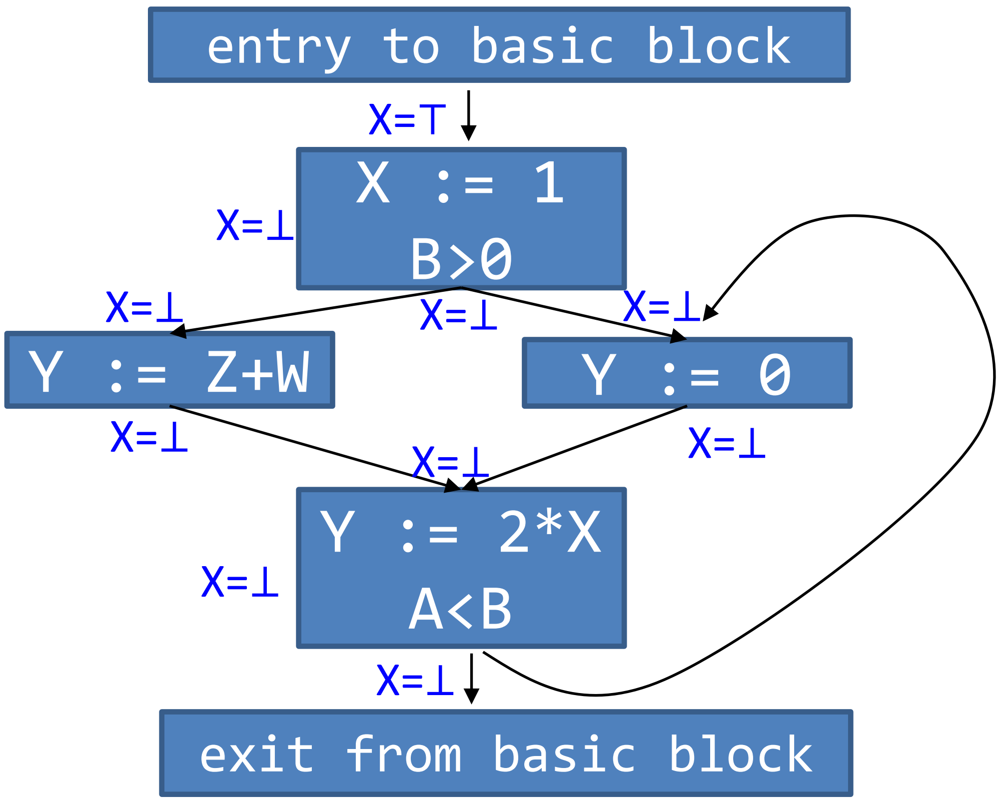


# Constant Propagation

- Putting it all together



# Constant Propagation - Loops



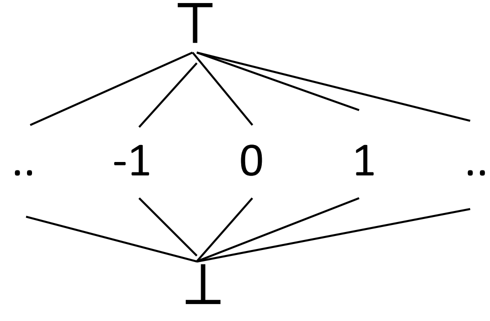
# Ordering of information: Generalizing

- We have been executing with symbols ⊥, ⊤ , 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:
      – ⊥ can change to constant and then to ⊤
      – ⊥ can change to ⊤
      – `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).

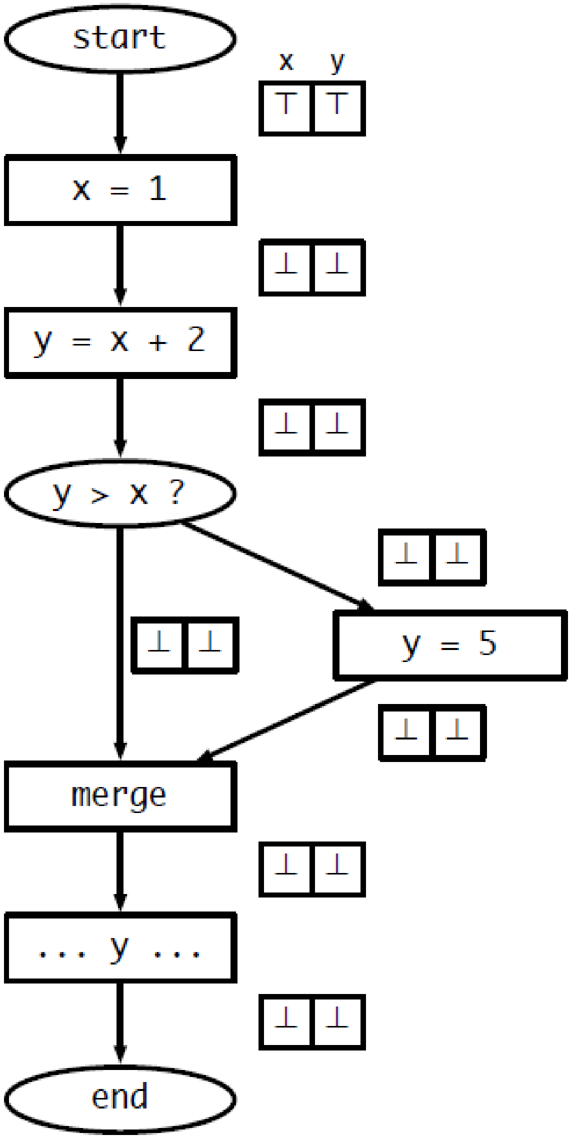
- Per program point, we evaluate the C function.
- The C function changes value at most two times (initialized to $\perp$ first and then could change to K and then to $\top$).
- There are two program points (entry/IN and exit/OUT) for every statement.

*This is the complexity of the analysis per variable*

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

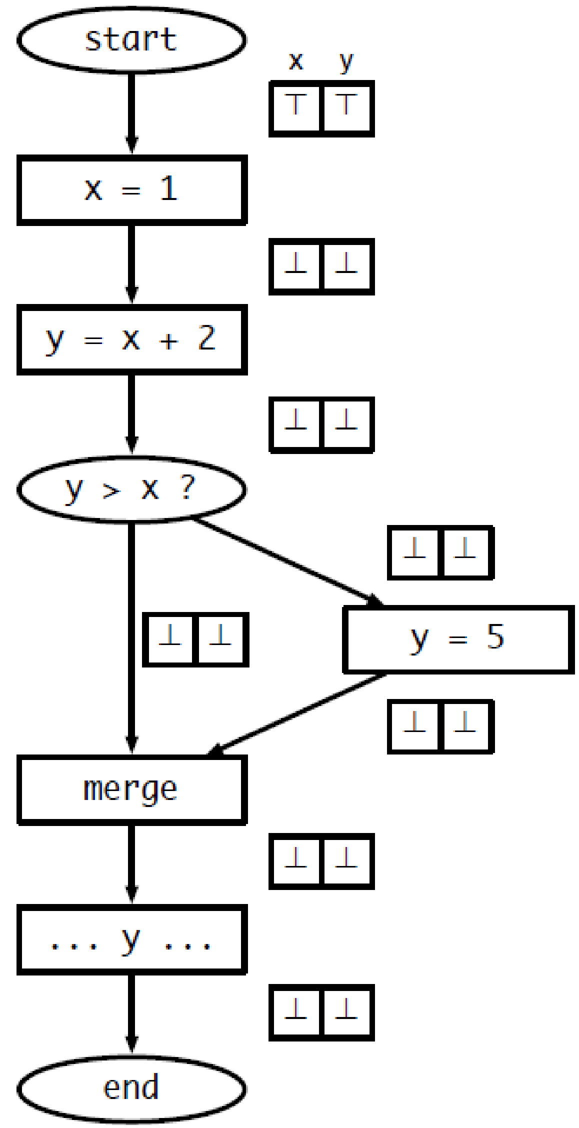
# Constant Propagation (Multiple Variables)

- 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 ⊤
    - Can't make any assumptions about inputs – must assume not constant
  - Everything else starts as ⊥, since we have no information about the variable at that point



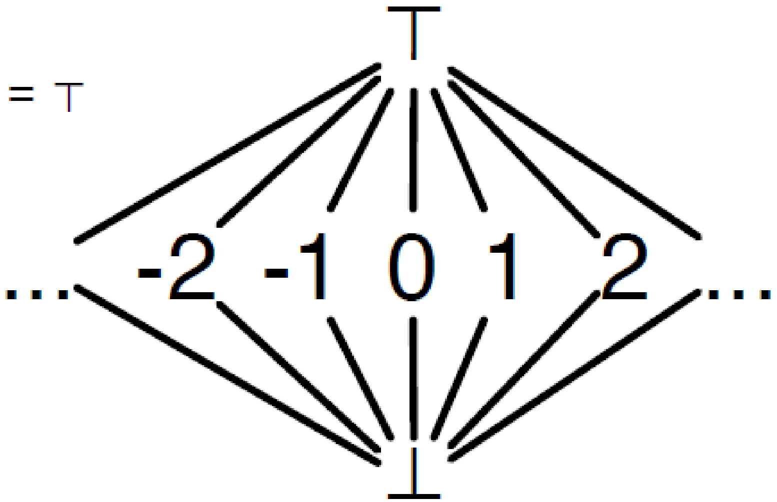
# Constant Propagation (Multiple Variables)

- 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?)

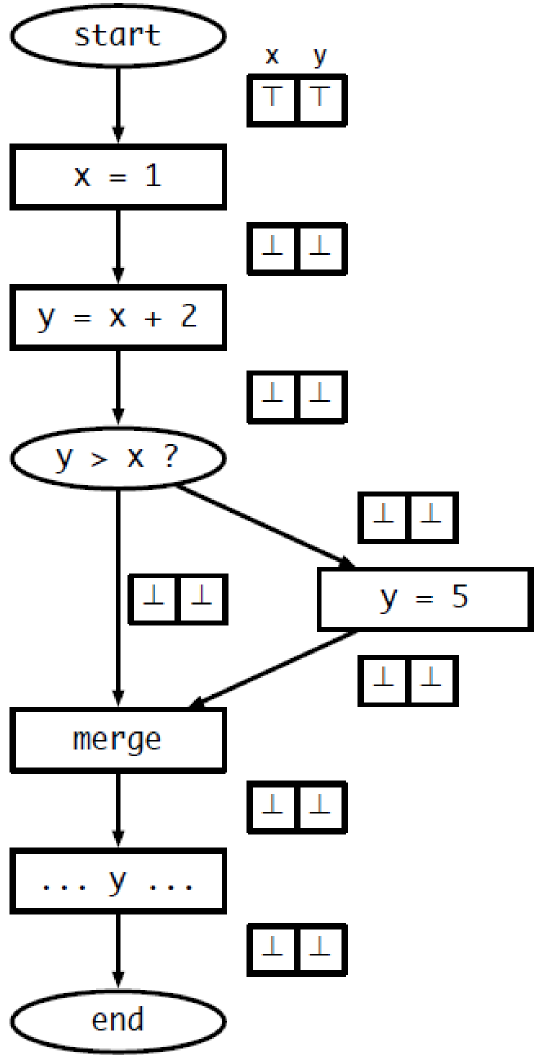
    If eval(e) is true or false, propagate $V_{in}$ to appropriate output edge

    Else, propagate $V_{in}$ along both output edges

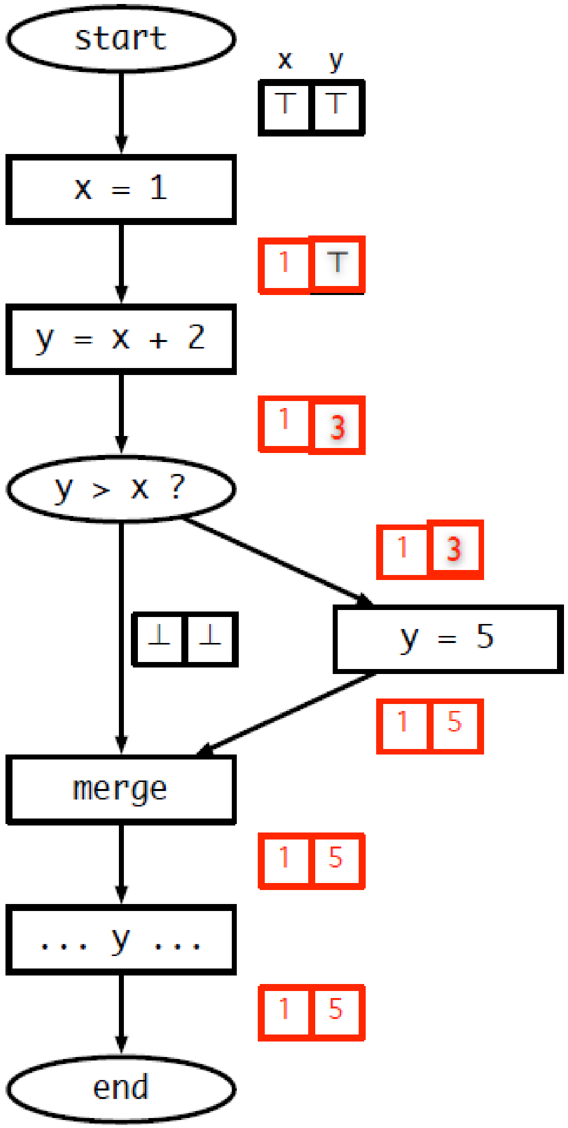
    If target node is merge, propagate join(all $V_{in}$) to output edge

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

# Running example



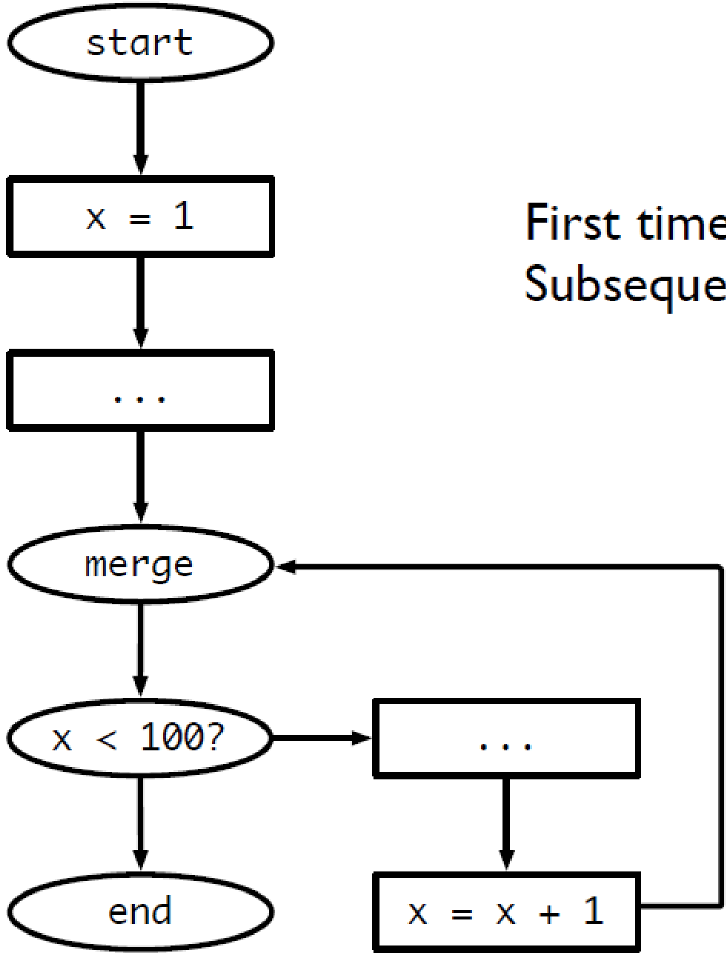
# Running example



# What do we do about loops?

- Unless a loop never executes, symbolic execution looks like it will keep going around to the same nodes over and over again
- Insight: if the input state vector(s) for a node don't change, then its output doesn't change
  - If input stops changing, then we are done!
- Claim: input will eventually stop changing. Why?

# Loop example




First time through loop, x = 1
Subsequent times, x = ⊤

# Complexity of algorithm

- V = # of variables, E = # of edges
- Height of lattice = 2 → each state vector can be updated at most 2 * V times.
- So each edge is processed at most 2 * V times, so we process at most 2 * E * V elements in the worklist.
- Cost to process a node: O(V)
- Overall, algorithm takes $O(EV^2)$ time

# Question

- Can we generalize this algorithm and use it for more analyses?

# Constant propagation

- Step 1: choose lattice (which values are you going to track during symbolic execution)?
  - Use constant lattice
- Step 2: choose direction of dataflow (if executing symbolically, can run program backwards!)
  - Run forward through program
- Step 3: create *transfer functions*
  - How does executing a statement change the symbolic state?
- Step 4: choose *confluence operator*
  - What do do at merges? For constant propagation, use join

# Recap: Constant Propagation

# How can we find constants?

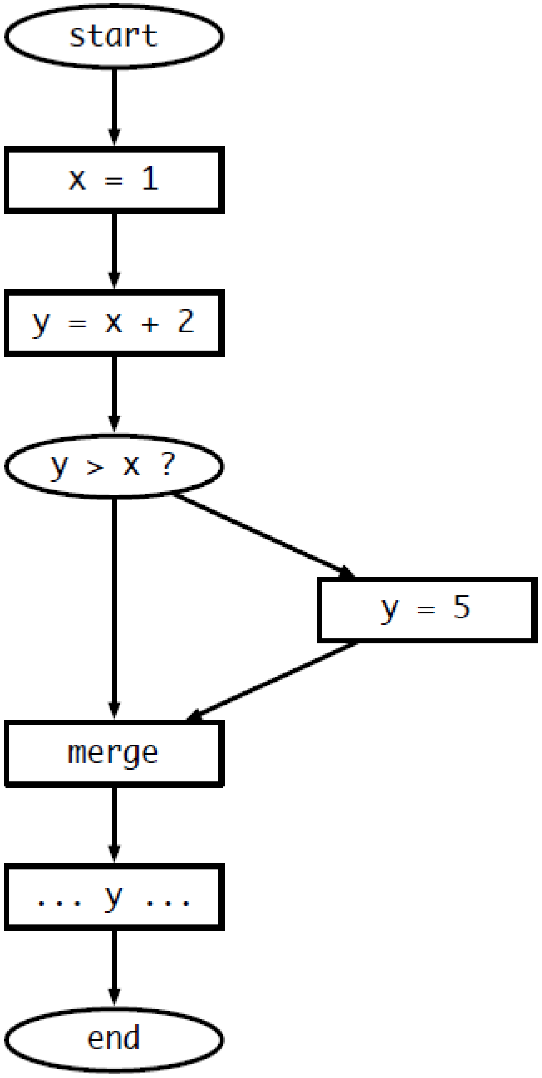
- Ideal: run program and see which variables are constant
  - Problem: variables can be constant with some inputs, not others – need an approach that works for all inputs!
  - Problem: program can run forever (infinite loops?) – need an approach that we know will finish
- Idea: run program *symbolically*
  - 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

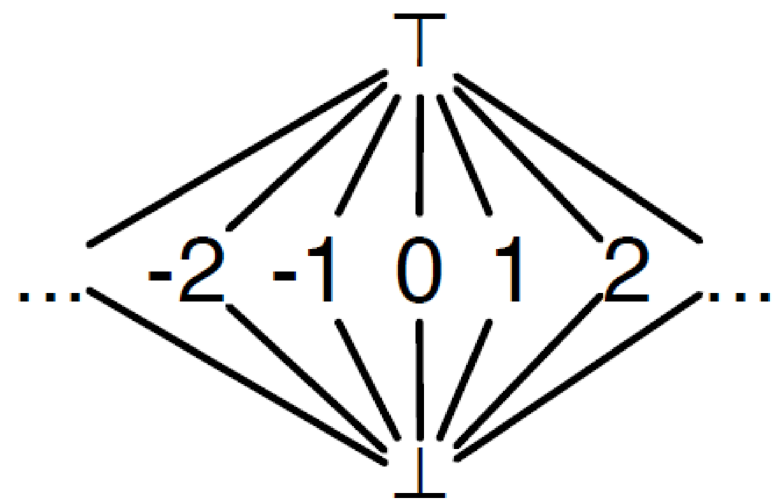
# Build CFG

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



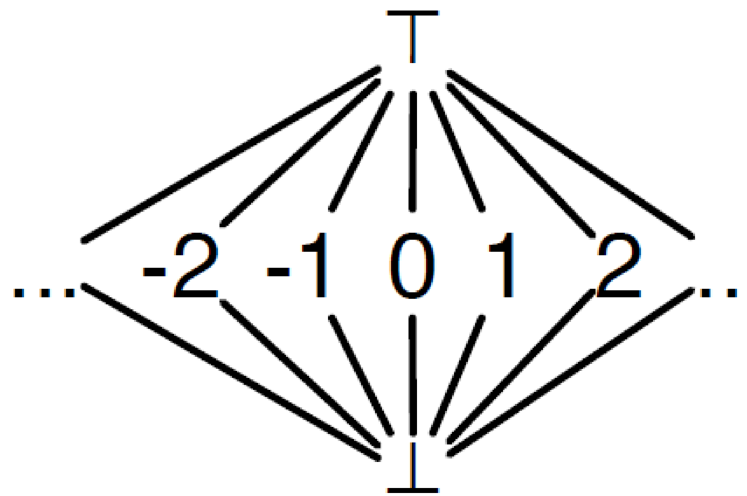
# Symbolic evaluation

- 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

- Evaluate expressions symbolically: eval(e, $V_{in}$)
  - If e evaluates to a constant, return that value. If any input is ⊤ (or ⊥), return ⊤ (or ⊥)
    - Why?
- Two special operations on lattice
  - meet(a, b) – highest value less than or equal to both a and b
  - join(a, b) – lowest value greater than or equal to both a and b




Join often written as a ⊔ b
Meet often written as a ⊓ b

# Exercises

- Analysis of uninitialized variables
- Analysis of available expressions

  - What is the direction of analysis?
  - What is the transfer function?