

CS406: Compilers

Spring 2022

Week 11: Loop Optimization, ..

Optimize Loops

- Example - Code Motion

Should be careful while doing optimization of loops

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

- Optimization: can move 10/I out of loop.

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

- Optimization: can move $10/I$ out of loop
- What if $I = 0$?

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

- Optimization: can move $10/I$ out of loop
- What if $I = 0$?
- What if $I \neq 0$ but loop executes zero times?

Optimization Criteria - Safety and Profitability

- **Safety** - is the code produced after optimization producing same result?
- **Profitability** - is the code produced after optimization running faster or uses less memory or triggers lesser number of page faults etc.

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

- E.g. moving I out of the loop introduces exception (when I=0)
- E.g. if the loop is executed zero times, moving $A(j) := 10/I$ out is not profitable

Optimize Loops – Code Generation

- The outline of code generation for 'for' loops looked like this:

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)  
    <stmt_list>  
end
```

↓

```
<init_stmt>  
LOOP:  
    <bool_expr>  
    j<!op> OUT  
    <stmt_list>  
INCR:  
    <incr_stmt>  
    jmp LOOP  
OUT:
```

```
for (i=0; i<=255;i++) {  
    <stmt_list>  
}
```

↓ **Naïve code generation**

```
code for i=0;  
LOOP:  code for i<=255  
        jump0 OUT  
        code for <stmt_list>  
INCR:  code for i++  
        jump LOOP  
OUT:
```

Question: why naïve is not good?

Optimize Loops – Code Generation

- What happens when `ub` is set to the maximum possible integer representable by the type of `i`?

```
for (i=0; i<=255;i++) {  
    <stmt_list>  
}
```

Better code:



```
code for i=0;  
code for lb=1, ub=255  
code for lb<=ub  
jump0 OUT
```

```
LOOP:  code for <stmt_list>  
       code for lb=ub  
       jump1 OUT
```

```
INCR:  code for i++  
       jump LOOP
```

```
OUT:
```

—————→
generalizing:

```
code for i=0;  
compute lb, ub  
code for lb<=ub  
jump0 OUT  
assign index=lb  
assign limit=ub
```

```
LOOP:  code for <stmt_list>  
       code for index=limit  
       jump1 OUT
```

```
INCR:  code for increment index  
       jump LOOP
```

```
OUT:
```


Optimize Loops -Identifying Invariant Expressions

- How do we identify expressions that can be moved out of the loop?
 - LoopDef = { } set of variables defined (i.e. whose values are overwritten) in the loop body
 - LoopUse = { } 'relevant' variables used in computing an expression

```
Mark_Invariants(Loop L) {  
    1. Compute LoopDef for L  
    2. Mark as invariant all expressions,  
       whose relevant variables don't belong  
       to LoopDef
```

```
}
```

Optimize Loops -Identifying Invariant Expressions

- Example

LoopDef{ }

```
for I = 1 to 100      → {A, J, K, I}
  for J = 1 to 100    → {A, J, K}
    for K = 1 to 100  → {A, K}
      A[I][J][K] = (I*J)*K
```

Optimize Loops -Identifying Invariant Expressions

- Example

LoopUse{}

```
for I = 1 to 100      _____→ {}  
  for J = 1 to 100    _____→ {I}  
    for K = 1 to 100  _____→ {I, J}  
      A[I][J][K] = (I*J)*K
```

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant
Expressions

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100 —————→ { I*J,
      A[I][J][K] = (I*J)*K          Addr(A[i][j]) }
```

For an array access, $A[m] \Rightarrow \text{Addr}(A) + m$

For 3D array above*, $\text{Addr}(A[I][J][K]) =$

$$\text{Addr}(A) + (I * 10000) - 10000 + (J * 100) - 100 + K - 1$$

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant
Expressions

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100
      A[I][J][K] = (I*J)*K
```

→ { Addr(A[i]) }

For an array access, $A[m] \Rightarrow \text{Addr}(A) + m$

For 3D array above*, $\text{Addr}(A[I][J][K]) =$

$$\text{Addr}(A) + (I*10000) - 10000 + (J*100) - 100 + K - 1$$

Optimize Loops -Factoring Invariant Expressions

- Move the invariant expressions identified

```
Factor_Invariants(Loop L) {  
    Mark_Invariants(L);  
    foreach expression E marked an invariant:  
        1. Create a temporary T  
        2. Replace each occurrence of E in L with T  
        3. Insert T:=E in L's header code  
           //If loop is known to execute at least once,  
           insert T:=E before LOOP:  
}
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100
      A[I][J][K] = (I*J)*K
```

//Invariant Expressions

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
  for J = 1 to 100
    temp1=A[I][J]
    temp2=I*J
    for K = 1 to 100
      temp1[K] = temp2*K
```


Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
  temp3=A[I]
  for J = 1 to 100
    temp1=temp3[J]
    temp2=I*J
    for K = 1 to 100
      temp1[K] = temp2*K
```

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case I: We can move $t = a \text{ op } b$ if the statement dominates all loop exits where t is live

A node $bb1$ dominates node $bb2$ if all paths to $bb2$ must go through $bb1$

```
for (...) {  
    if(*)  
        a = 100  
}  
c=a
```

Cannot move $a=100$ because it does not dominate $c=a$ i.e. there is one path (when if condition is false) $c=a$ can be executed /'reached' without going to $a=100$

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case II: We can move $t = a \text{ op } b$ if there is only one definition of t in the loop

```
for (...) {  
    if(*)  
        a = 100  
    else  
        a = 200  
}
```

Multiple definition of a

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case III: We can move $t = a \text{ op } b$ if t is not defined before the loop, where the definition reaches t 's use after the loop

```
a=5
for (...) {
    a = 4+b
}
c=a
```

Definition of a in $a=5$ reaches $c=a$, which is defined after the loop

Optimize Loops –Strength Reduction

- Like strength reduction in peephole optimization
 - E.g. replace $a*2$ with $a<<1$
- Applies to uses of **induction variable** in loops
 - **Basic induction variable (I)** – only definition within the loop is of the form $I = I \pm S$, (S is loop invariant)
I usually determines number of iterations
 - **Mutual induction variable (J)** – defined within the loop, its value is linear function of other induction variable, I, such that
$$J = I * C \pm D \quad (C, D \text{ are loop invariants})$$

Optimize Loops –Strength Reduction

```
strength_reduce(Loop L) {  
    Mark_Invariants(L);  
    foreach expression  $E$  of the form  $I * C + D$  where  $I$  is  
    L's loop index and  $C$  and  $D$  are loop invariants  
        1. Create a temporary  $T$   
        2. Replace each occurrence of  $E$  in  $L$  with  $T$   
        3. Insert  $T := I_0 * C + D$ , where  $I_0$  is the initial value of the  
           induction variable, immediately before  $L$   
        4. Insert  $T := T + S * C$ , where  $S$  is the step size, at the end of  
           L's body  
}
```

Optimize Loops –Strength Reduction

- Suppose induction variable I takes on values I_0 , I_0+S , I_0+2S , $I_0+3S \dots$ in iterations 1, 2, 3, 4, and so on...
- Then, in consecutive iterations, Expression $I*C+D$ takes on values

$$I_0 * C + D$$

$$(I_0 + S) * C + D = I_0 * C + S * C + D$$

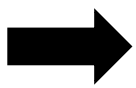
$$(I_0 + 2S) * C + D = I_0 * C + 2S * C + D$$

- The expression \ddots changes by a constant $S * C$
- Therefore, we have replaced a $*$ and $+$ with a $+$


Optimize Loops – Strength Reduction

- Example (Applying to innermost loop)

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100
      A[I][J][K] = (I*J)*K
    . . .
    temp2=I*J
    temp4=temp2
    for K=1 to 100
      temp1[K]=temp4
      temp4=temp4+temp2
    //S=1
    //C=temp2
```

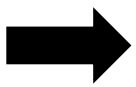



```
for I=1 to 100
  temp3=Addr(A[i])
  for J=1 to 100
    temp1=Addr(temp3(J))
    temp2=I*J
    for K=1 to 100
      temp1[K]=temp2*K
    temp1[K]=temp2*K
    temp4=temp4+temp2
```



Optimize Loops – Strength Reduction

- Exercise (Apply to intermediate loop)

<pre>for I=1 to 100 temp3=Addr(A[i]) for J=1 to 100 temp1=Addr(temp3(J)) temp2=I*J for K=1 to 100 temp1[K]=temp2*K</pre>		<pre>. . . temp2=I*J temp4=temp2 for K=1 to 100 temp1[K]=temp4 temp4=temp4+temp2</pre>
		

```
// Induction var = J
// S = 1
// Expression = I * J
```

Optimize Loops – Strength Reduction

- Exercise (Apply to intermediate loop)

...
temp5=I

for J=1 to 100

temp1=Addr(temp3(J))

temp2=temp5

temp4=temp2

for K=1 to 100

temp1[K]=temp4

temp4=temp4+temp2

temp5=temp5+I

... → ...



Optimize Loops – Strength Reduction

- Further strength reduction possible?

```
for I=1 to 100
    temp3=Addr(A[i])
    temp5=I
    for J=1 to 100
        temp1=Addr(temp3(J))
        temp2=temp5
        temp4=temp2
        for K=1 to 100
            temp1[K]=temp4
            temp4=temp4+temp2
        temp5=temp5+I
```

Optimize Loops – Loop Unrolling

- Modifying induction variable in each iteration can be expensive
- Can instead *unroll* loops and perform multiple iterations for each increment of the induction variable
- What are the advantages and disadvantages?

```
for (i = 0; i < N; i++)  
    A[i] = ...
```



Unroll by factor of 4

```
for (i = 0; i < N; i += 4)  
    A[i] = ...  
    A[i+1] = ...  
    A[i+2] = ...  
    A[i+3] = ...
```

Optimize Loops - Summary

- Low level optimization
 - Moving code around in a single loop
 - Examples: loop invariant code motion, strength reduction, loop unrolling
- High level optimization
 - Restructuring loops, often affects multiple loops
 - Examples: loop fusion, loop interchange, loop tiling

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 defined here

1: $X = 10$

.....

N: $Y = X + 5$

X used here

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: $X = 10$ X is dead at 1

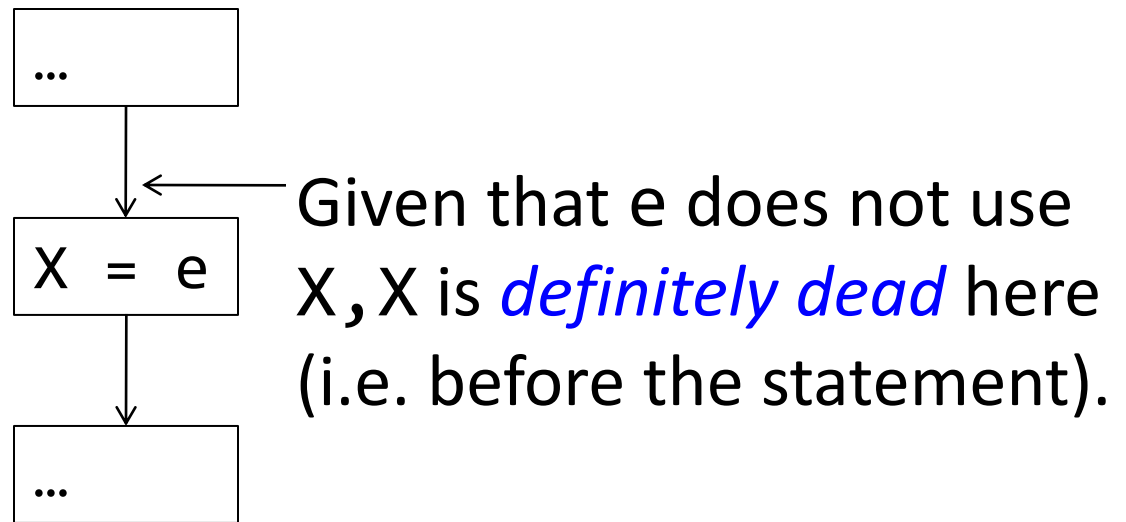
2: $X = Y + 2$

...

N: $Y = X + 5$

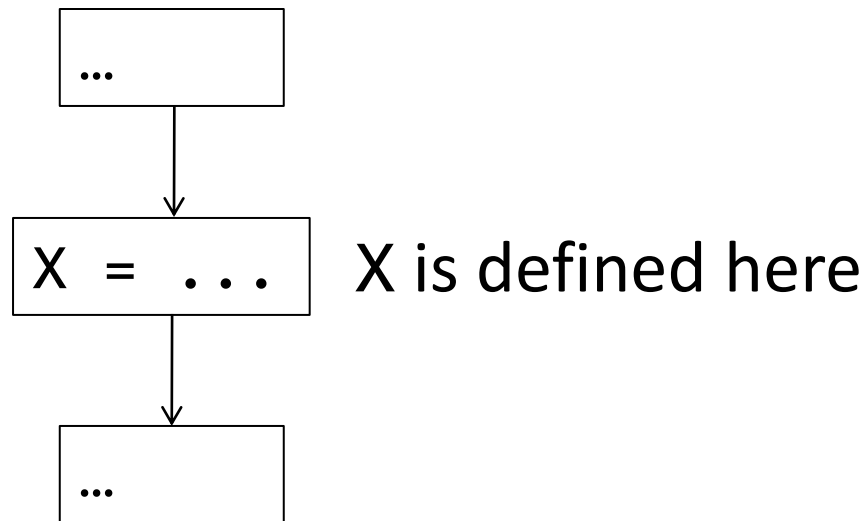
- A variable X is dead at statement S if it is not live at S
 - What about $\dots; X = X + 1$?

Liveness in a CFG



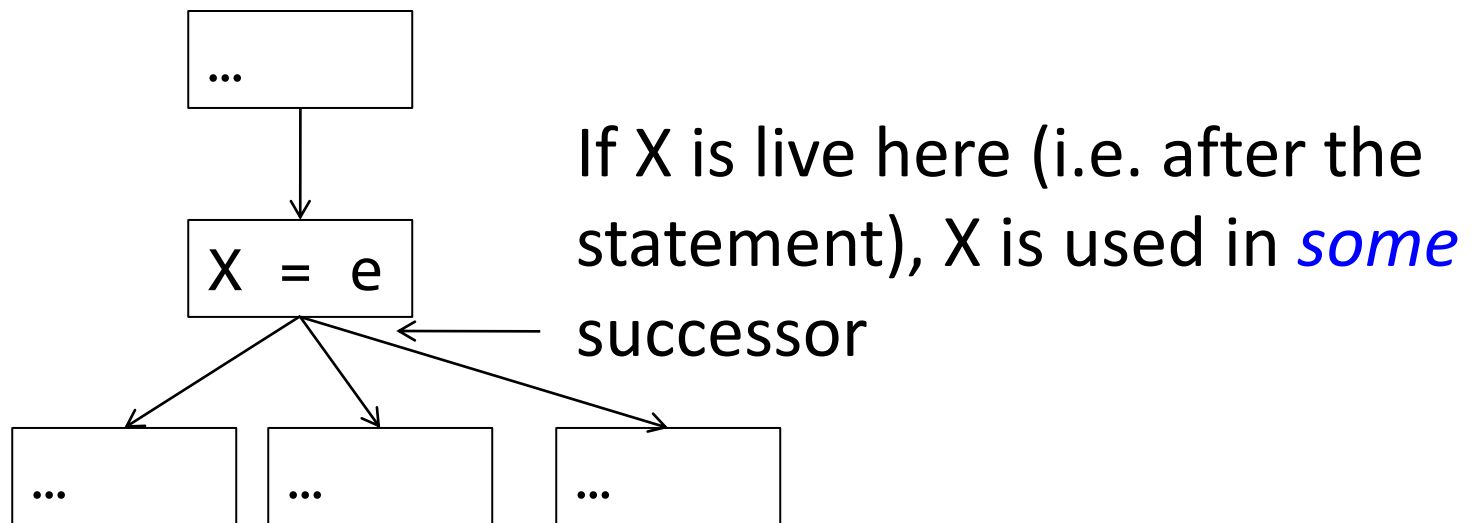
- Define a set $\text{LiveIn}(b)$, where b is a basic block, as: the set of all variables live at the entrance of a basic block

Liveness in a CFG



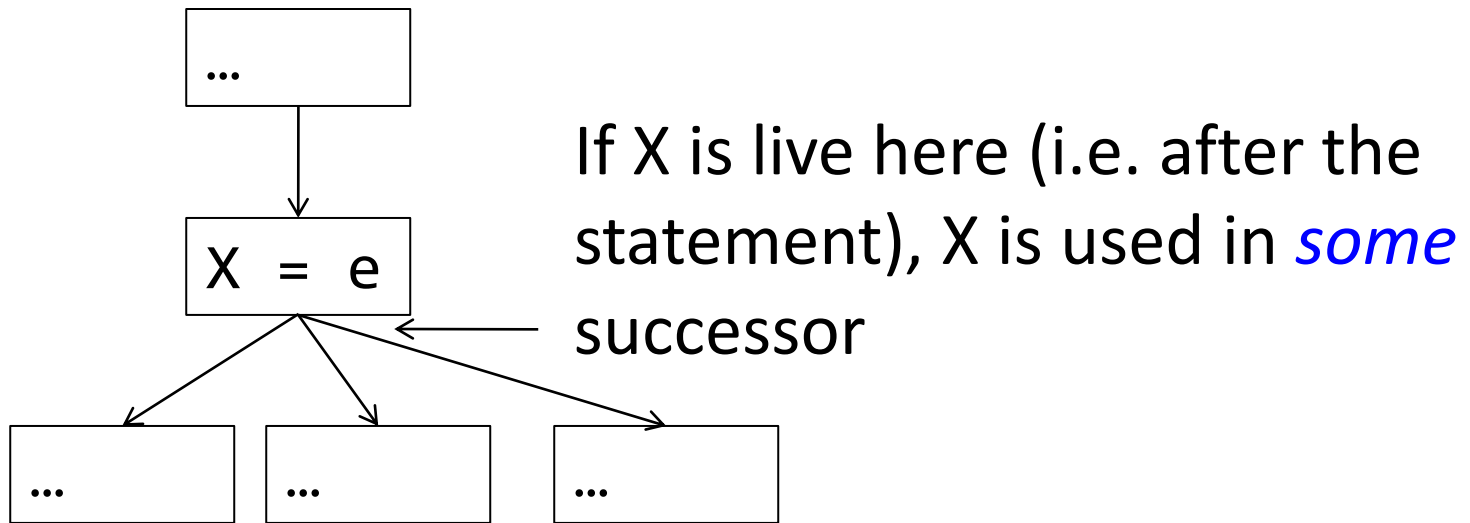
- Define a set $\text{Def}(b)$, where b is a basic block, as: the set of all variables that are defined within block b

Liveness in a CFG



- Define a set $\text{LiveOut}(b)$, where b is a basic block, as: the set of all variables live at the exit of a basic block

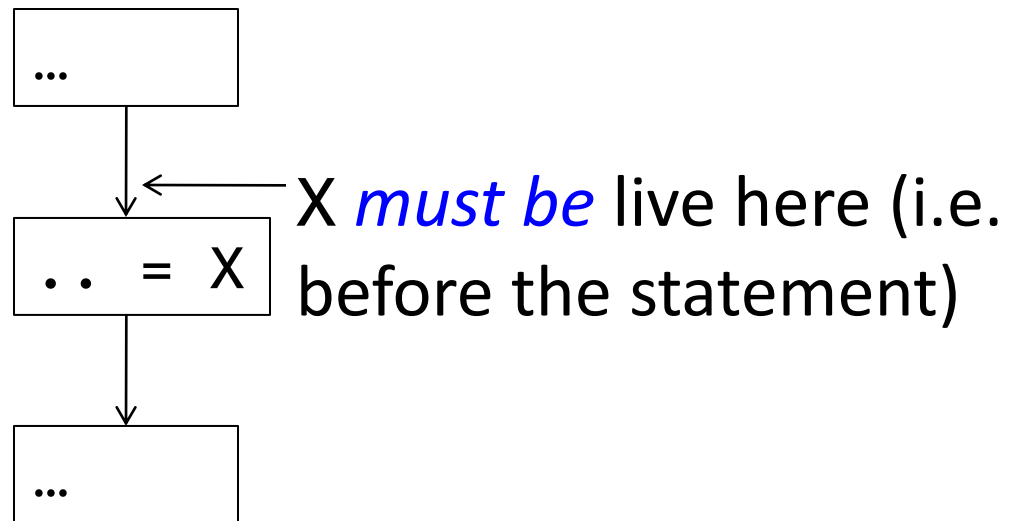
Liveness in a CFG



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

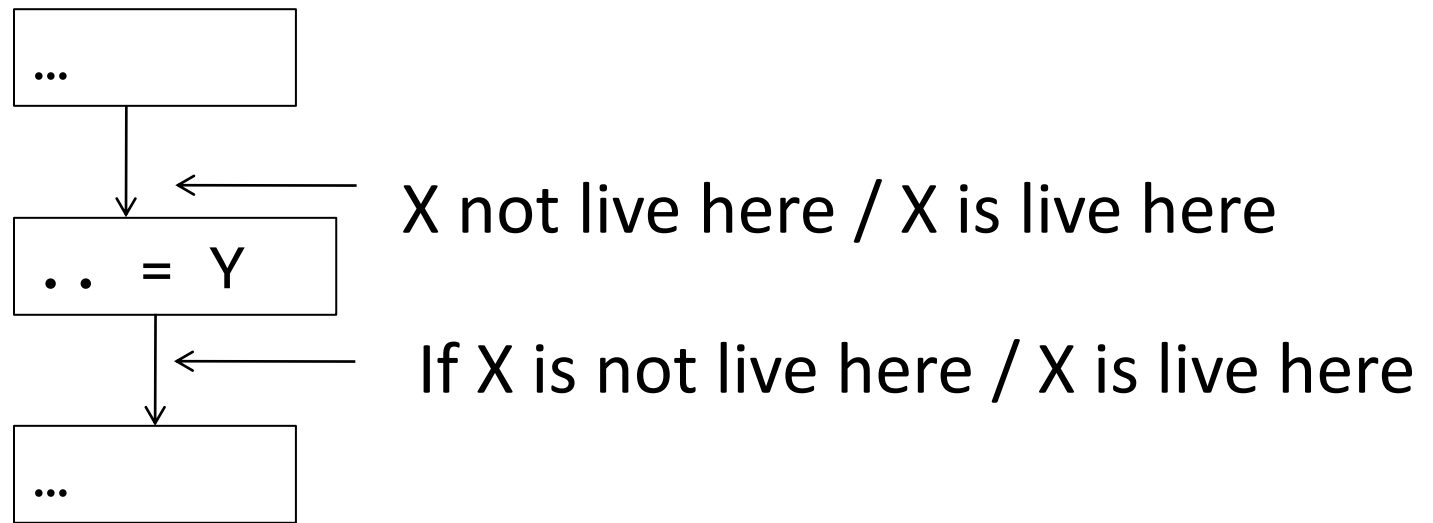
$$\text{LiveOut}(b) = \bigcup_{i \in S(b)} \text{LiveIn}(i)$$

Liveness in a CFG



- Define a set $\text{LiveUse}(b)$, where b is a basic block, as the set of all variables that are used within block b . $\text{LiveIn}(b) \supseteq \text{LiveUse}(b)$

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

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

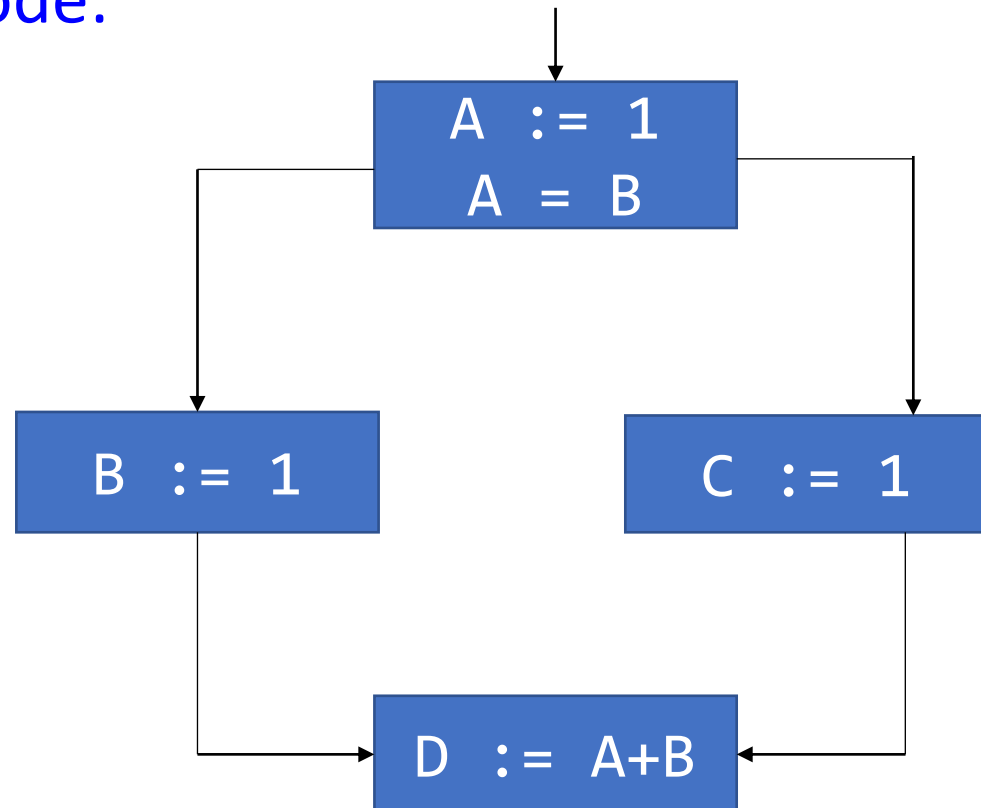
- Under what scenarios can a variable be live at the entrance of a basic block?
 - Either the variable is used in the basic block
 - OR the variable is live at exit and not defined within the block

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

Liveness in a CFG - Example

- Draw CFG for the code:

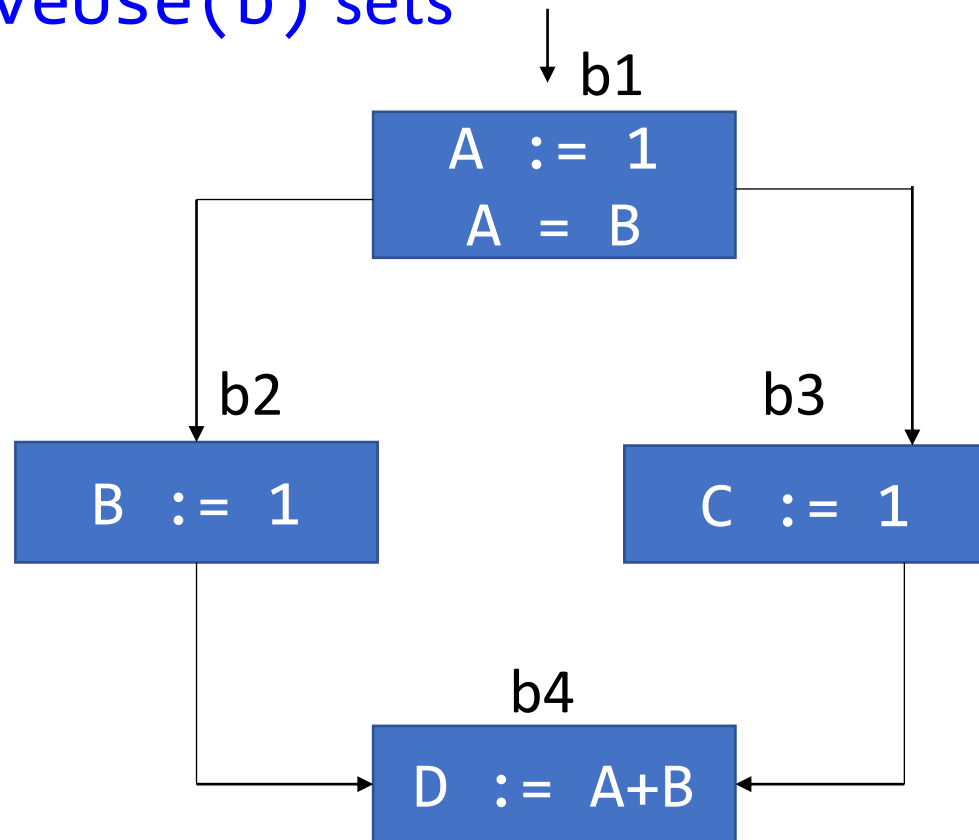
```
A := 1
if A=B then
    B := 1
else
    C := 1
endif
D := A+B
```



Liveness in a CFG - Example

- Compute $\text{Def}(b)$ and $\text{LiveUse}(b)$ sets

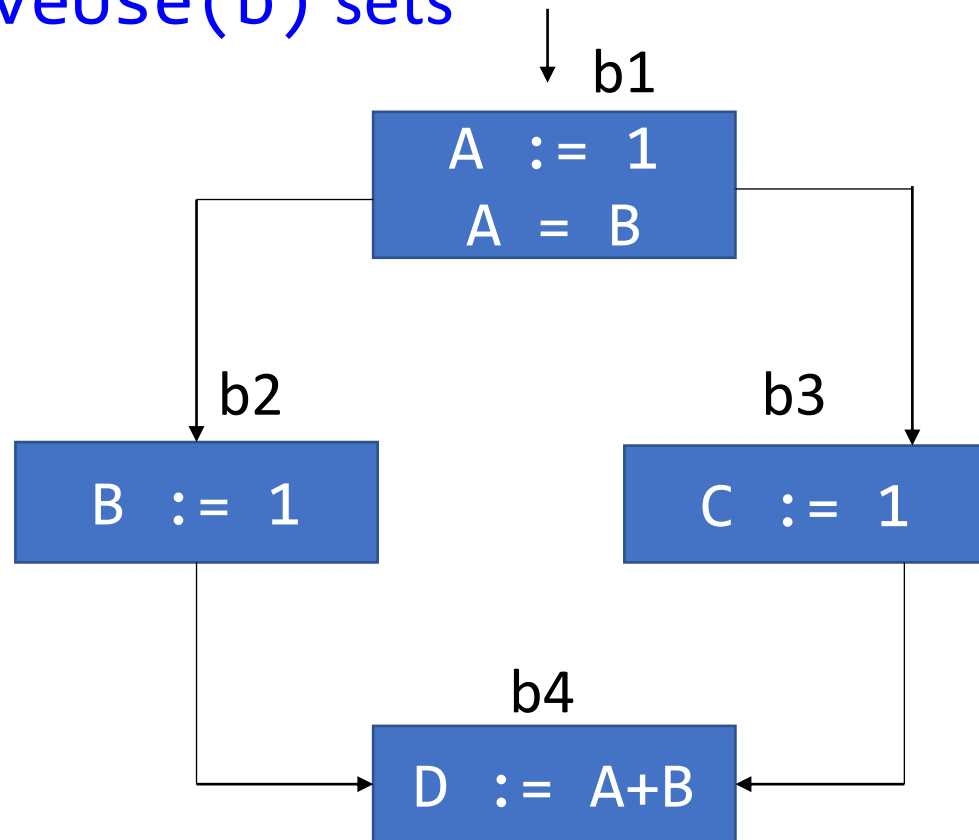
Block	Def	LiveUse
b1		
b2		
b3		
b4		



Liveness in a CFG - Example

- Compute $\text{Def}(b)$ and $\text{LiveUse}(b)$ sets

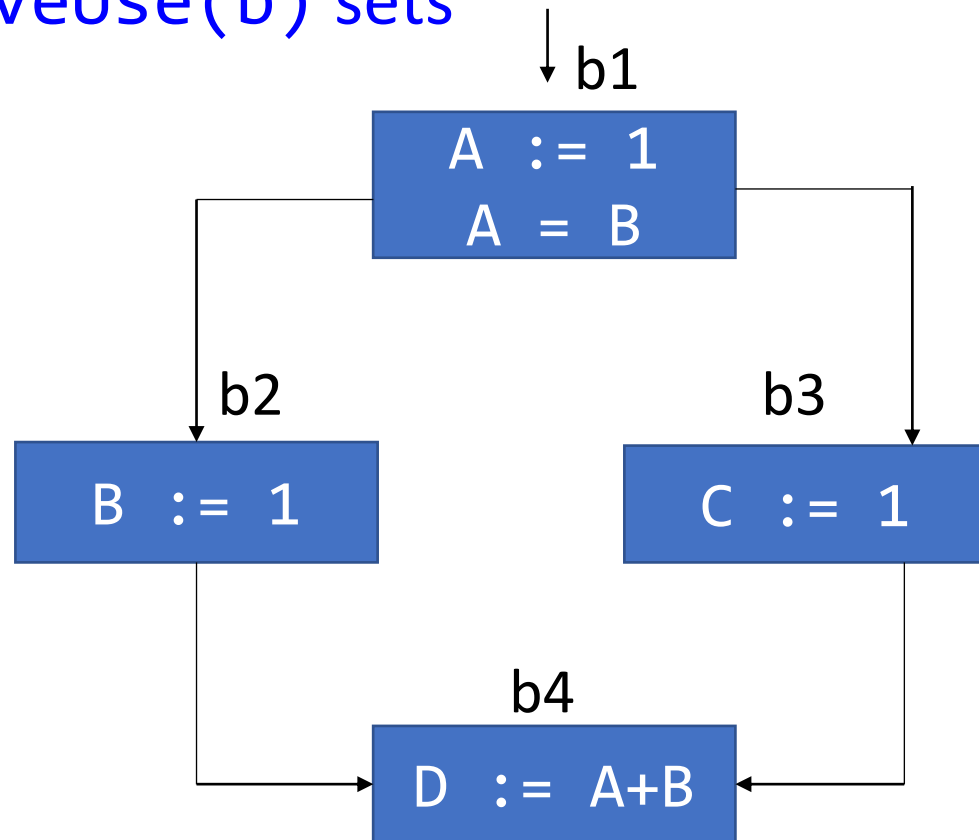
Block	Def	LiveUse
b1	{A}	{B}
b2		
b3		
b4		



Liveness in a CFG - Example

- Compute $\text{Def}(b)$ and $\text{LiveUse}(b)$ sets

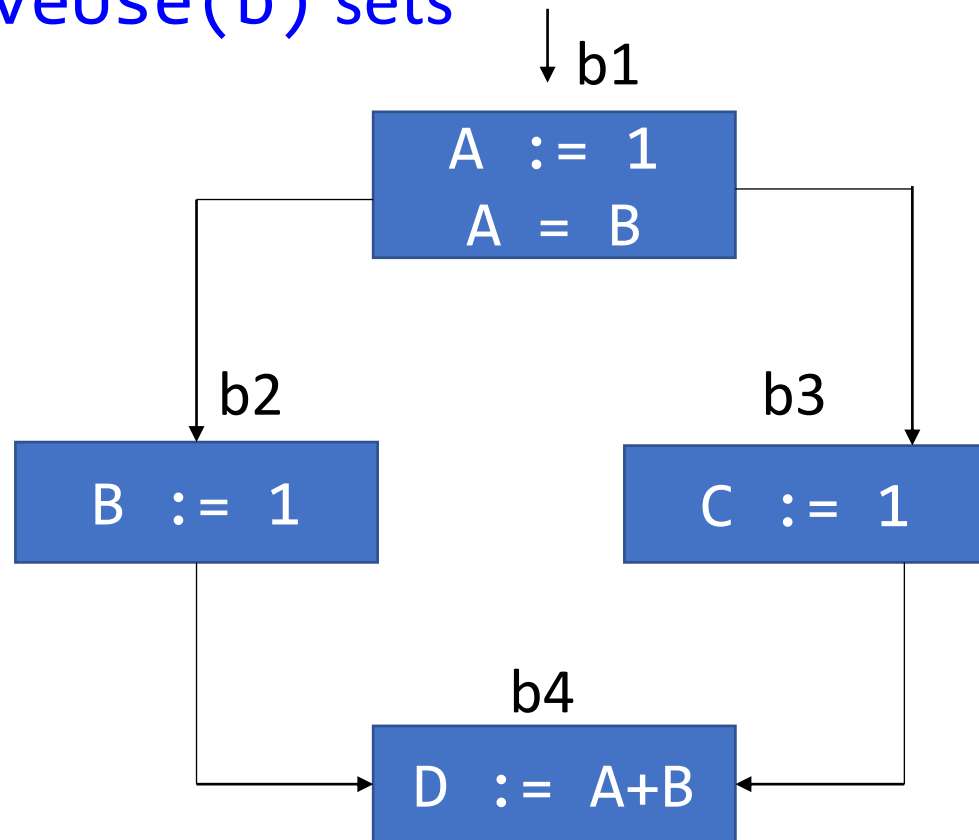
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3		
b4		



Liveness in a CFG - Example

- Compute $\text{Def}(b)$ and $\text{LiveUse}(b)$ sets

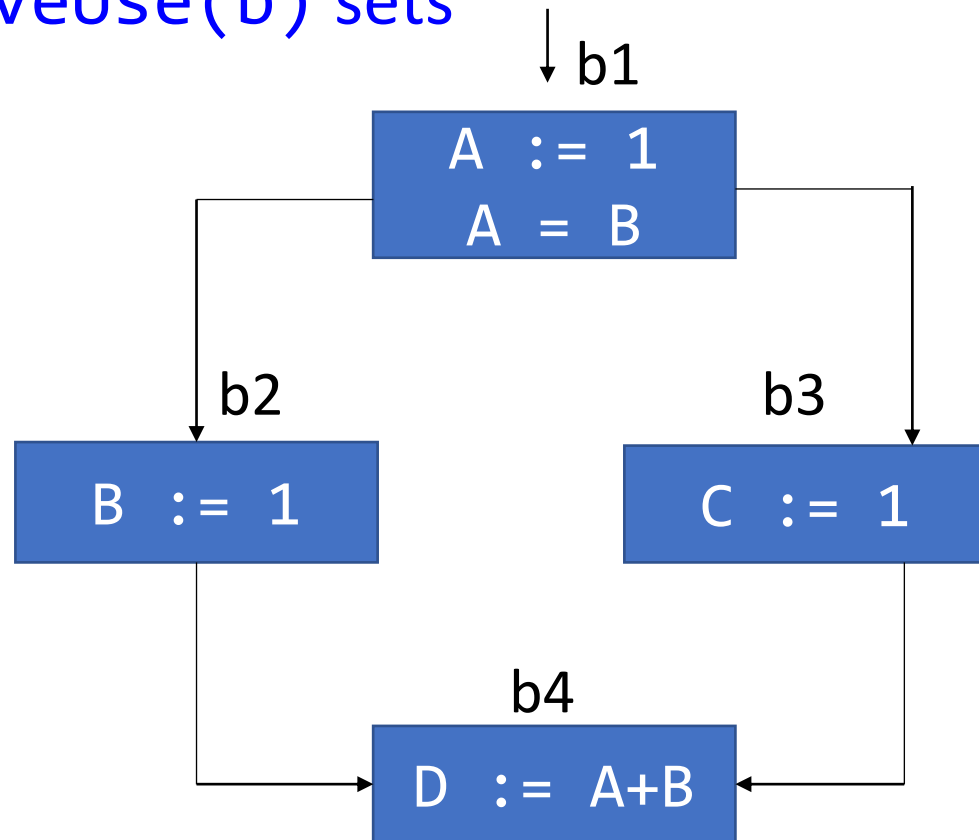
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4		



Liveness in a CFG - Example

- Compute $\text{Def}(b)$ and $\text{LiveUse}(b)$ sets

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

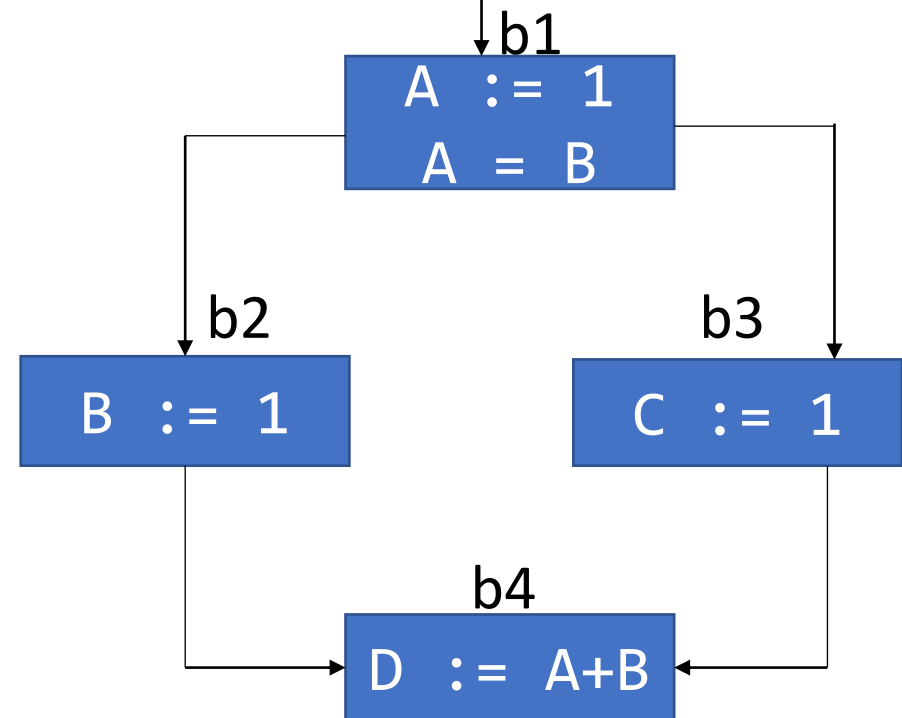


Liveness in a CFG - Example

- 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}

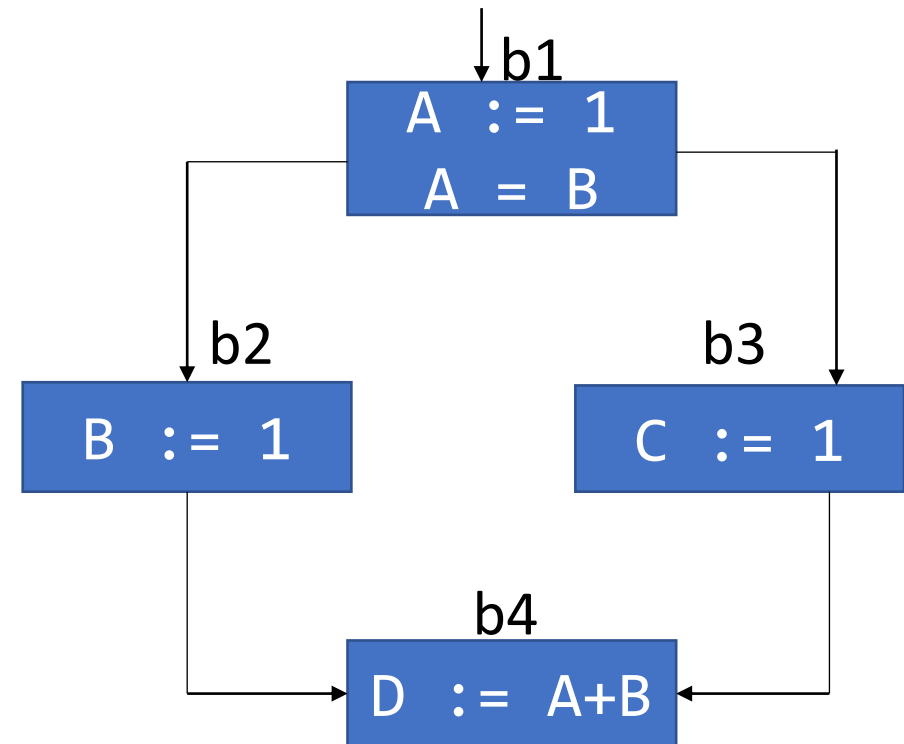


Liveness in a CFG - Example

- 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}

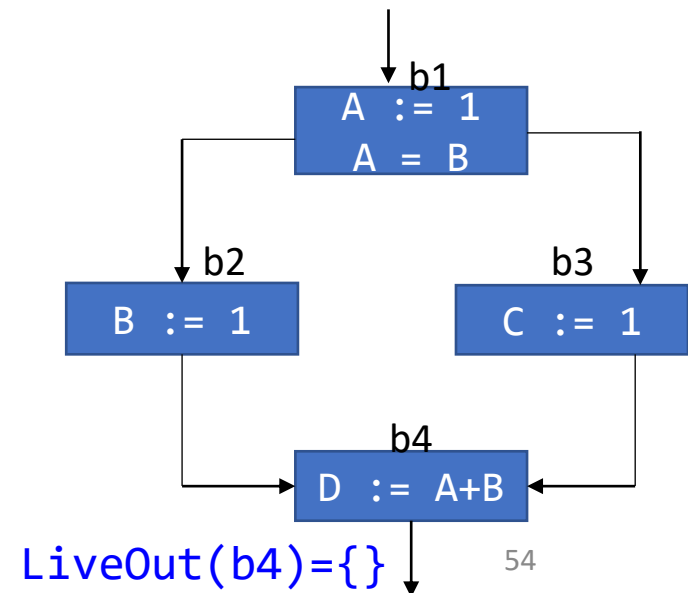


Liveness in a CFG - Example

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

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

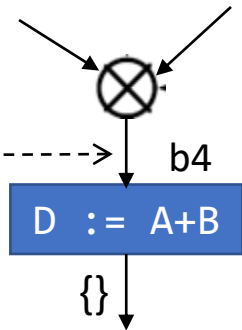
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}



Liveness in a CFG - Example

$$\begin{aligned}\text{LiveIn}(b4) &= \text{LiveUse}(b4) \cup (\text{LiveOut}(b4) - \text{Def}(b4)) \\ &= \{A, B\} \cup (\{\} - \{D\})\end{aligned}$$

Program point



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

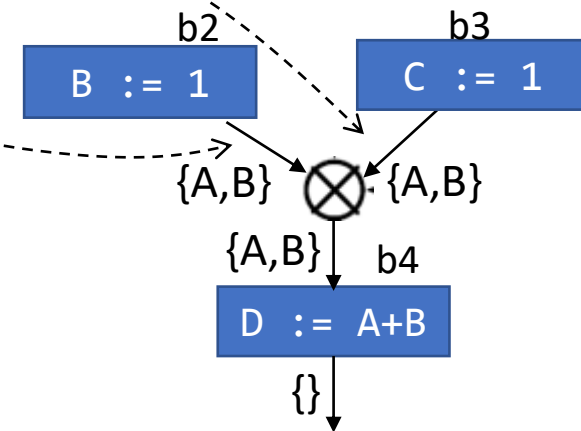
Liveness in a CFG - Example

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

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

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

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

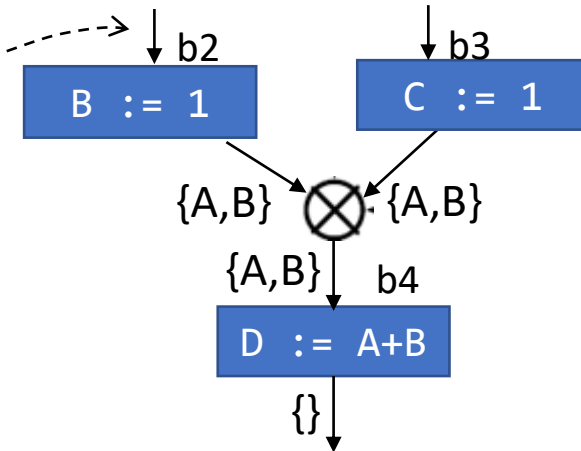


Liveness in a CFG - Example

$$\begin{aligned}\text{LiveIn}(b3) &= \text{LiveUse}(b3) \cup (\text{LiveOut}(b3) - \text{Def}(b3)) \\ &= \{\} \cup (\{A,B\} - \{C\}) = \{A,B\}\end{aligned}$$

$$\begin{aligned}\text{LiveIn}(b2) &= \text{LiveUse}(b2) \cup (\text{LiveOut}(b2) - \text{Def}(b2)) \\ &= \{\} \cup (\{A,B\} - \{B\}) = \{A\}\end{aligned}$$

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

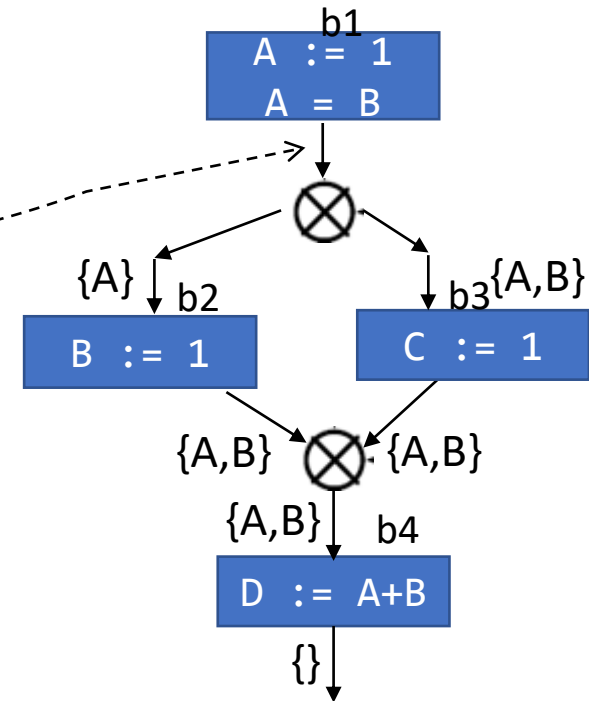


Liveness in a CFG - Example

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

$$\begin{aligned} \text{LiveOut}(b1) &= \text{LiveIn}(b2) \cup \text{LiveIn}(b3) \\ &= \{A\} \cup \{A, B\} = \{A, B\} \end{aligned}$$

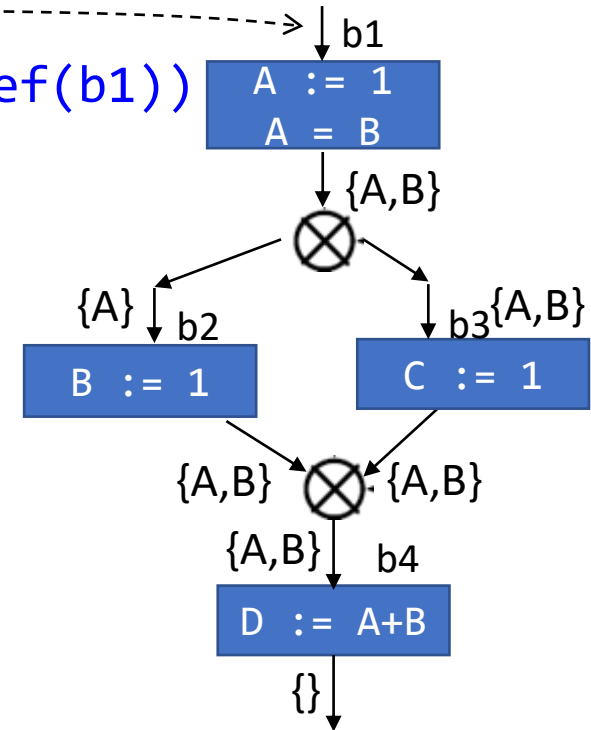
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A, B}



Liveness in a CFG - Example

$$\begin{aligned} \text{LiveIn}(b1) &= \text{LiveUse}(b1) \cup (\text{LiveOut}(b1) - \text{Def}(b1)) \\ &= \{B\} \cup (\{A,B\} - \{A\}) = \{B\} \end{aligned}$$

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



Liveness in a CFG - Example

- Summary: Compute $\text{LiveIn}(b)$ and $\text{LiveOut}(b)$

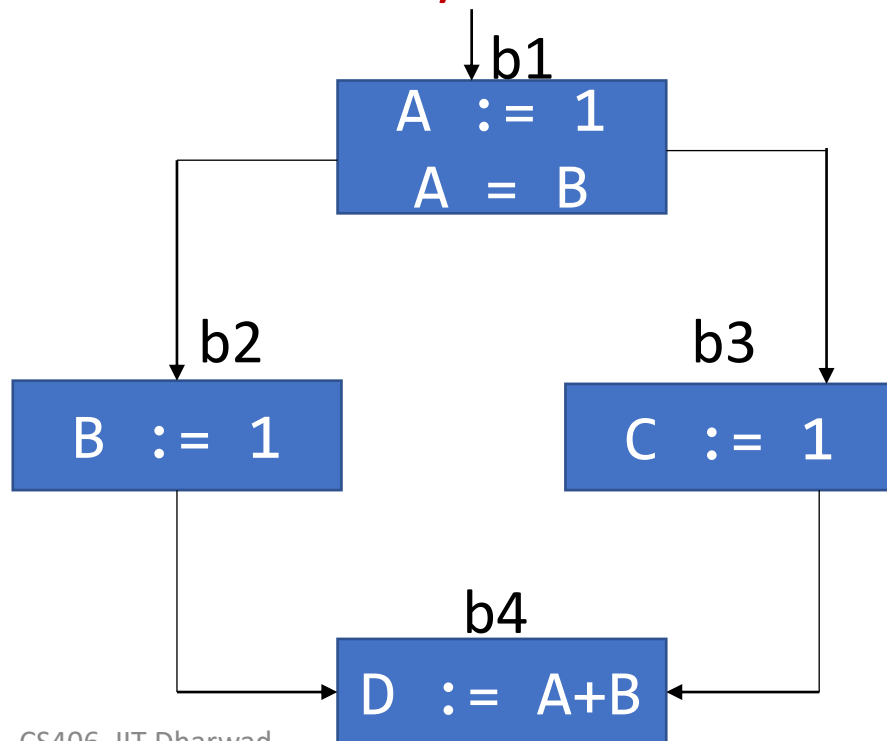
$$\text{LiveIn}(b) = \text{LiveUse}(b) \cup (\text{LiveOut}(b) - \text{Def}(b))$$

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

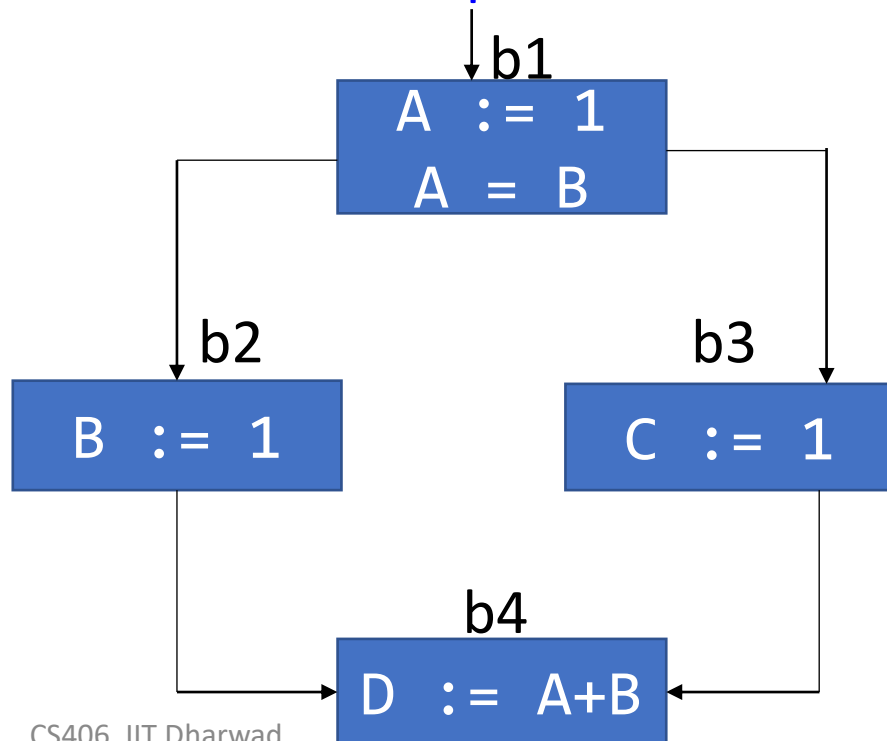
- Assume that the CFG below represents *your entire program* (b1 is the entry to program and b4 is the exit)
- What can you infer from the table?



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



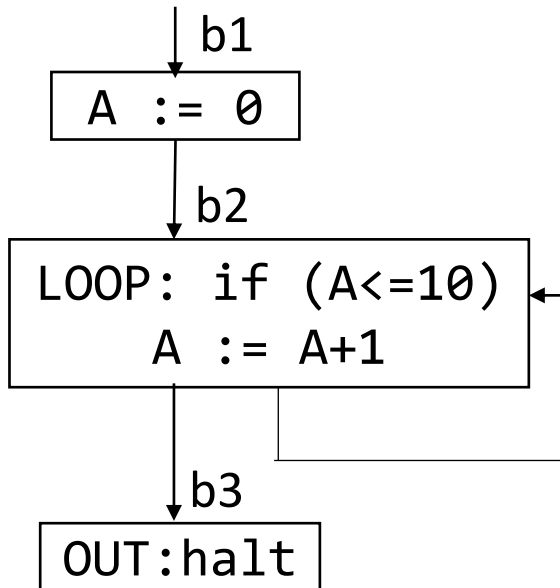
X = 1
Y = 1 + 2
Z = Y + A

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.

How can we find constants?

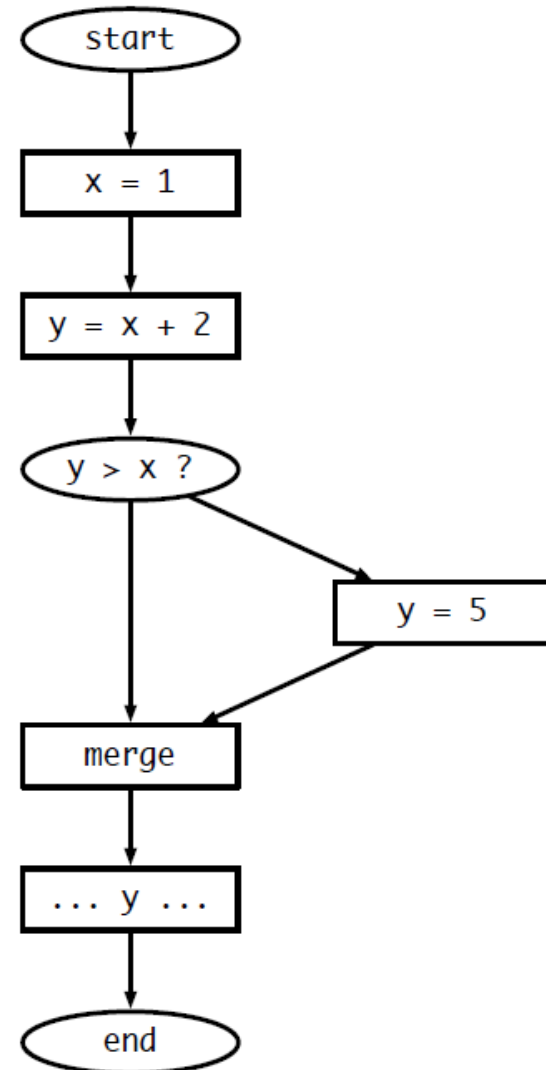
- Ideal: run program and see which variables are constant
 - Problem: variables can be constant with some inputs, not others – need an approach that works for all inputs!
 - Problem: program can run forever (infinite loops?) – need an approach that we know will finish
- Idea: run program *symbolically*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

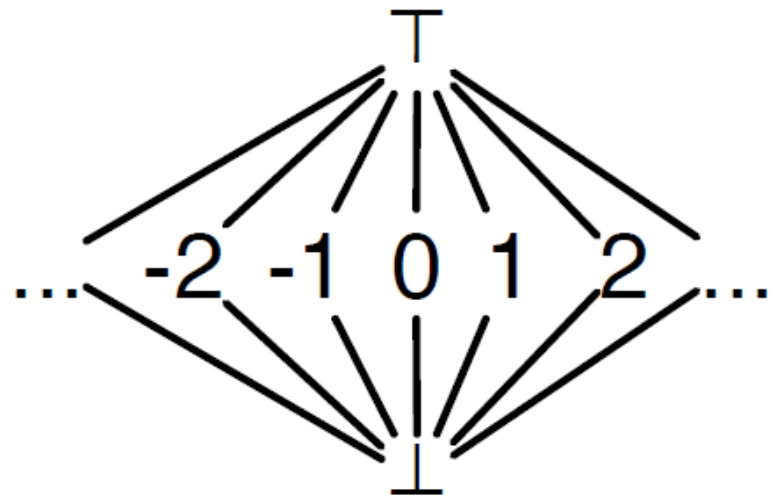
Build CFG

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



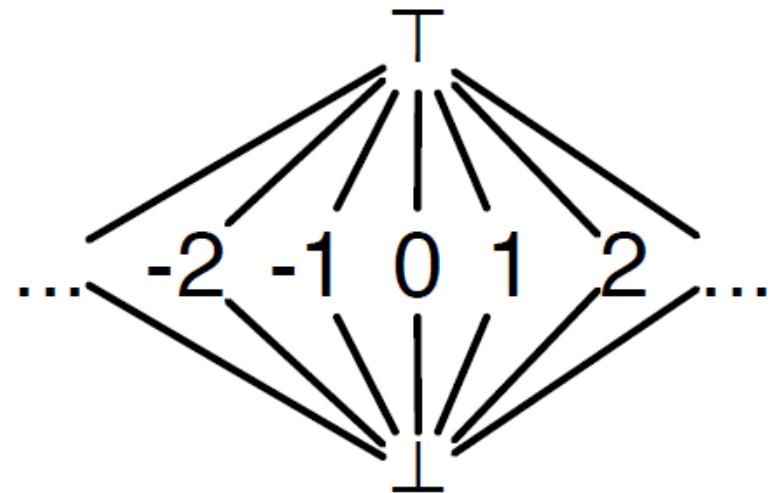
Symbolic evaluation

- 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

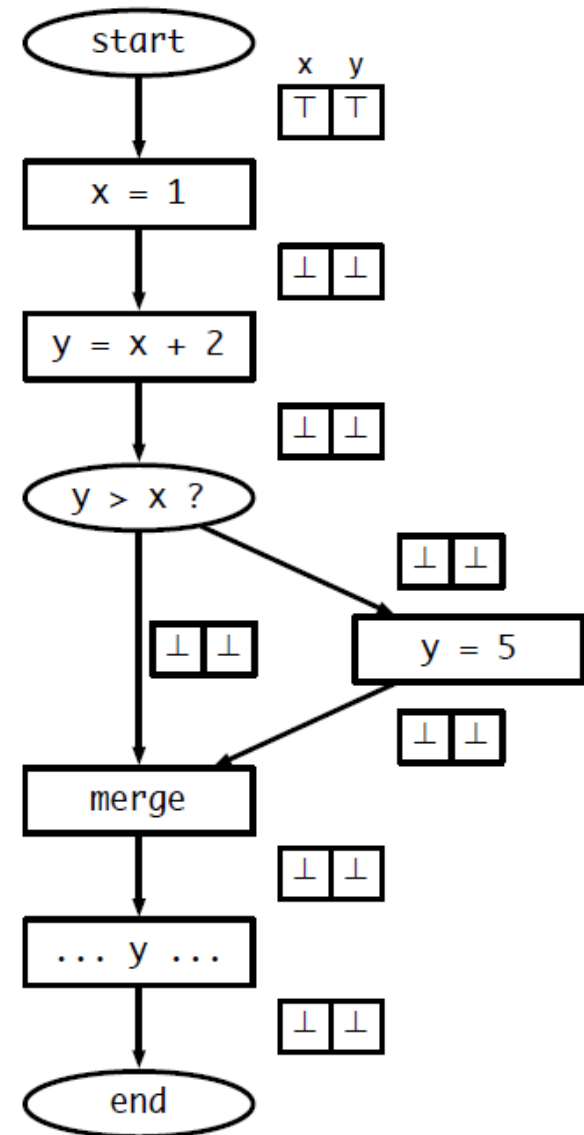
- Evaluate expressions symbolically:
 $\text{eval}(e, V_{\text{in}})$
- If e evaluates to a constant, return that value. If any input is \top (or \perp), return \top (or \perp)
 - Why?
- Two special operations on lattice
 - $\text{meet}(a, b)$ – highest value less than or equal to both a and b
 - $\text{join}(a,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



Join often written as $a \sqcup b$
Meet often written as $a \sqcap b$

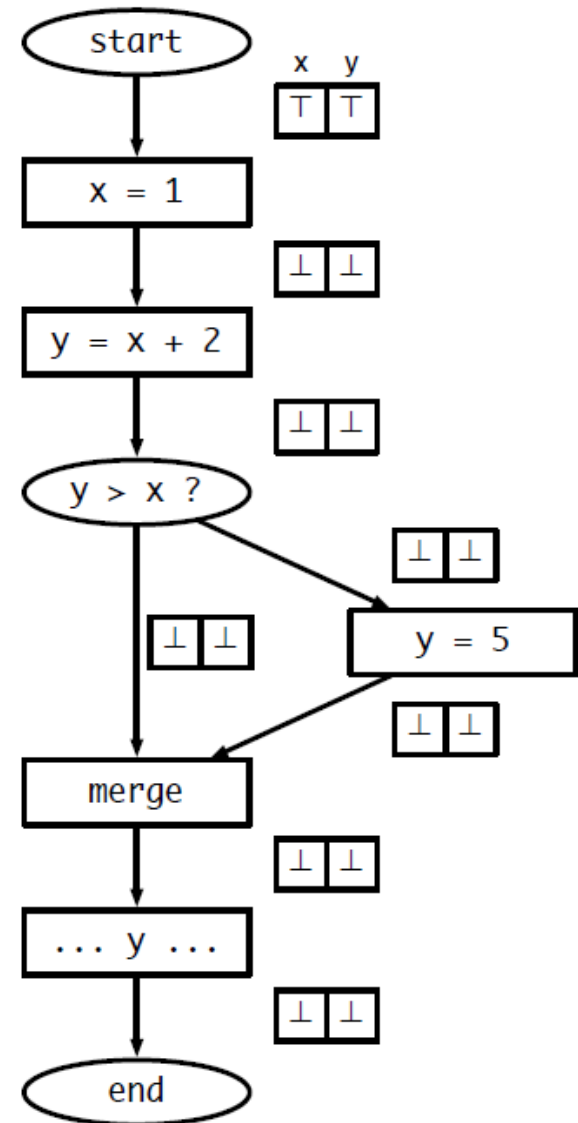
Putting it together

- Keep track of the symbolic value of a variable at every program point (on every CFG edge)
- State vector
- 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 \perp , since we have no information about the variable at that point



Executing symbolically

- 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:
 - $V_1 = \perp, V_2 = * \rightarrow V_{out} = *$
 - $V_1 = \text{constant } x, V_2 = x \rightarrow V_{out} = x$
 - $V_1 = \text{constant } x, V_2 = \text{constant } y \rightarrow V_{out} = \top$
 - $V_1 = \top, V_2 = * \rightarrow V_{out} = \top$
- Generalization:
 - $V_{out} = V_1 \sqcup V_2$

