Dataflow Analysis
(Reaching Definitions, Live Variable Analysis)

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Content

- Recap (with LLVM examples):
  - Generating CFG
  - Printing Dominators
  - Visualizing Domtree
  - Printing Domfrontier of a basic block
  - SSA construction in mem2reg pass

- Dataflow Analysis

SSA

- Converting from unrestricted form to SSA form

```
      i1 = 1
L1:   max = 10
      if i1 < max {
       i2 = i1 + i1
       goto L1;
      }
      print(i);
```

`i1 or i2 ?`

Phi nodes – special instructions that help deal with control flow. E.g.,
`i3 = phi [i1, bb1], [i2, bb4]`

- Where should we insert phi nodes?

  Dominance frontiers

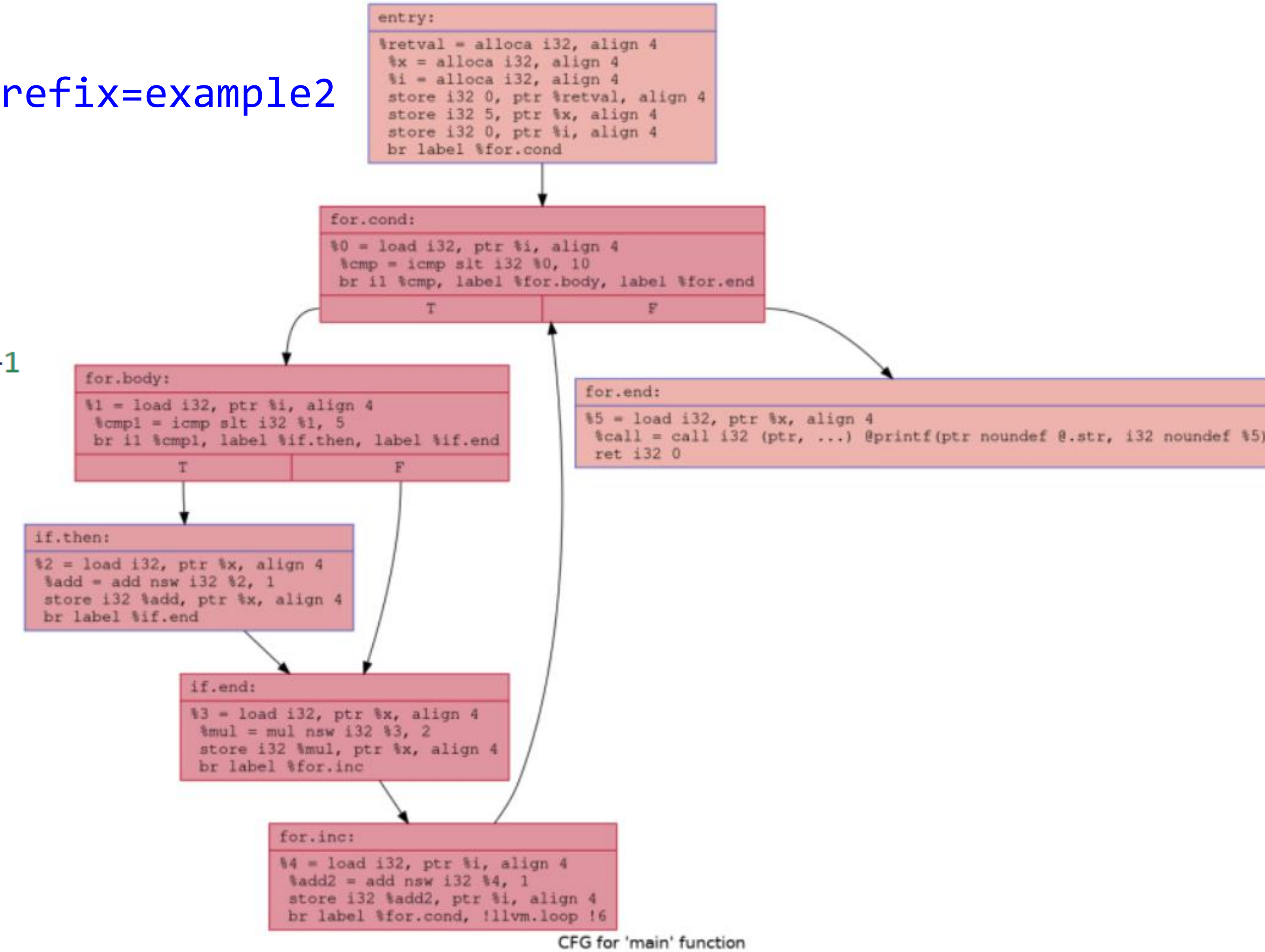
- What do we need to do after inserting phi nodes?

  Rename variables so that every assignment gets a unique name

Flow Graphs - Representation

```
opt -passes=dot-cfg
-cfg-dot-filename-prefix=example2
example2.ll
```

```c
int main() {
    int x, i;
    x = 5;
    for(i=0;i<10;i=i+1
        if (i < 5)
            x = x+1;
        x = x*2;
    }
    printf("%d",x);
    return 0;
}
```



CFG for 'main' function

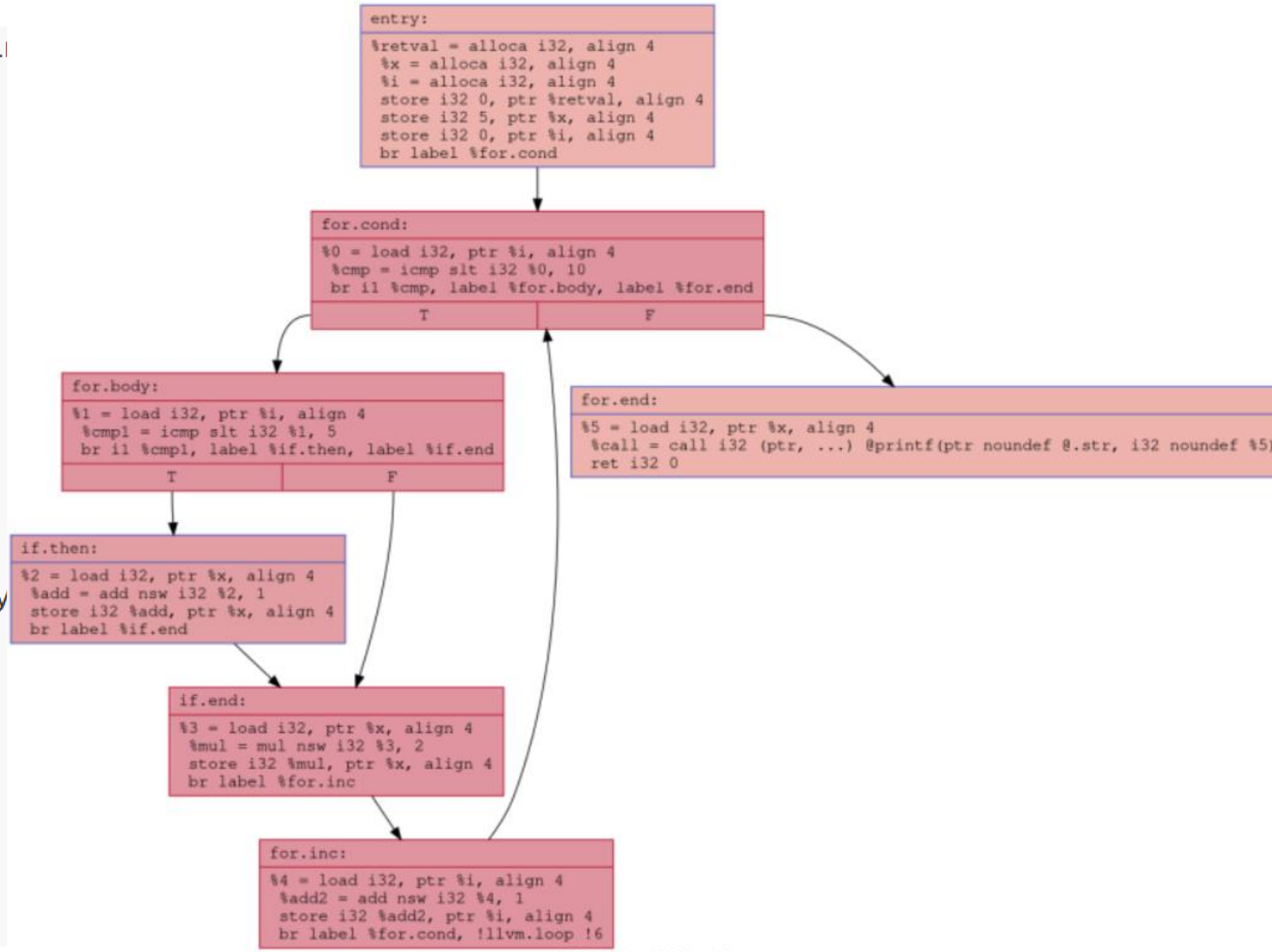
Dominators

- Describe relationship between basic blocks
  - A block dominates other if it is guaranteed to execute before the other
  - **Formally:**
    - if all paths from entry of CFG to node B pass through node A then we say that A dominates B
  - The relationship is reflexive i.e. node B dominates itself

Dominators

opt -load-pass-plugin=./fnmodpass/build/MyPass.so -passes="my-module-pass" -disable-output example2.bc

```
  [ModulePass] Function: mai
entry dominates entry
entry dominates for.cond
entry dominates for.body
entry dominates if.then
entry dominates if.end
entry dominates for.inc
entry dominates for.end
for.cond dominates for.cond
for.cond dominates for.body
for.cond dominates if.then
for.cond dominates if.end
for.cond dominates for.inc
for.cond dominates for.end
for.body dominates for.body
for.body dominates if.then
for.body dominates if.end
for.body dominates for.inc
if.then dominates if.then
if.end dominates if.end
if.end dominates for.inc
for.inc dominates for.inc
for.end dominates for.end
```

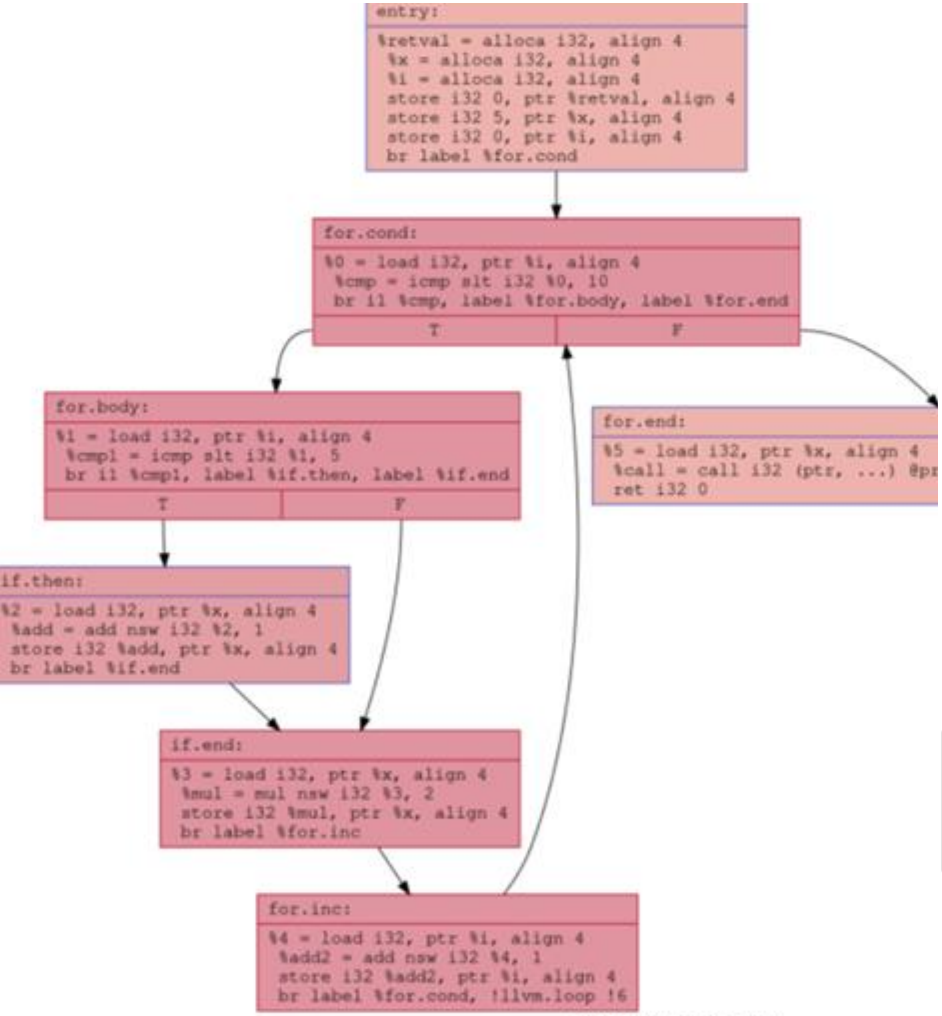


CFG for 'main' function

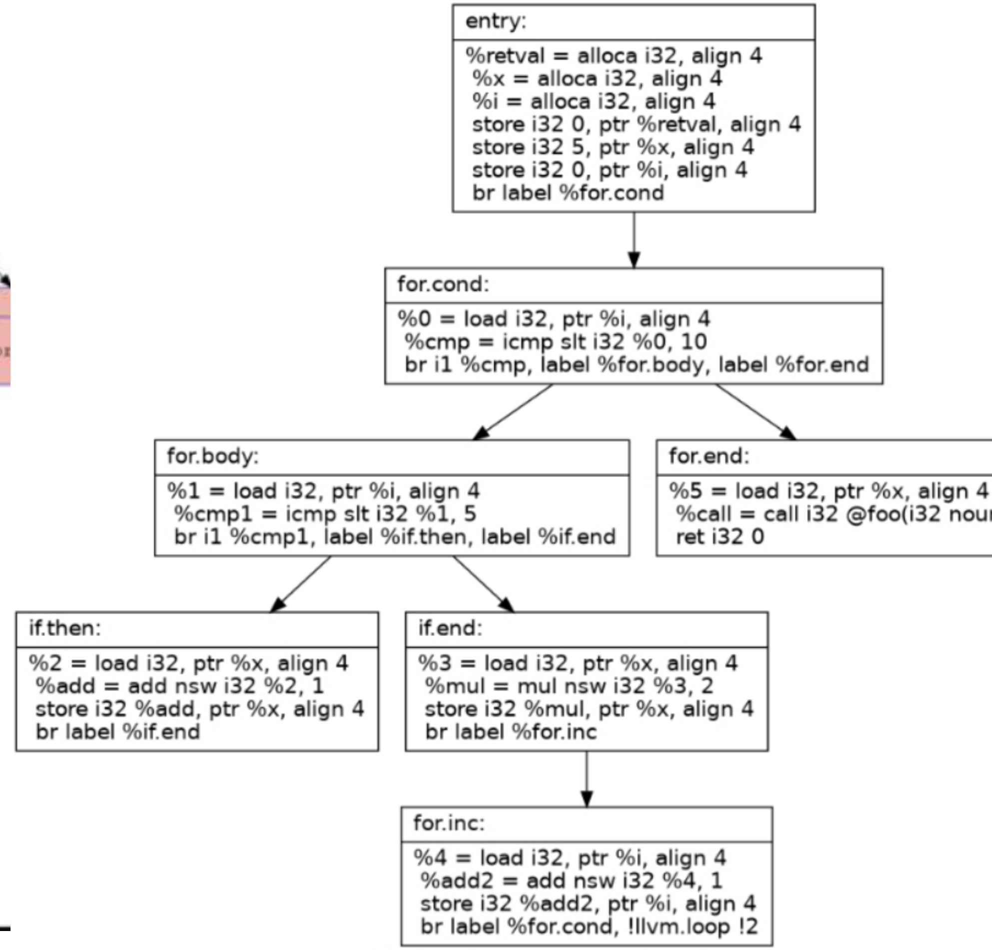
Dominator Tree

- A data structure for tracking dominator relation
- A node in the tree dominates all the nodes of the subtree, for which the node is the root
- Terminology:
  - Strict domination: A strictly dominates B if it dominates B and A ≠ B
  - Immediate domination: A dominates B and does not strictly dominate any other node that strictly dominates B (e.g. A is B's parent in the dominator tree)
  - Domination frontier (DF): B is in the DF of A if
    - A does not dominate B but
    - dominates a predecessor of B
  - Post domination: A post-dominates B if on *all* paths from B to the exit node, A appears.

Dominator Tree



CFG for 'main' function



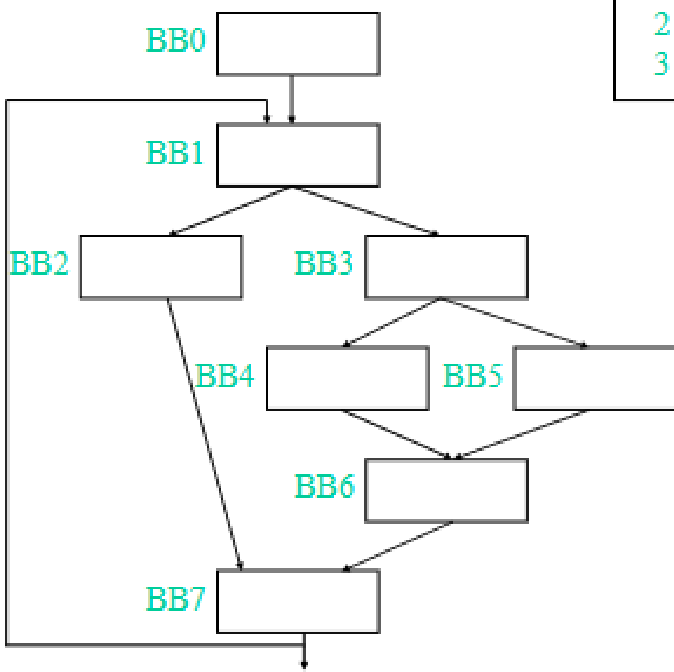
Dominator tree for 'main' function

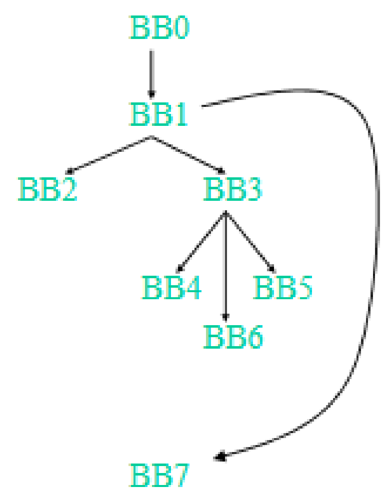
```
opt -passes=dot-dom example2.ll
dot -Tpng dom.main.dot -o example2_dom.png
```

Dominator Tree

First BB is the root node, each node dominates all of its descendants

BB	DOM	BB	DOM
0	0	4	0,1,3,4
1	0,1	5	0,1,3,5
2	0,1,2	6	0,1,3,6
3	0,1,3	7	0,1,7





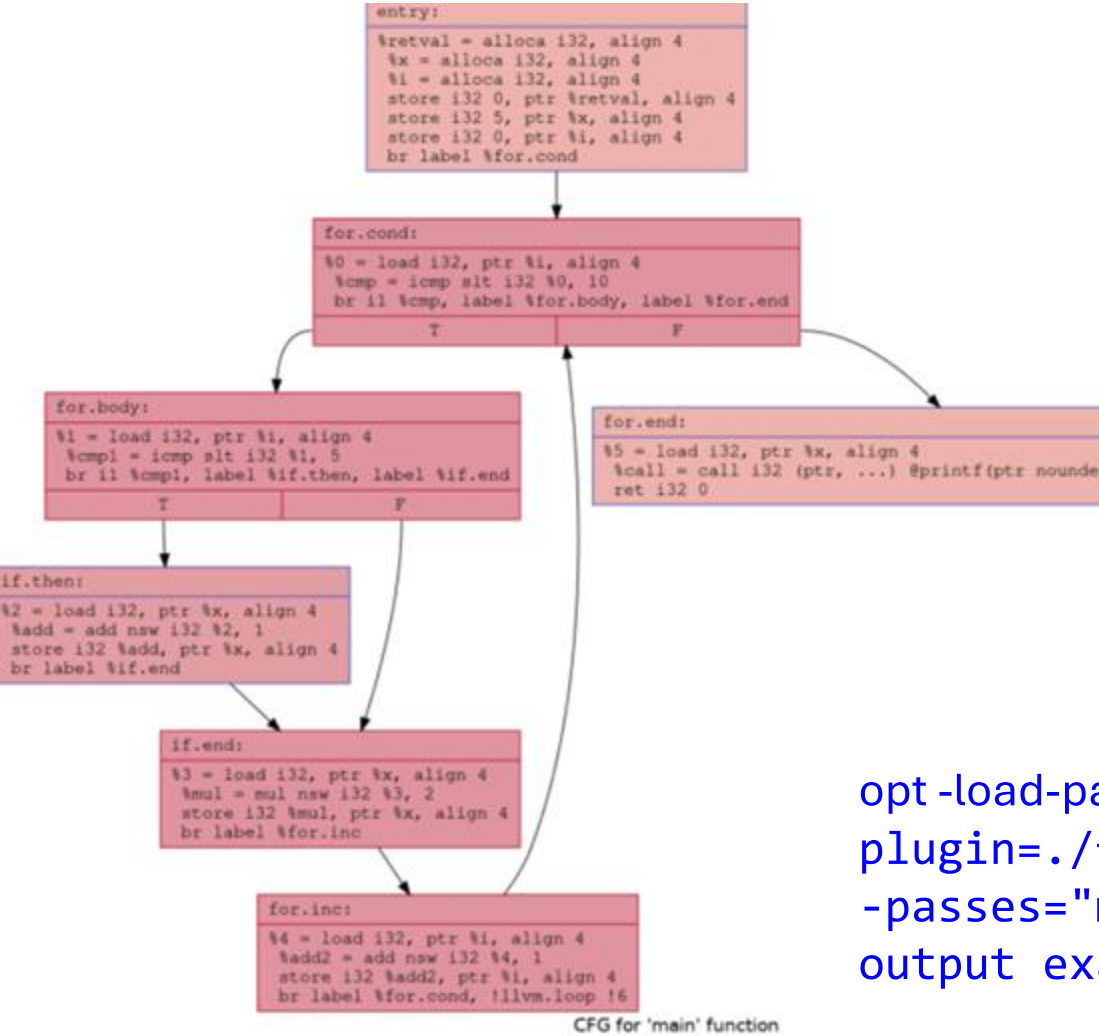
Dom tree

Finding Dominance Frontiers

Recall:

- Dominance frontier of a node X is the set of nodes Y such that
  - X dominates a predecessor of Y
  - X does not strictly dominate Y

Finding Dominance Frontiers

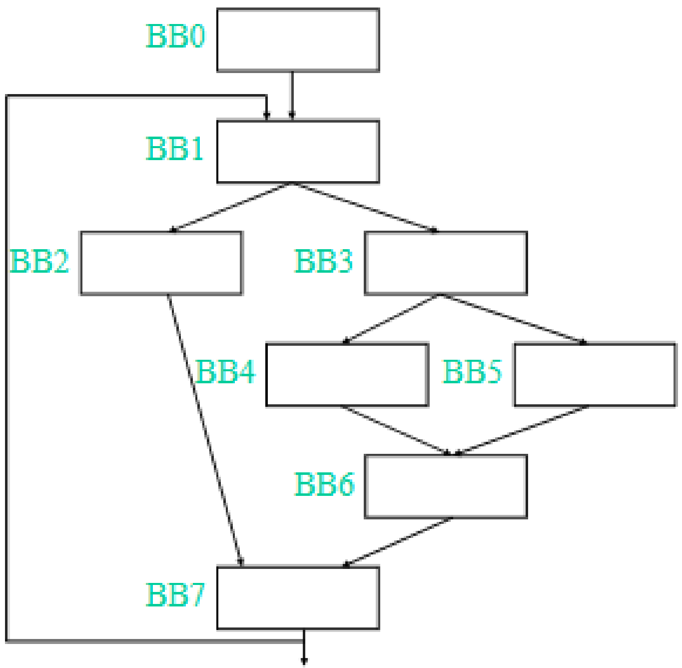


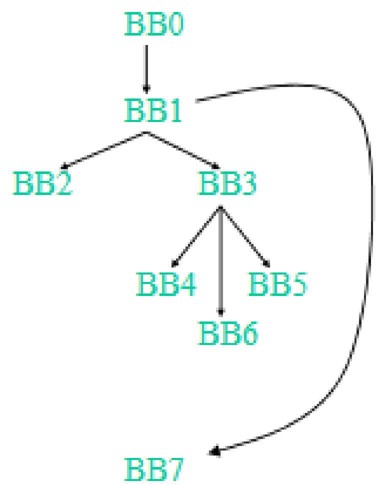

```
 [ModulePass] Function: main
Function: main
Dominance frontier of Block entry:
 empty
Dominance frontier of Block for.cond:
        for.cond
Dominance frontier of Block for.body:
        for.cond
Dominance frontier of Block if.then:
        if.end
Dominance frontier of Block if.end:
        for.cond
Dominance frontier of Block for.inc:
        for.cond
Dominance frontier of Block for.end:
 empty
```

opt -load-pass-plugin=./fnmodpass/build/MyPass.so -passes="my-module-pass" -disable-output example2.bc

Note: opt –passes=domtree …
does not work as domtree is an internal pass

Computing Dominance Frontiers





BB	DF
0	-
1	-
2	7
3	7
4	6
5	6
6	7
7	1

For each join point X in the CFG
For each predecessor of X in the CFG
Run up to the IDOM(X) in the dominator tree,
adding X to DF(N) for each N between X and IDOM(X)

Converting to SSA form

- Where do we insert phi nodes?

```
for v in vars {
    for d in defs[v] {
        for block in DF[d] {
            if (block does not have a phi node)
                add phi node to block
            if (block is not part of defs[v])
                add block to defs[v]
        }
    }
}
```

```
clang -O0 -emit-llvm -S ex3.c -o ex3.ll
```

```c
int test(int a, int b, _Bool c) {
    int res;
    if (c)
        res = a;
    else
        res = b;
    return res;
}
```

```llvm
entry:
  %x = alloca i32
  br i1 %cond, label %then, label %else

then:
  store i32 %a, i32* %x
  br label %merge

else:
  store i32 %b, i32* %x
  br label %merge

merge:
  %val = load i32, i32* %x
  ret i32 %val
}
```

after **mem2reg** pass:

```
opt -passes=mem2reg ex3.ll -S -o ex3_mem2reg.ll
```

```llvm
entry:
  br i1 %cond, label %then, label %else

then:                                             ; preds = %entry
  br label %merge

else:                                             ; preds = %entry
  br label %merge

merge:                                            ; preds = %else, %then
  %x.0 = phi i32 [ %a, %then ], [ %b, %else ]
  ret i32 %x.0
}
```

Try it yourself

- Generate CFG (in png format)
- Use the code provided and modify to print *immediate dominators* for each basic block
- Generate domtree (in png format)
- Print dominance frontier of each block

Do your observations coincide with those presented in slides?

Dataflow Analysis – Motivation

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case I: We can move `t = a op b` if the statement <u>dominates</u> 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
```

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case II: We can move `t = a op b` if there is only one definition of `t` in the loop

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

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case III: We can move `t = a 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
```

Dataflow Analysis – More motivation

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 <u>live</u> at 1

<u>..used in future</u>

- 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`?

Choose the statements that are true with reference to the code snippet shown. Assume that this is a basic block (a sequence of statements) and this basic block is a part of a CFG.

1. X:=1
2. X:=4
3. Y:=X

a ☐ X is definitely dead at statement 1 (value of X is never used again)

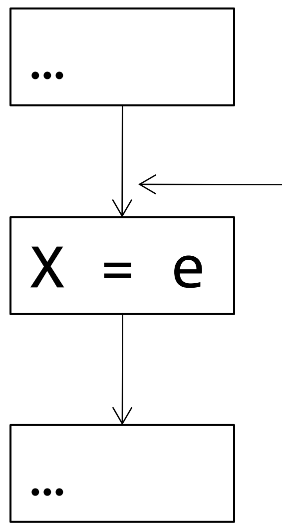
b ☐ X is definitely dead at statement 2

c ☐ X is live at statement 2

d ☐ Y must be live at statement 3

e ☐ None

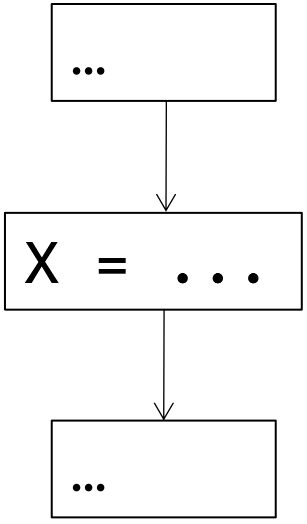
Liveness in a CFG



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

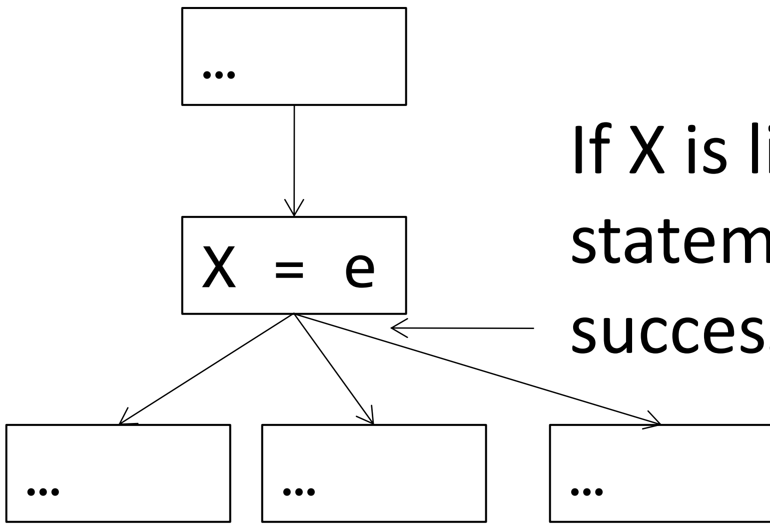
Liveness in a CFG



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

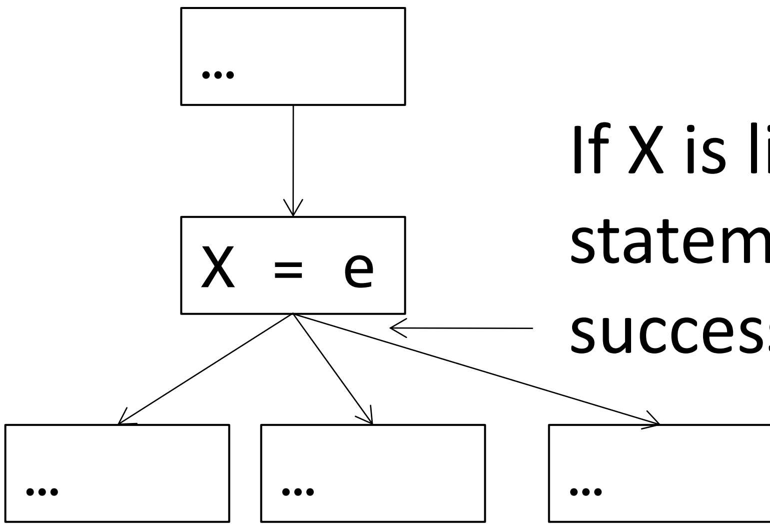
Liveness in a CFG



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

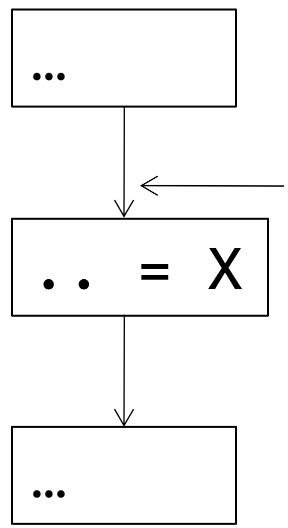


If X is live here (i.e. after the statement), X is used in *some* successor

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

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

Liveness in a CFG

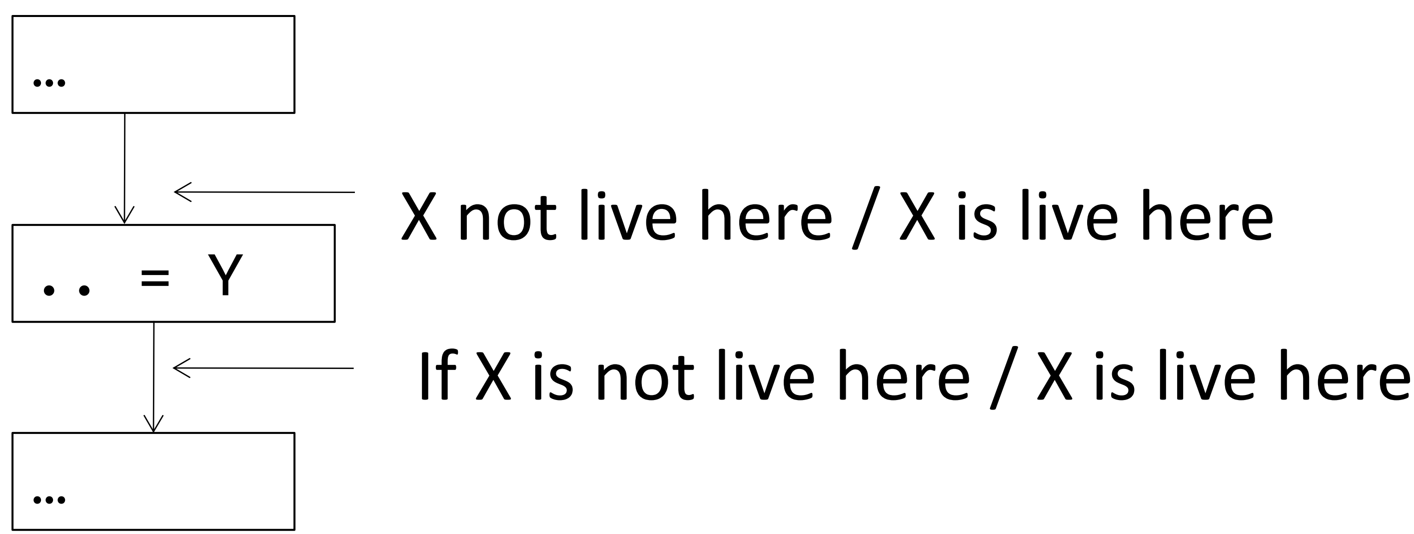


X *must be* live here (i.e. before the statement)

- 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)` $\supseteq$ `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

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

- 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

```
LiveIn(b)⊇ LiveOut(b) – 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

$$\texttt{LiveIn(b)} \supseteq \texttt{LiveOut(b) – Def(b)}$$

- 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

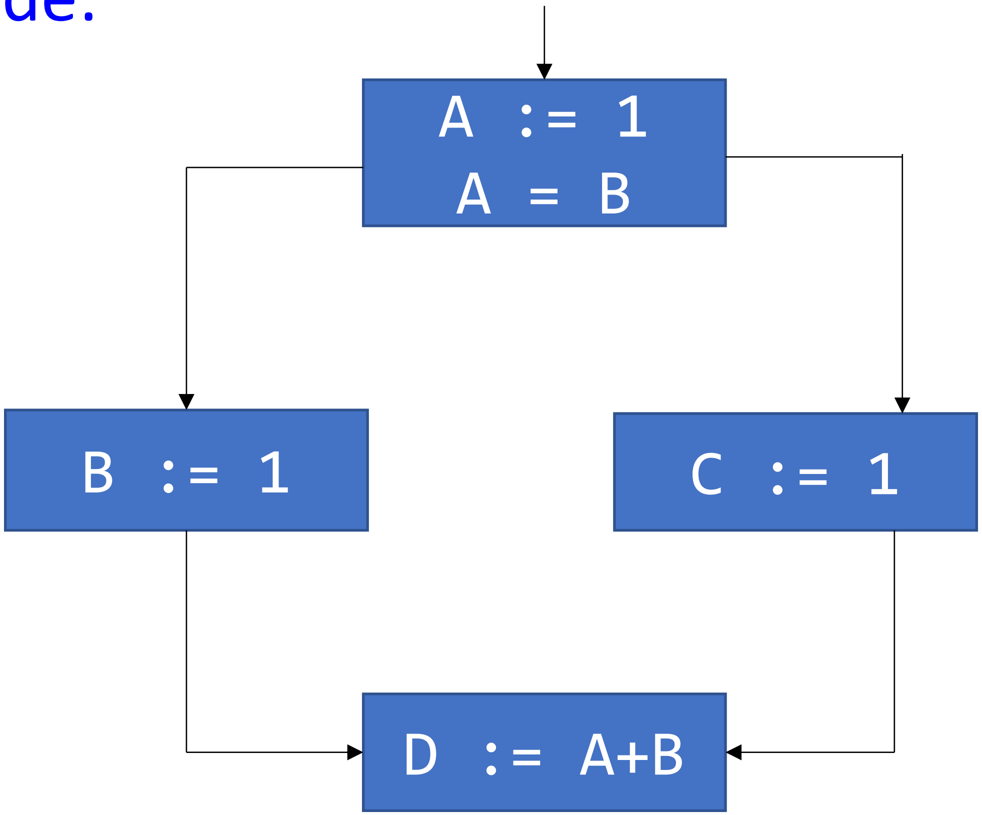
- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block
  - OR the variable is live at exit and not defined within the block

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

Liveness in a CFG - Example

- Draw CFG for the code:

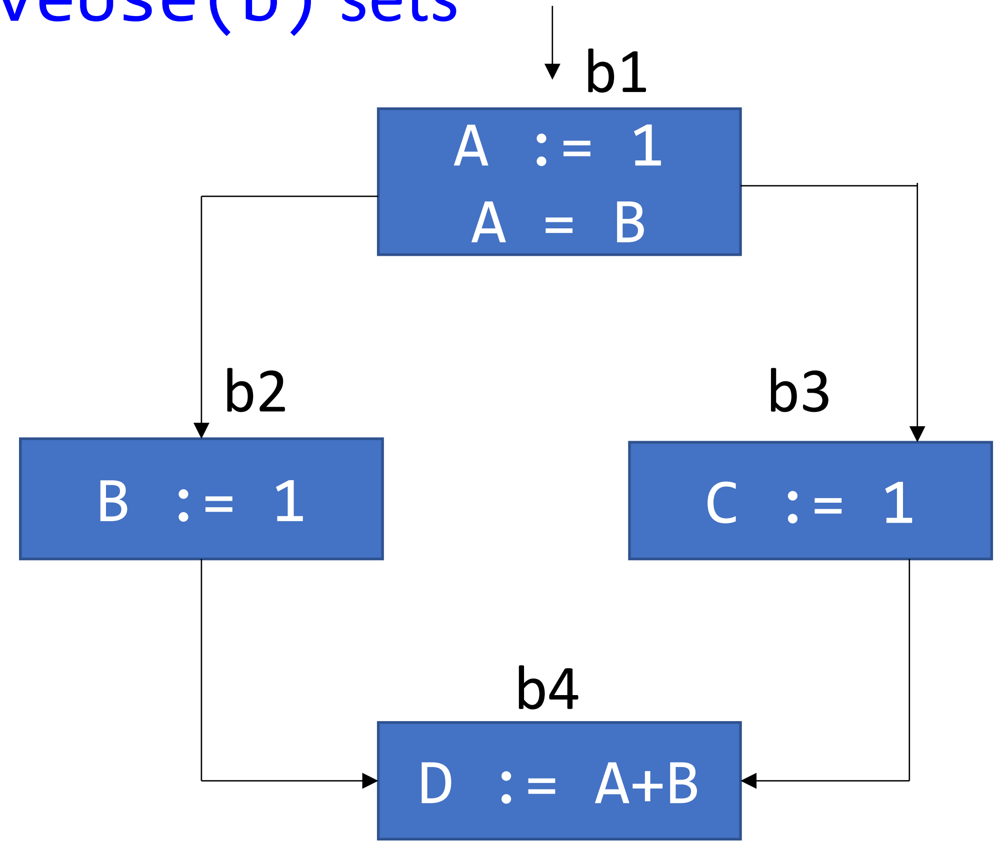
```
A:=1
if A=B then
   B:=1
else
   C:=1
endif
D:=A+B
```



Liveness in a CFG - Example

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

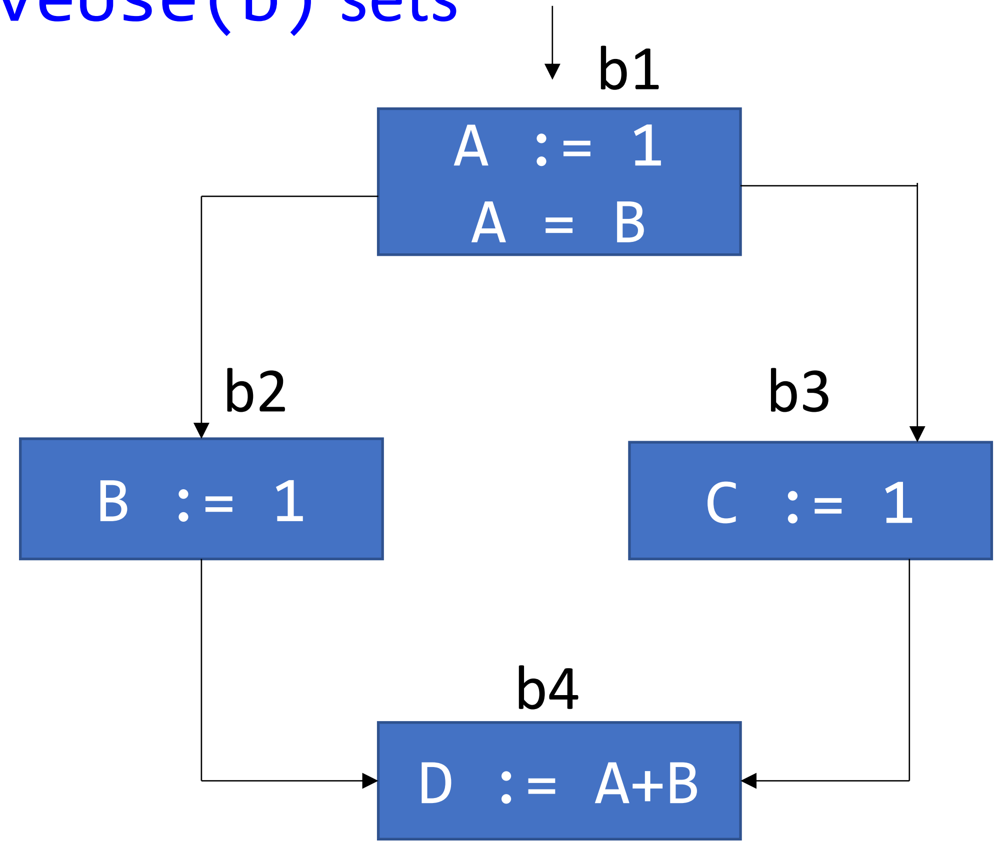
Block	Def	LiveUse
b1		
b2		
b3		
b4		



Liveness in a CFG - Example

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

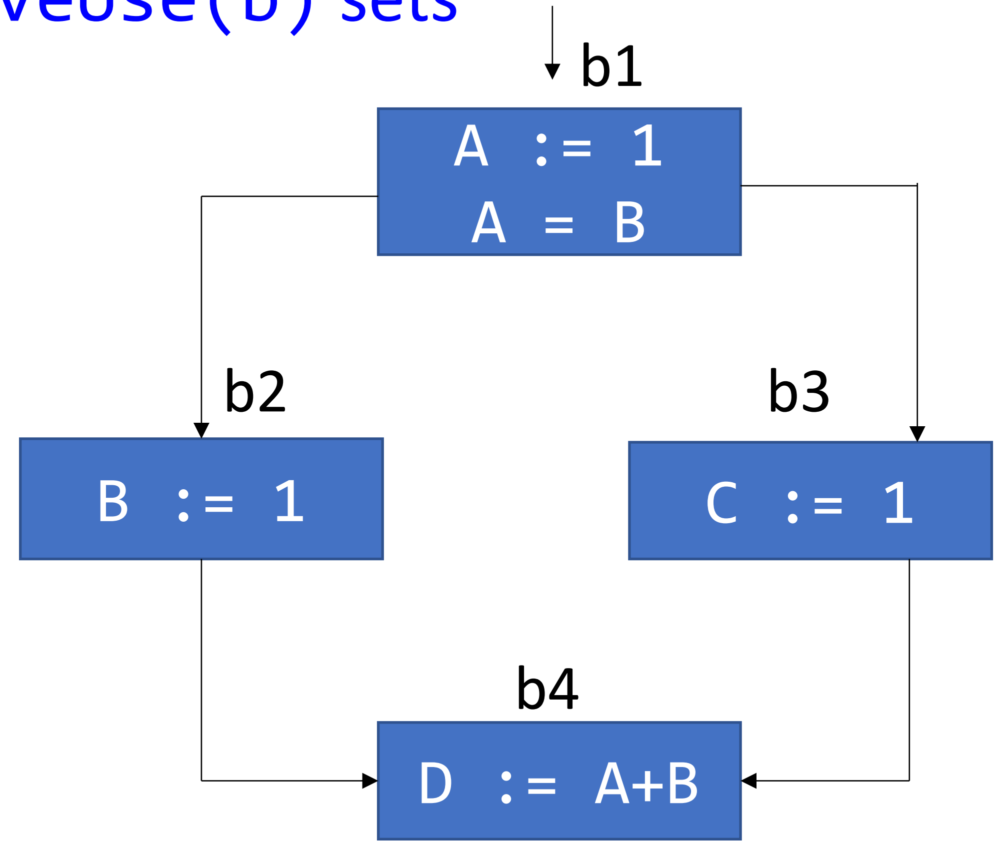
Block	Def	LiveUse
b1	{A}	{B}
b2		
b3		
b4		



Liveness in a CFG - Example

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

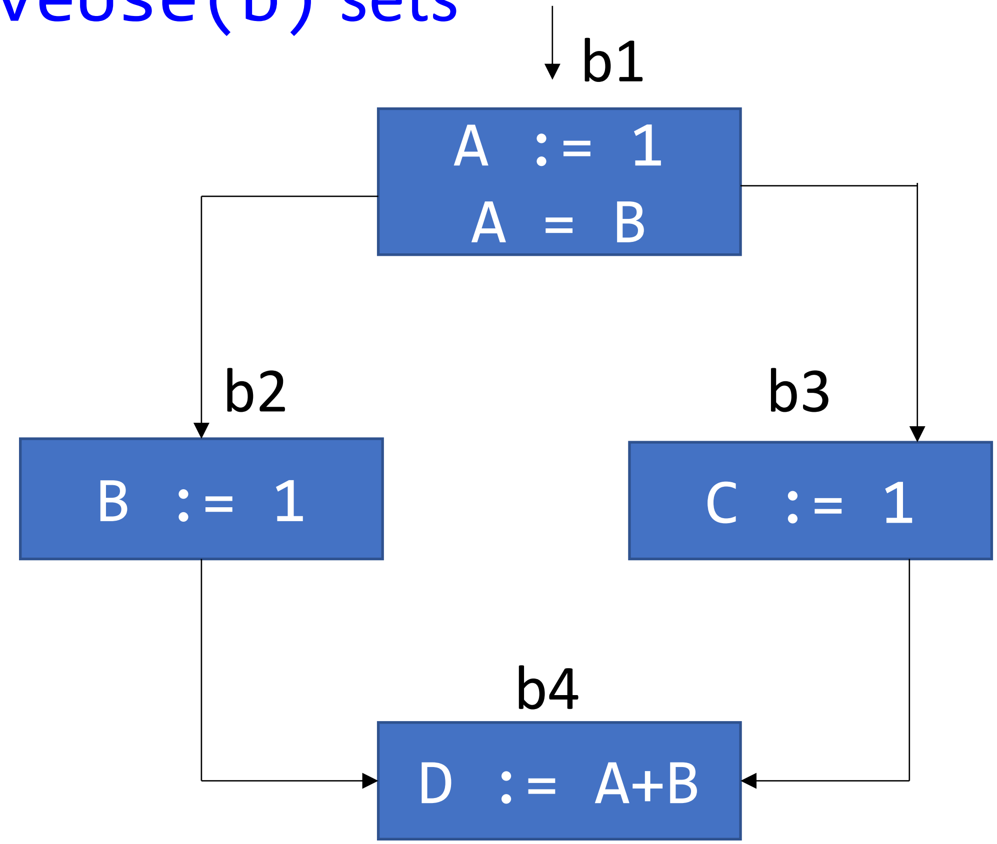
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3		
b4		



Liveness in a CFG - Example

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

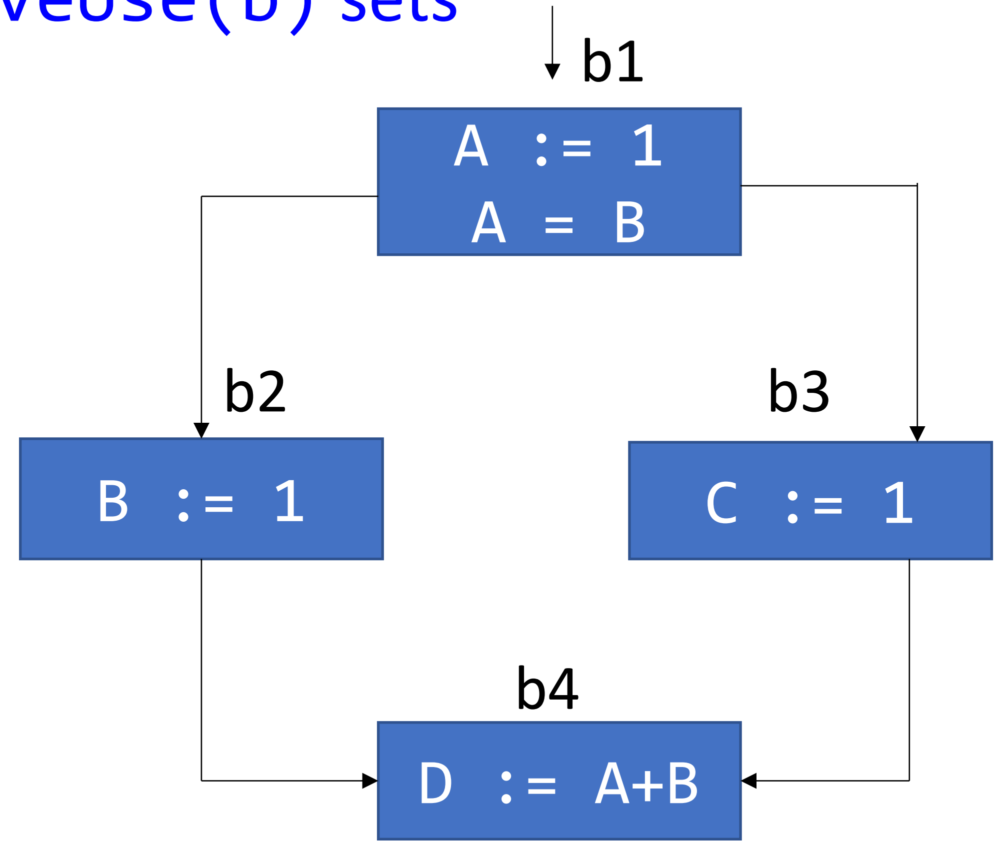
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4		



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}

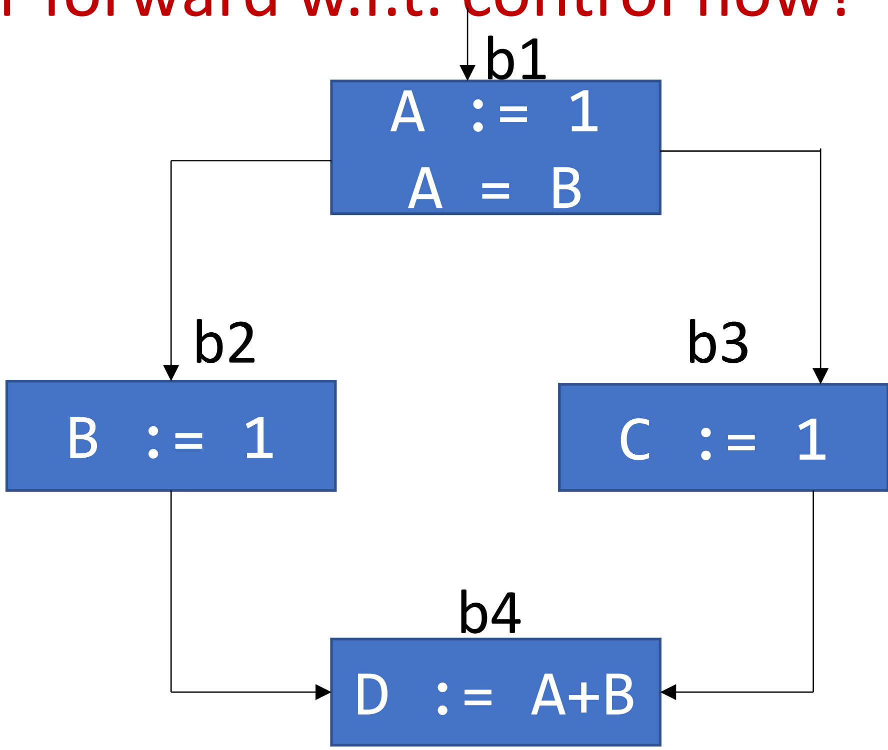


Liveness in a CFG - Example

- 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}

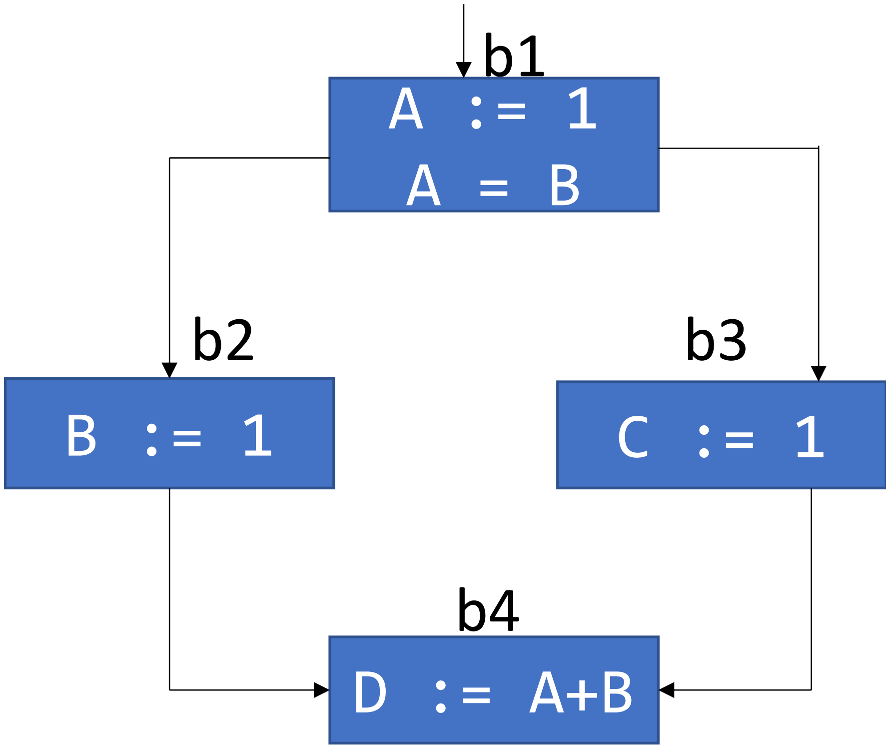


Liveness in a CFG - Example

- 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}

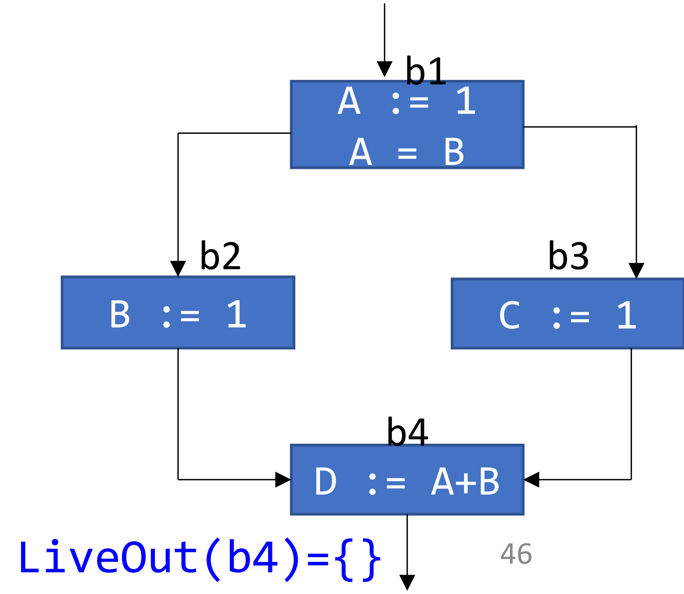


Liveness in a CFG - Example

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

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

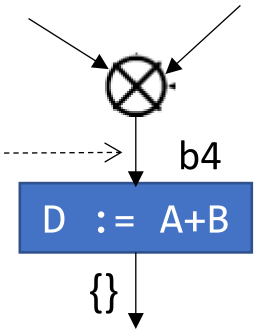
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}



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}

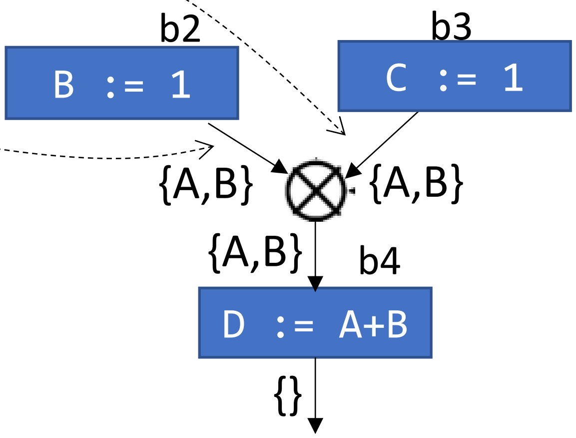
Liveness in a CFG - Example

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

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

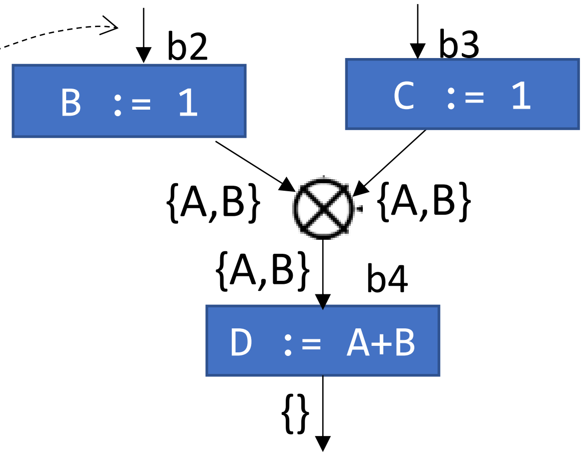
`LiveOut(b2) = LiveIn(b4) = {A,B}`

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



Liveness in a CFG - Example

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



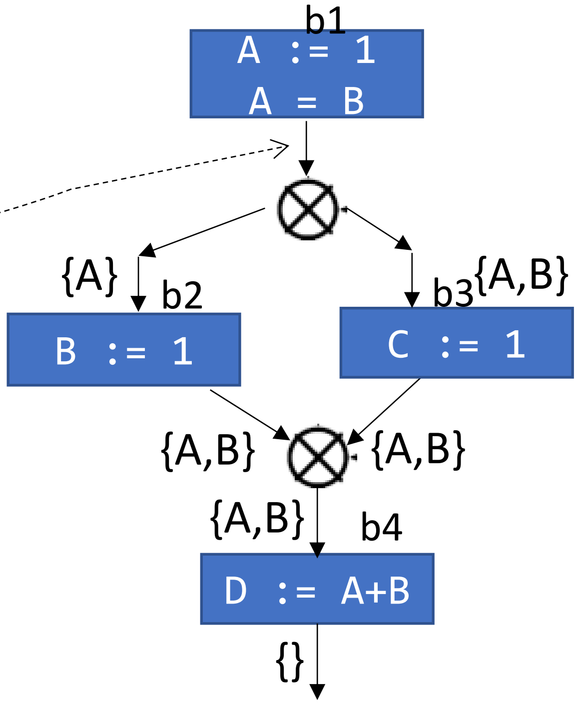
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Liveness in a CFG - Example

$\texttt{LiveOut(b)} = \cup_{i\, \in S(b)}\, \mathrm{LiveIn}(i)$

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

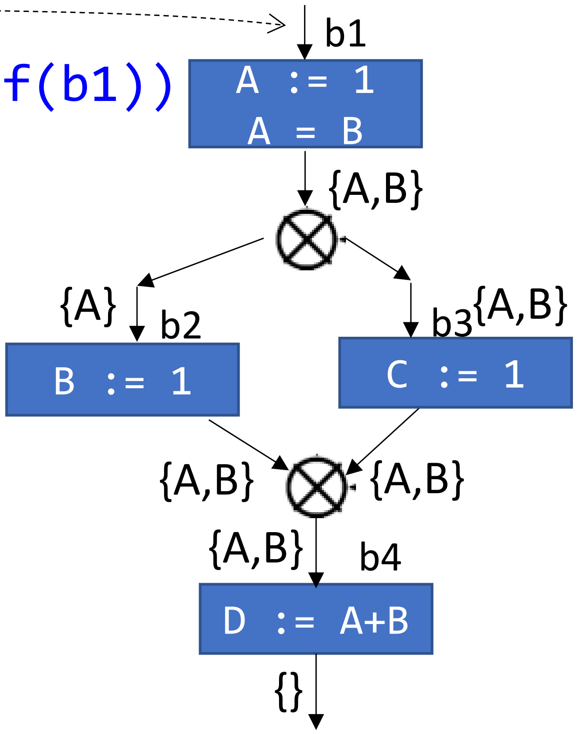
$= \{A\} \cup \{A,B\} = \{A,B\}$



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

Liveness in a CFG - Example

```
LiveIn(b1) = LiveUse(b1) U  (LiveOut(b1) – Def(b1))
           = {B} U  ({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)`

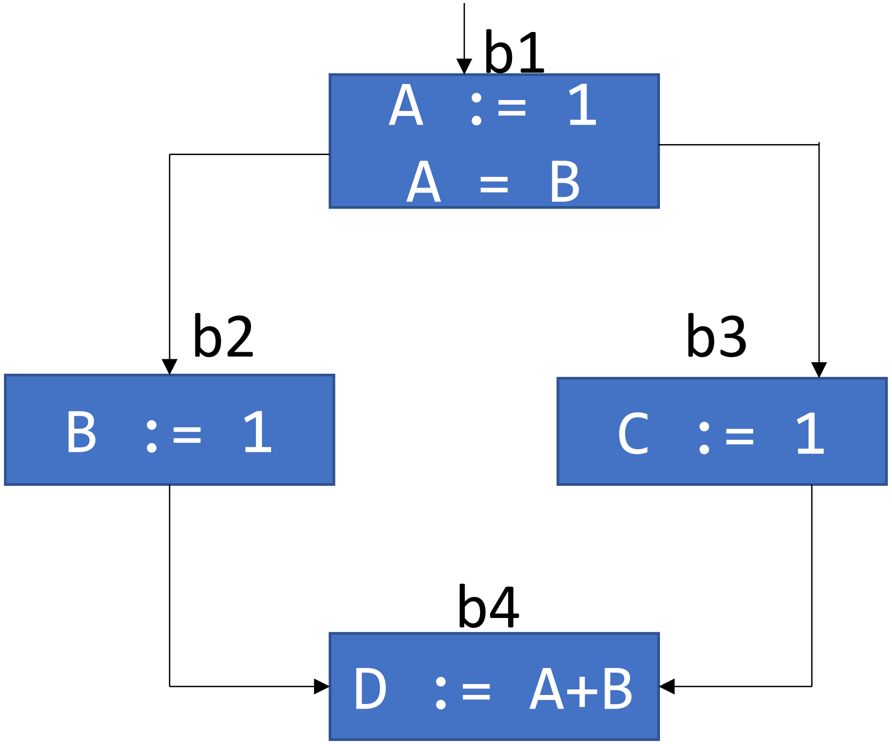
```
LiveIn(b) = LiveUse(b) U  (LiveOut(b) – Def(b))
```

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

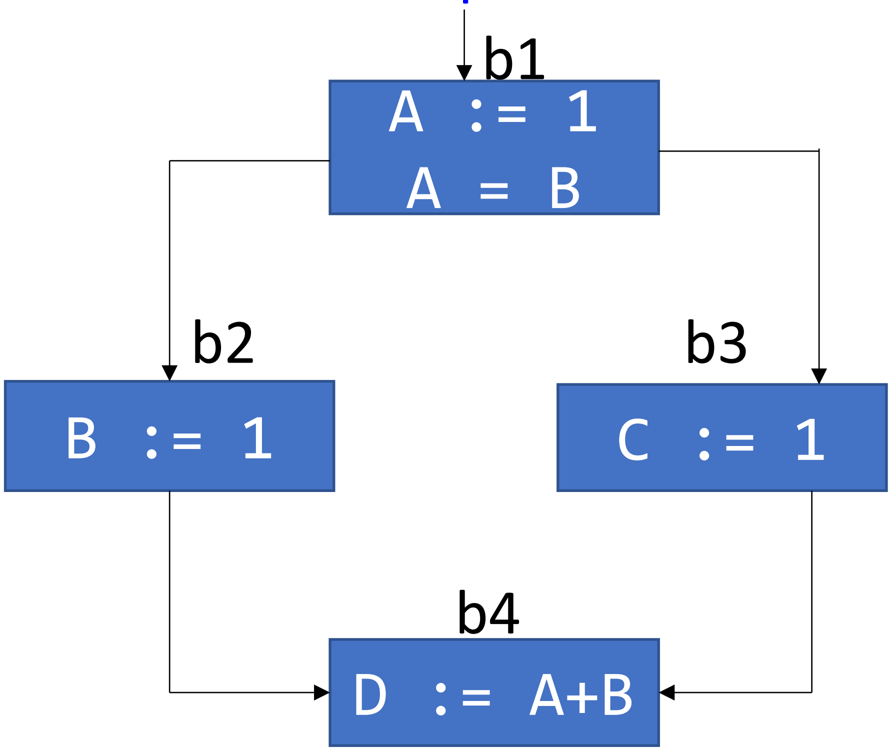
- Assume that the CFG below represents *your entire program* (b1 is the entry to program and b4 is the exit)
    - What can you infer from the table?



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
```

⇨

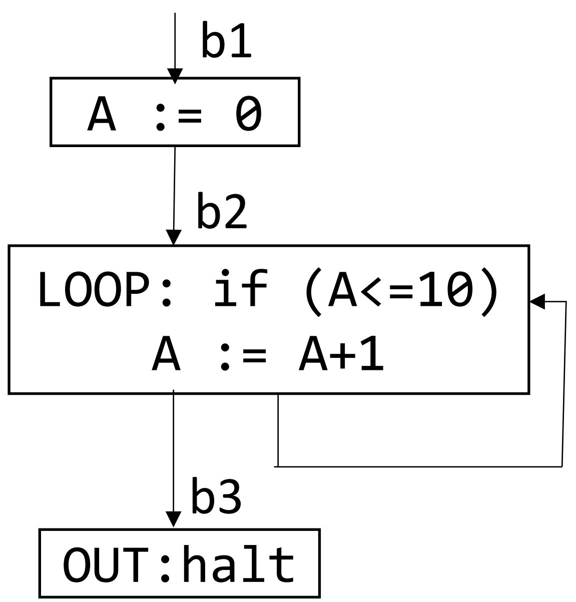
```
X = 1
Y = 1 + 2
Z = Y + A
```

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	{}	{}

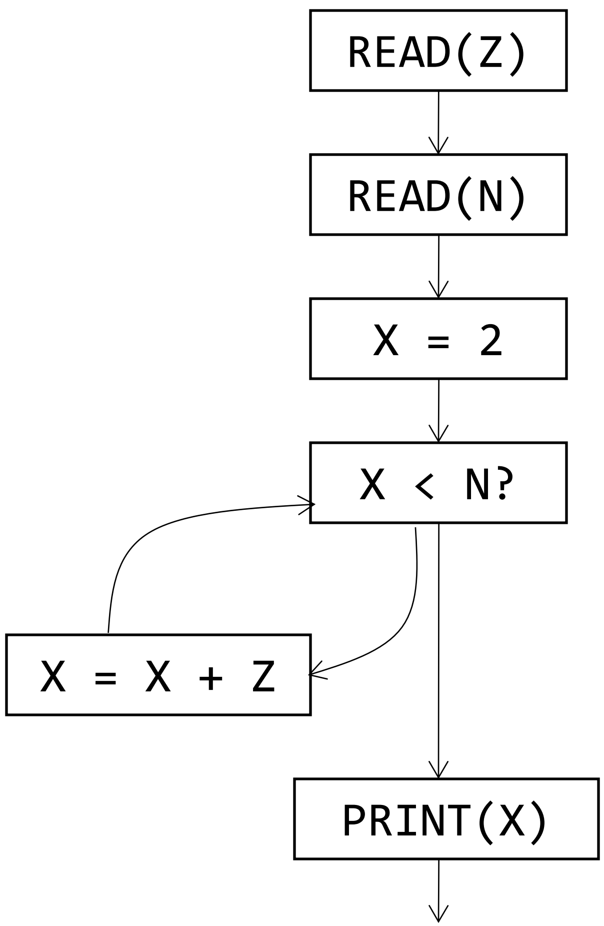
Liveness in a CFG - Observations

- 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

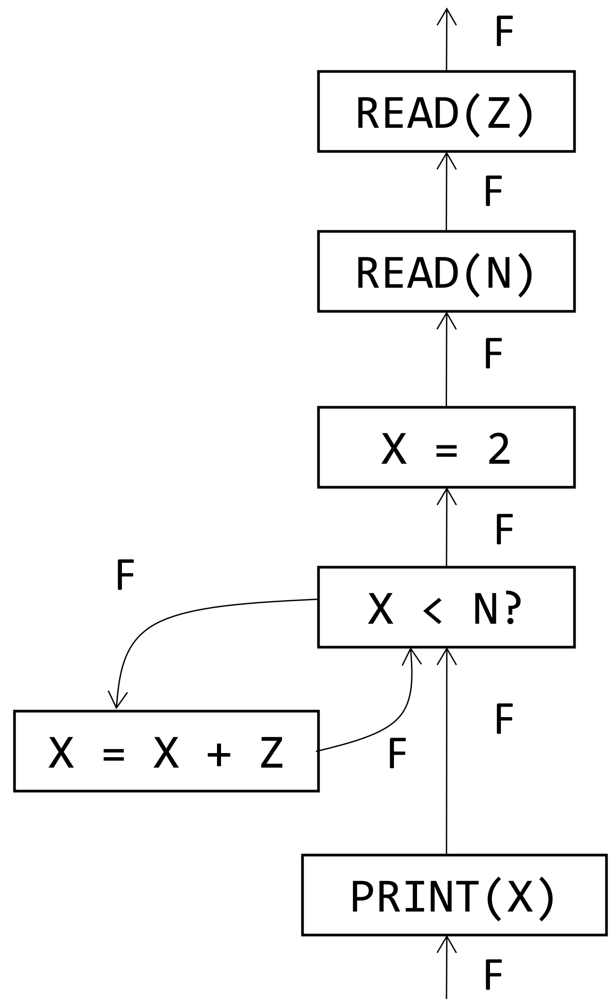
<u>At a program point</u> can the liveness information change?

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

Recap: Liveness

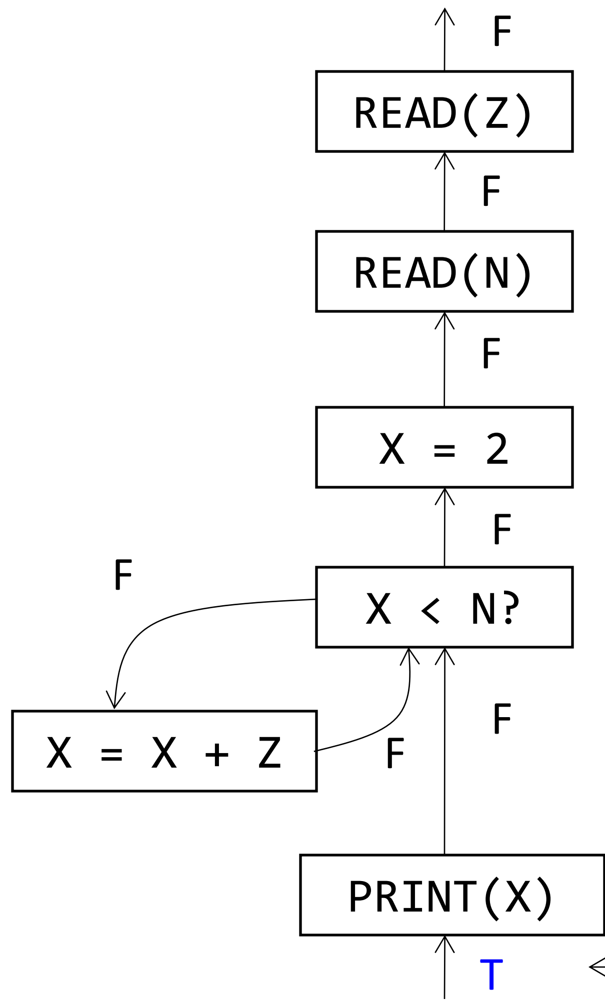


Original CFG



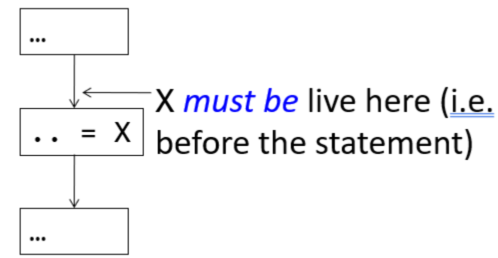
CFG with edges reversed (and initialized) for backwards analysis: is X live? (F=`false`, T=`true`)

Recap: Liveness



X must be live here (why?)

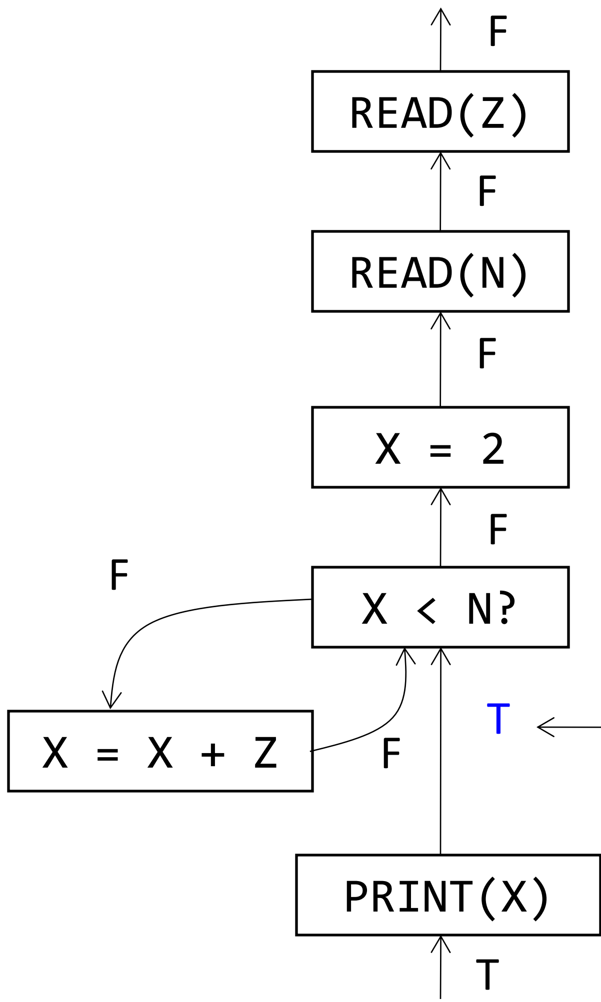
Liveness in a CFG



- 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)`

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

Recap: Liveness



X must be live here (why?)

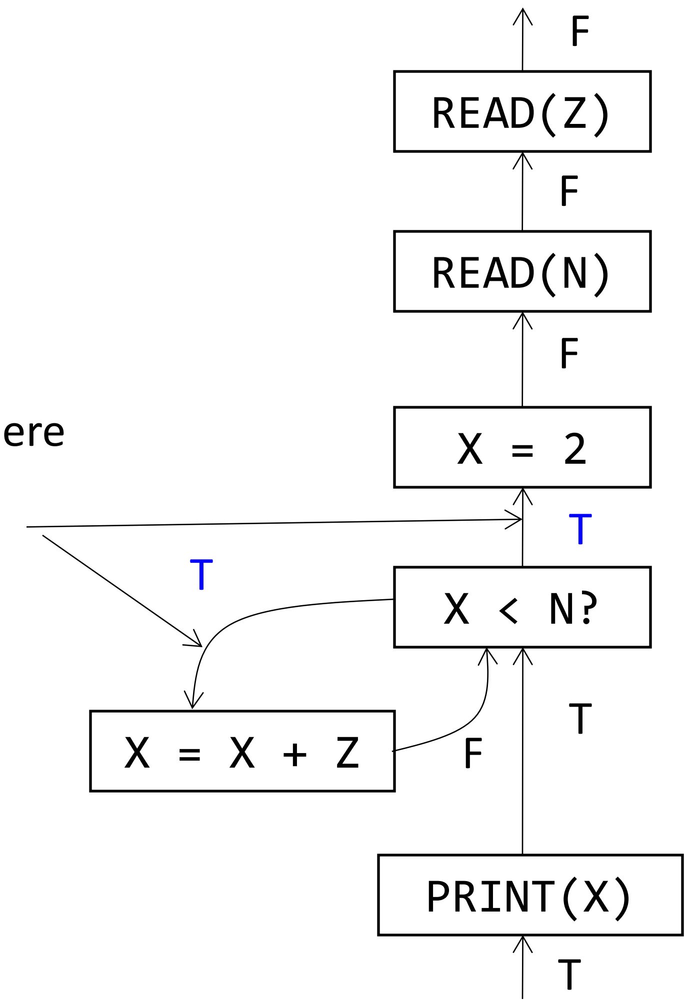
Liveness in a CFG

- 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

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

CS406, IIT Dharwad 45

Recap: Liveness



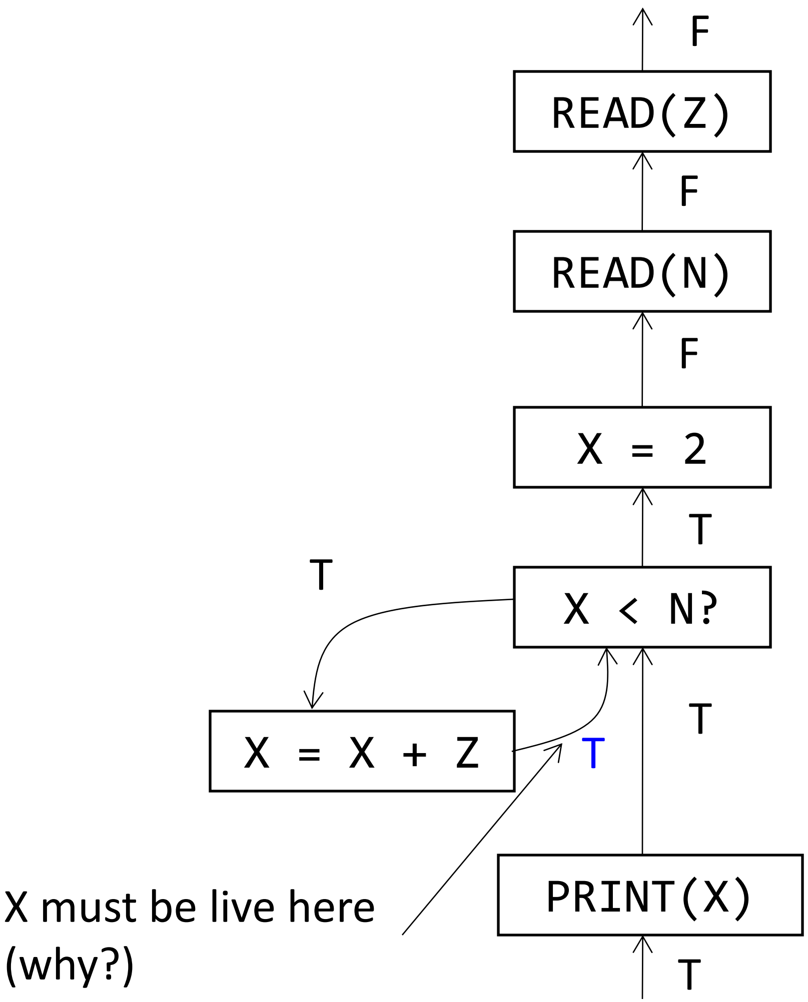
X must be live here (why?)

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

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

Recap: Liveness

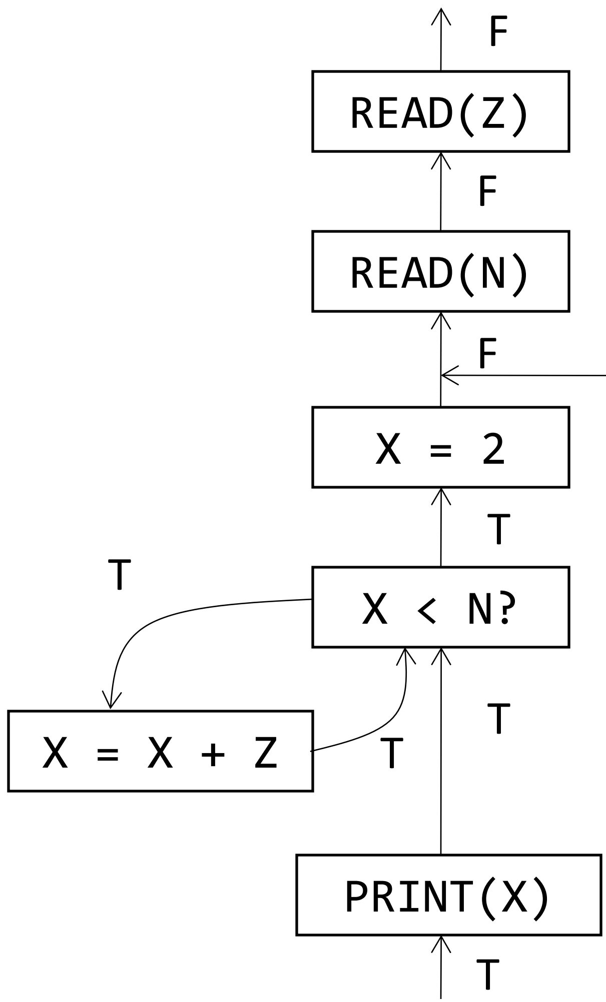


Liveness in a CFG

- 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

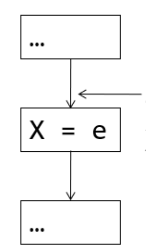
`LiveIn(b) = LiveUse(b) ∪ (LiveOut(b) – Def(b))`

Recap: Liveness



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

Liveness in a CFG

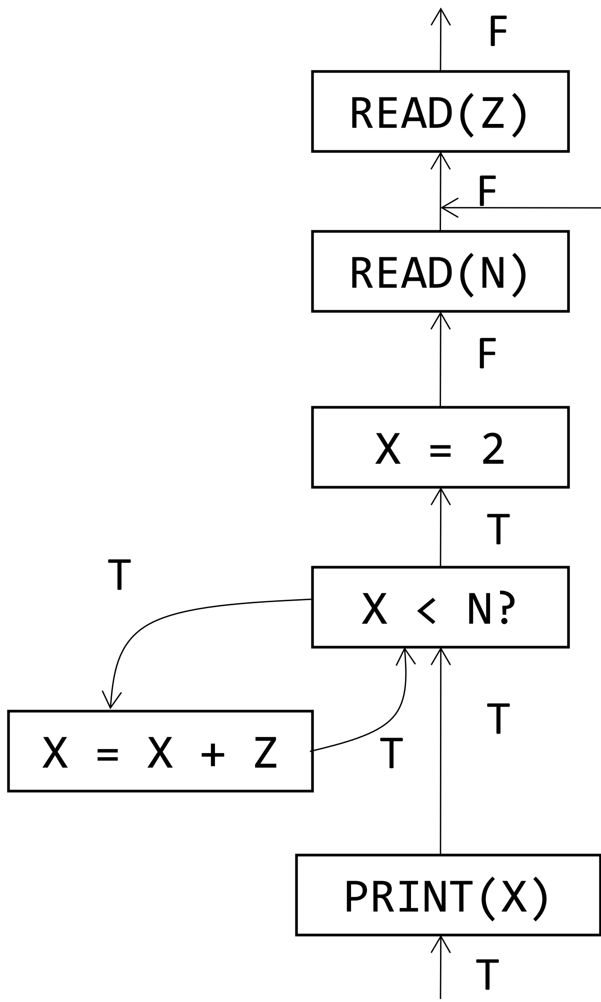


Given that e does not use `X`, X is ***definitely dead*** here (i.e. before the statement).

- 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

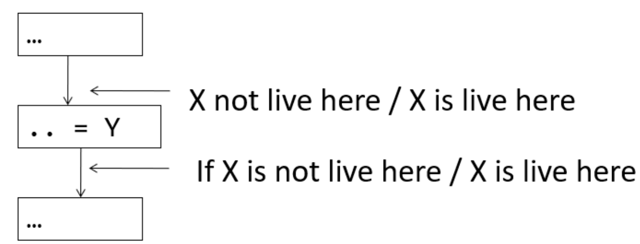
CS406, IIT Dharwad 36

Recap: Liveness



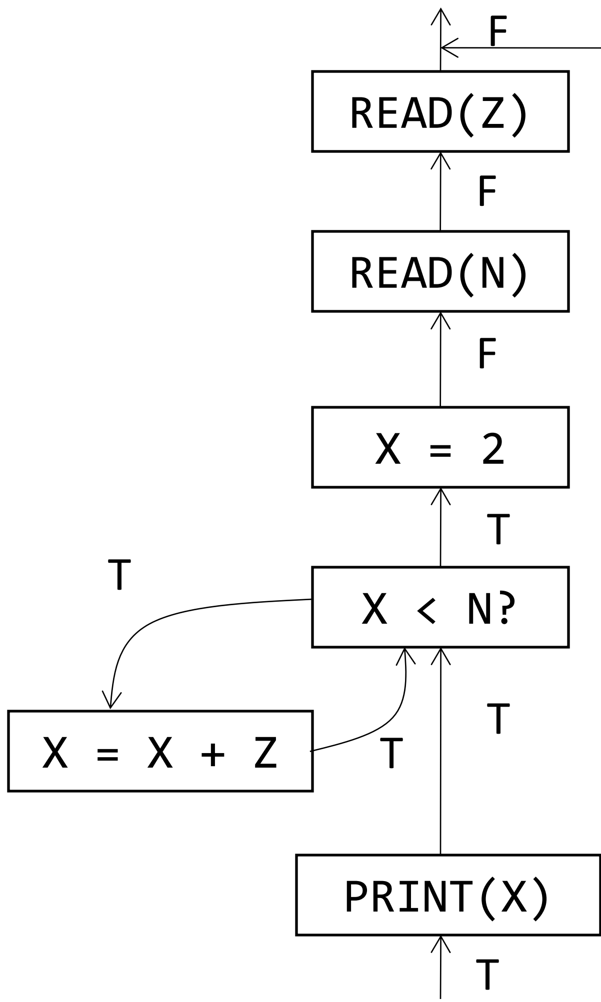
X dead here (why?). No change in information.

Liveness in a CFG - Observation



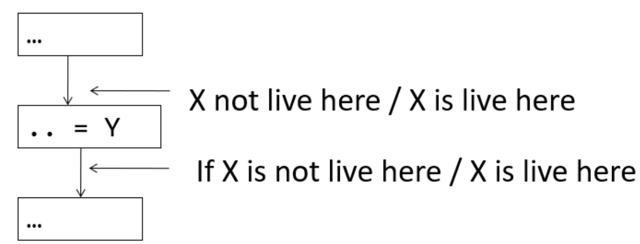
- If a node neither uses nor defines X, the liveness property remains the same before and after executing the node

Recap: Liveness



X dead here (why?). 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

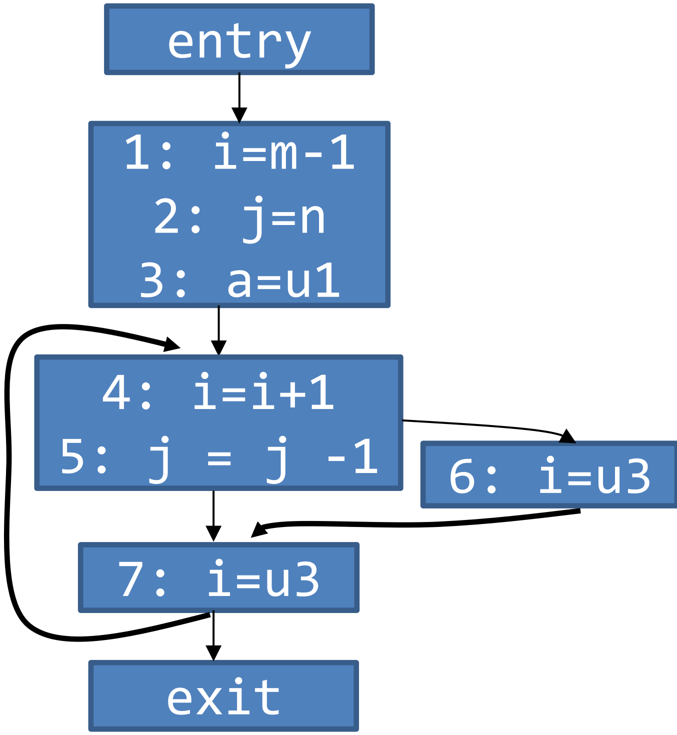
CS406, IIT Dharwad 41

Exercise: Repeat for Z and N

Reaching Definitions - Example

- **Goal:** to know where in a program each variable x may have been defined when control reaches block b
- Definition d reaches block b if there is a path from point immediately following d to b, such that the variable defined in d is not redefined / killed along that path

$$\mathtt{In(b)} = \bigcup_{i \in Pred(b)} Out(i)$$



$$\mathtt{Out(b) = gen(b) \cup (In(b) - kill(b))}$$

```
//set that contains all statements
that may define some variable x in
b. E.g. gen(1:a=3;2:a=4)={2}
```

```
//set that contains all statements
that define a variable x that is
also defined in b. E.g.
kill(1:a=3; 2:a=4)={1,2}
```