Dataflow Analysis

Week 14: Liveness Analysis

Recap

1. Dataflow analysis is the framework for optimizing the whole program (not just basic blocks)
2. A complex program is analyzed by looking at a pair of adjacent statements
   – 'Push' / 'transfer' information from one statement to another.

     E.g. in constant propagation, the information consisted of a state vector containing special values (Top, Bottom, K)
3. Construct CFG
4. Symbolically execute the program by traversing the CFG
   – Determine the parameters: lattice, transfer function, direction of execution, and how to compute information at merge points (confluence operator). *Run work list algorithm.*

Recap..

1. We used abstract values (Bottom (⊥), Top (⊤), K) to associate with a *set of* concrete values of variables (e.g. in constant propagation)
2. The abstract values are ordered according to the information that they convey (from least to most information):

   ⊥ < K < ⊤ (⊥ - Don't know / statement not executed, K – some constant, ⊤ - definitely not constant)
3. The value for a variable changes monotonically (meet / ⊓ and join / ⊔ operators ensure this)

   The monotonicity property also ensures that the worklist algorithm terminates

How do we use dataflow analysis for computing liveness property of variable (s)?

Liveness – Recap..

X **defined** here

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

X **used** here

X is live at 1

may be 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..

X **defined** here

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

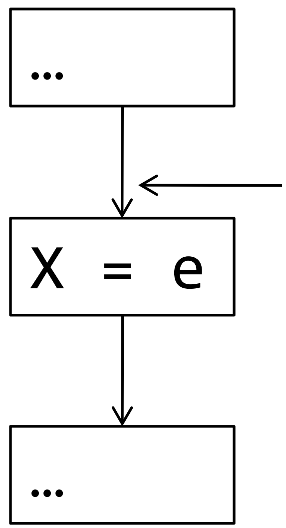
X is dead at 1

X **used** here

A variable X is dead at statement S if it is not live at S:

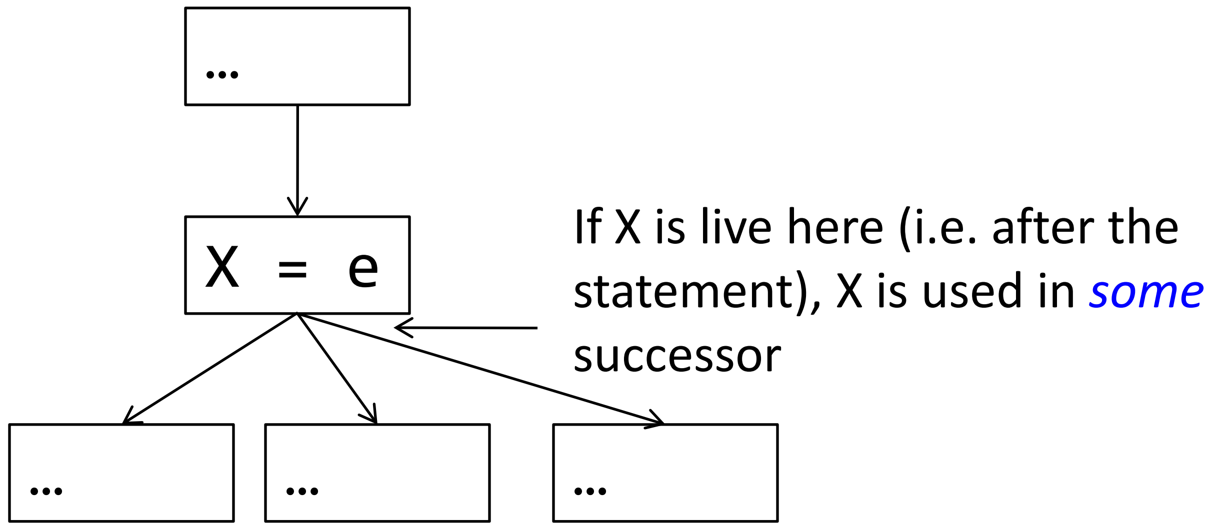
- E.g. If statement S is of the form X=exp, then there exists no statement that uses the value of X computed at S.

Liveness in a CFG

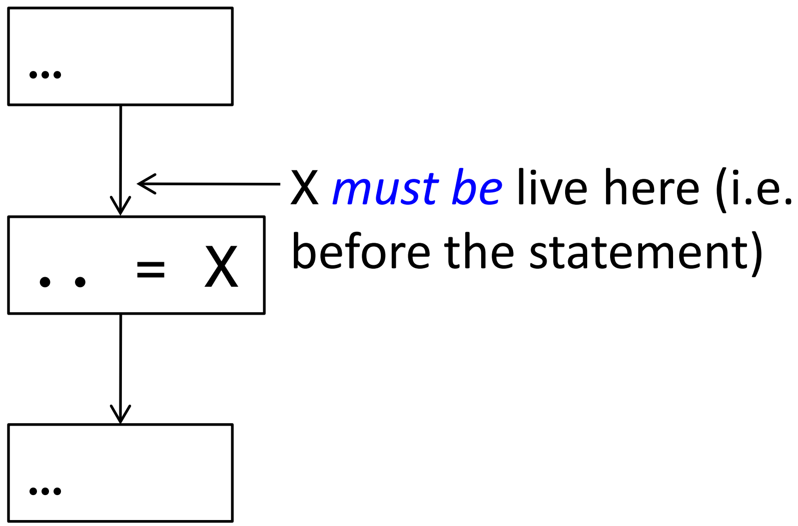


Given that e does not use X, X is *definitely dead* here (i.e. before the statement).

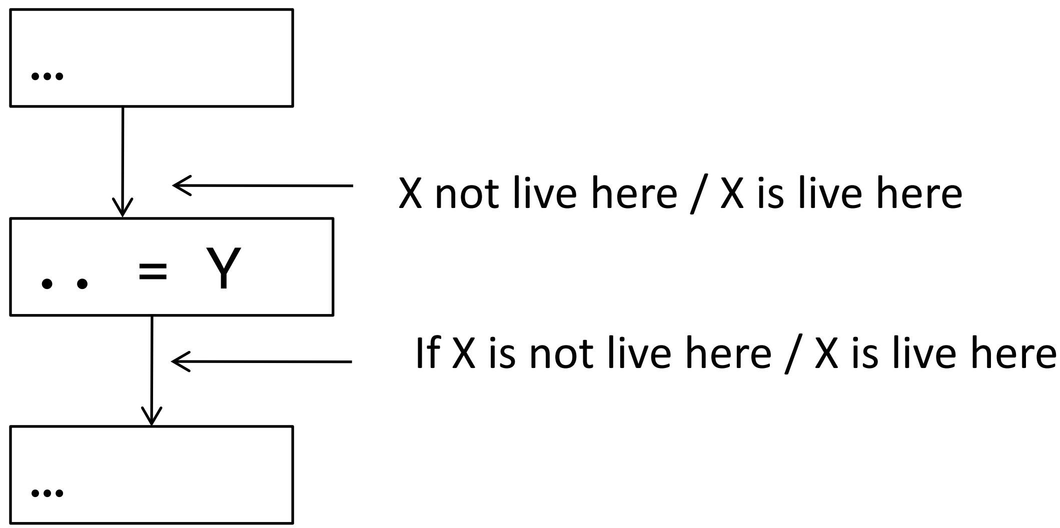
Liveness in a CFG



Liveness in a CFG



Liveness in a CFG



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

Choose dataflow direction

- A variable is *live* if it is used later in the program without being redefined
  - At a given program point, we want to know information about what happens later in the program
  - This means that liveness is a *backwards* analysis
    - Recall that we did liveness backwards when we looked at single basic blocks

Create x-fer functions

- Let's generalize
- For any statement s, we can look at which live variables are *killed*, and which new variables are made live (*generated*)
- Which variables are killed in s?
  - The variables that are *defined* in s: DEF(s)
- Which variables are made live in s?
  - The variables that are *used* in s: USE(s)
- If the set of variables that are live after s is X, what is the set of variables live before s?

$$T_s(X) = \mathbf{use}(s) \cup (X - \mathbf{def}(s))$$

Dealing with aliases

- Aliases, as usual, cause problems
- Consider

```
int x, y, r, s
int *z, *w;
if (...) z = &y else z = &x
if (...) w = &r else w = &s
*z = *w; //which variable is defined? which is used?
```

- What should USE(*z = *w) and DEF(*z = *w) be?
  - Keep in mind: the goal is to get a list of variables that *may* be live at a program point
- For now, assume there is no aliasing

Dealing with function calls

- Similar problem as aliases:

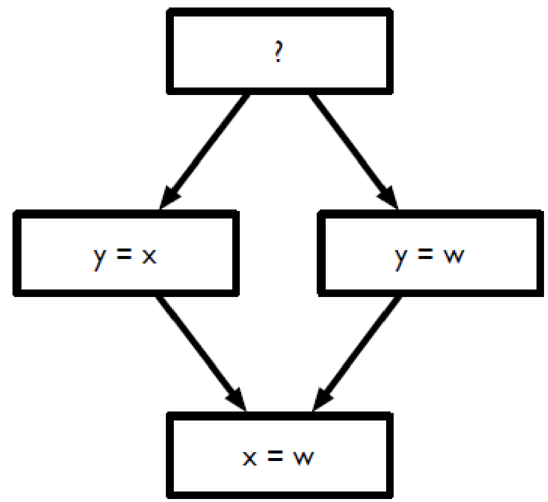
```
int foo(int &x, int &y); //pass by reference!

void main() {
  int x, y, z;
  z = foo(x, y);
}
```

- Simple solution: functions can do *anything* – redefine variables, use variables
  - So DEF(foo()) is { } and USE(foo()) is V
- Real solution: *interprocedural* analysis, which determines what variables are used and defined in foo

Choose confluence operator

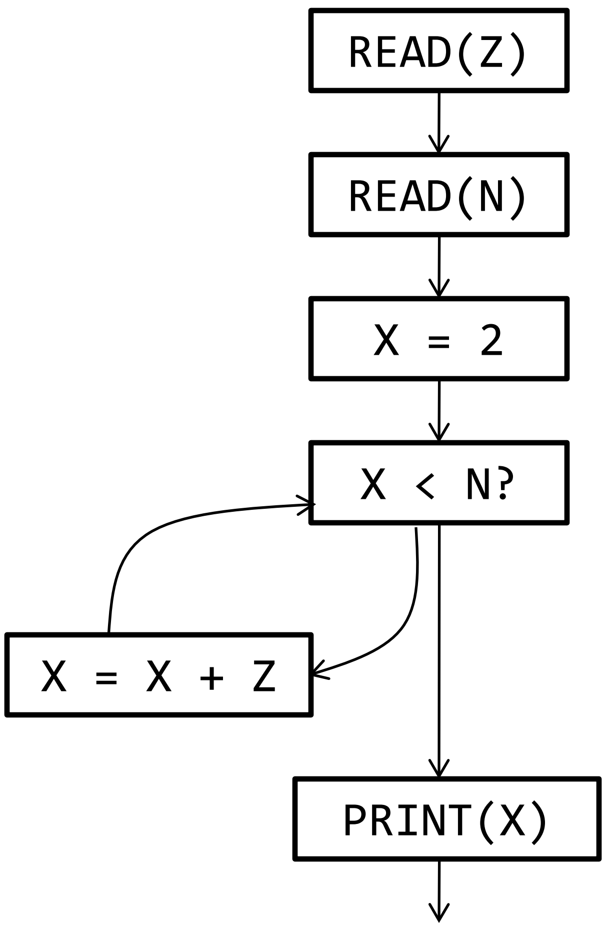
- What happens at a merge point?
  - The variables live in to a merge point are the variables that are live along *either* branch
  - Confluence operator: Set union ($\sqcup$) of all live sets of outgoing edges



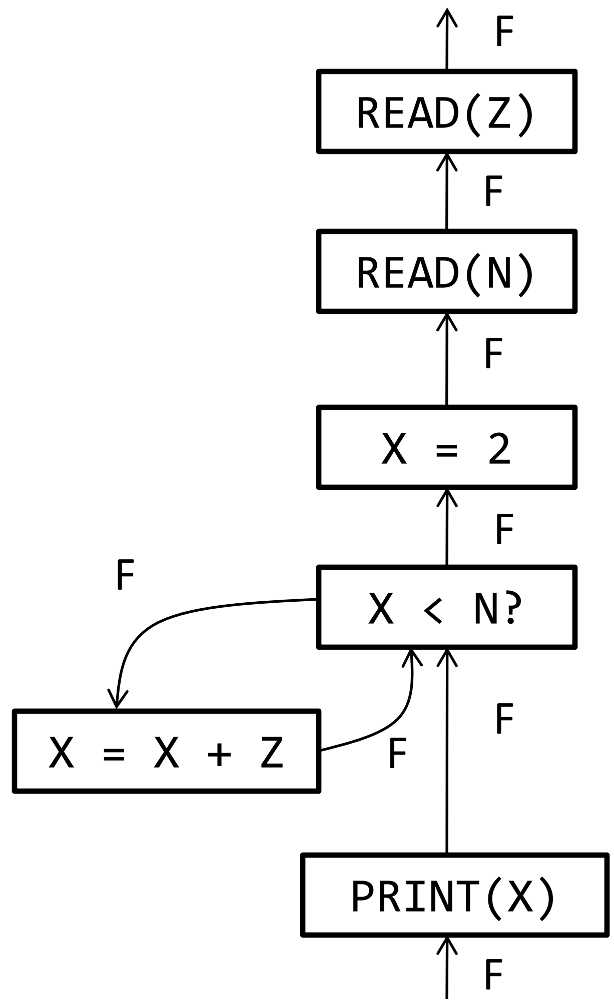
$$T_{merge} = \bigcup_{X \in succ(merge)} X$$

How to initialize analysis?

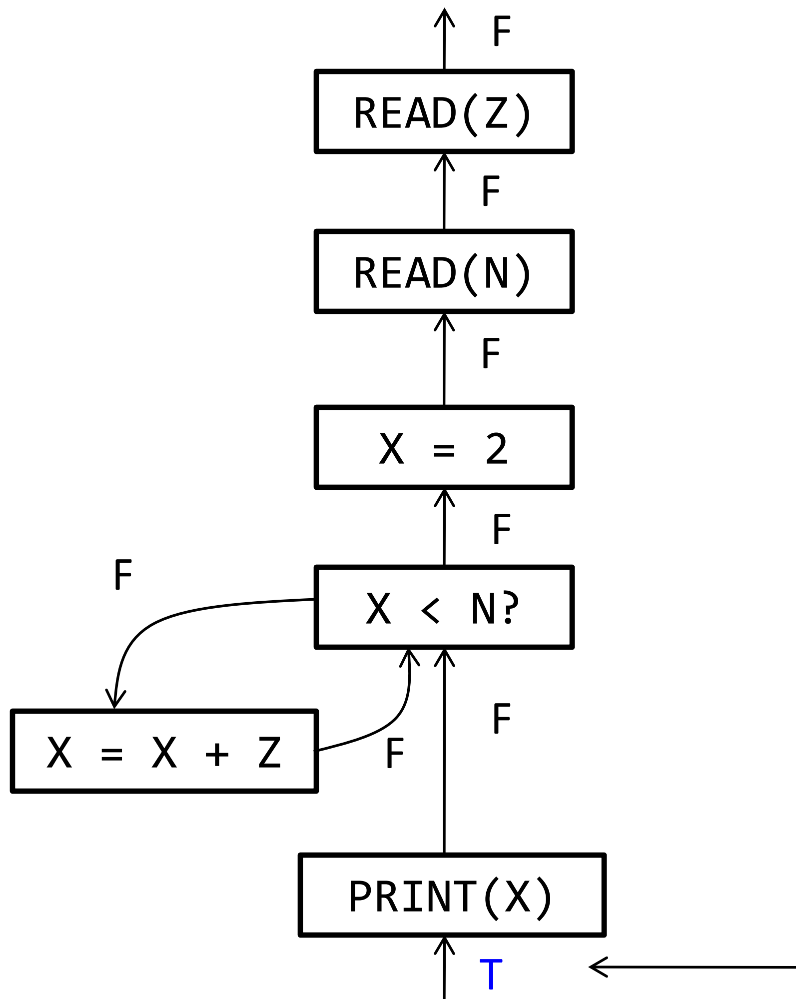
- At the end of the program, we know no variables are live
  → value at exit point is { }
  - What about if we're analyzing a single function? Need to make conservative assumption about what may be live
- What about elsewhere in the program?
  - We should initialize other sets to { }



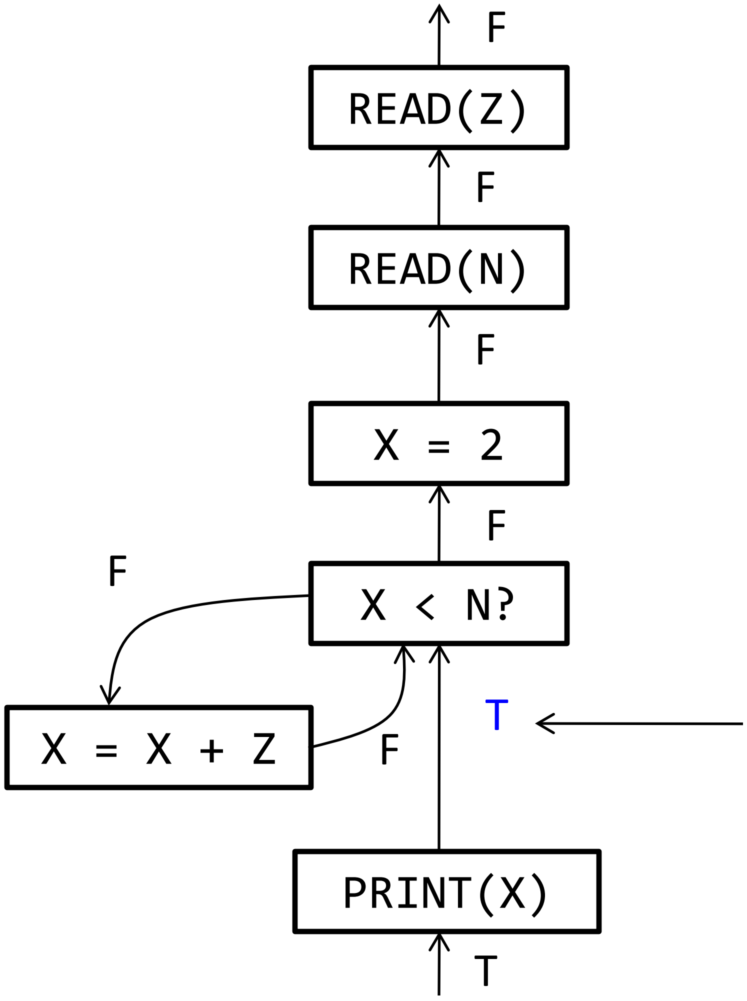
Original CFG



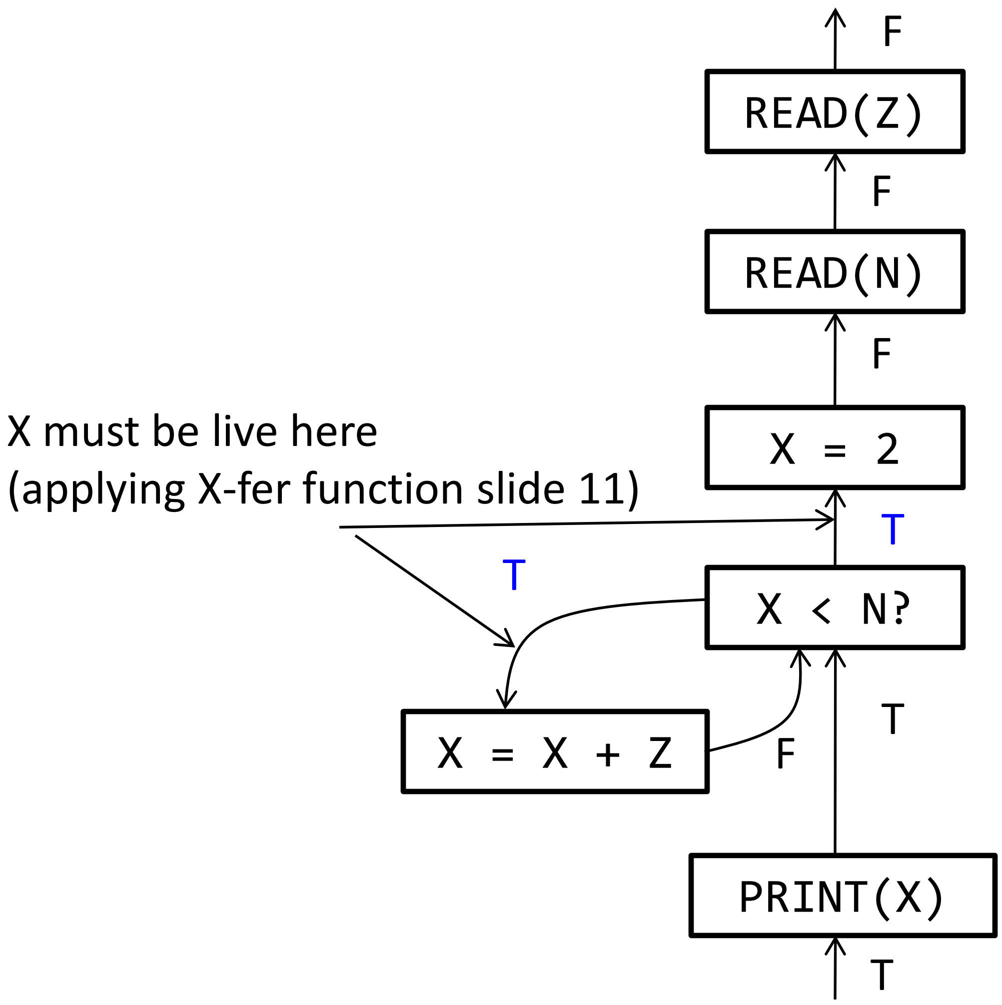
CFG with edges reversed (and initialized) for backwards analysis: is X live? (F=false, T=true)

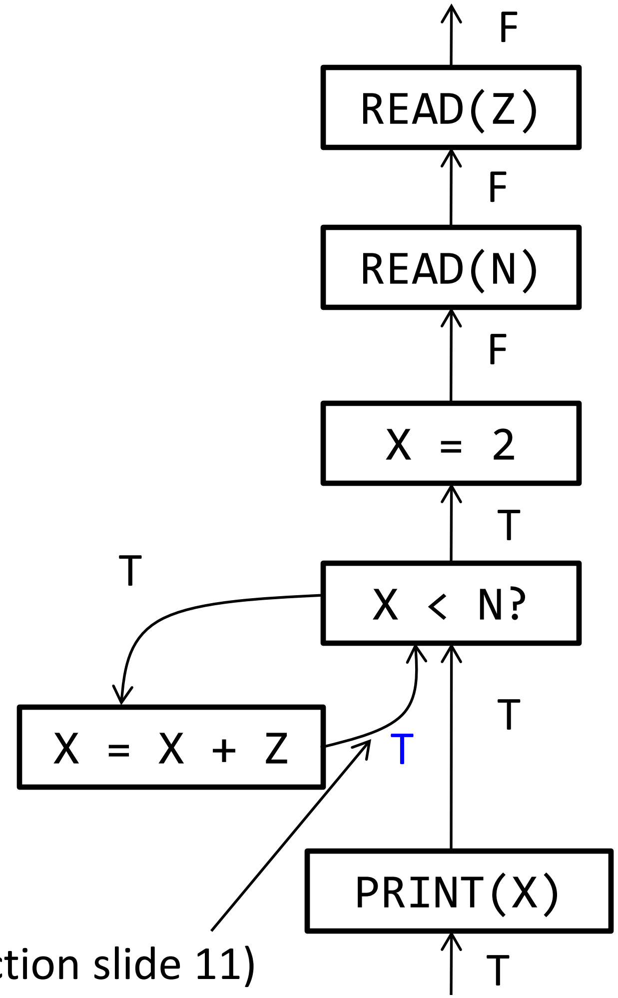


X must be live here
(rule: slide 8)

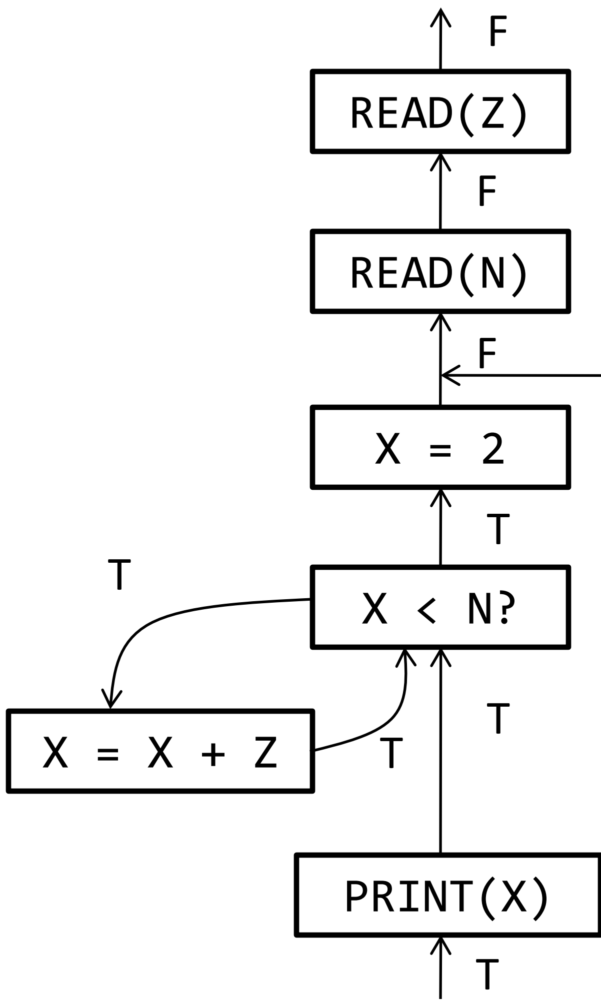


X must be live here
(applying X-fer function slide 11)

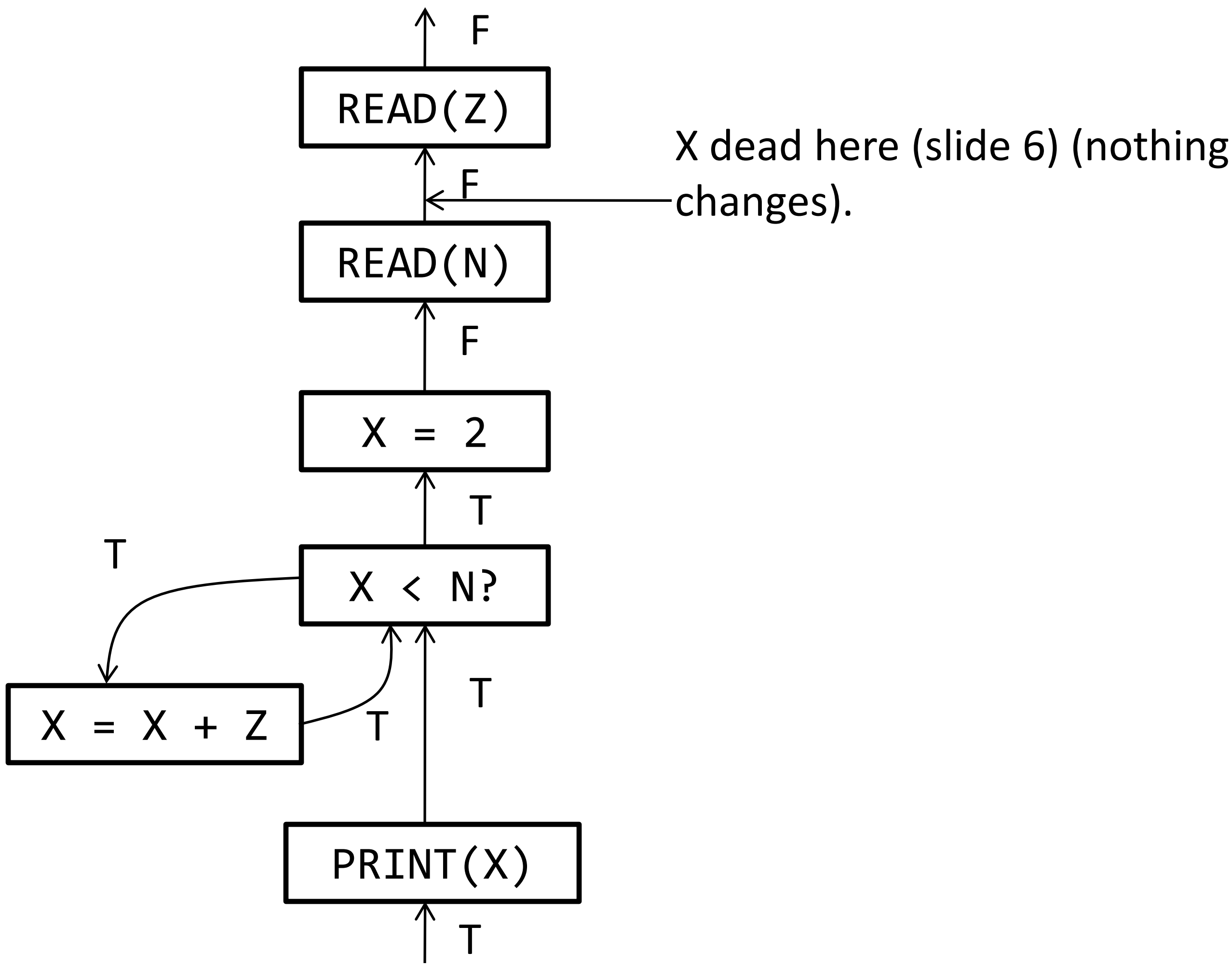

X must be live here
(applying X-fer function slide 11)
F
READ(Z)
F
READ(N)
F
X = 2
T
T
X < N?
X = X + Z
F
T
PRINT(X)
T



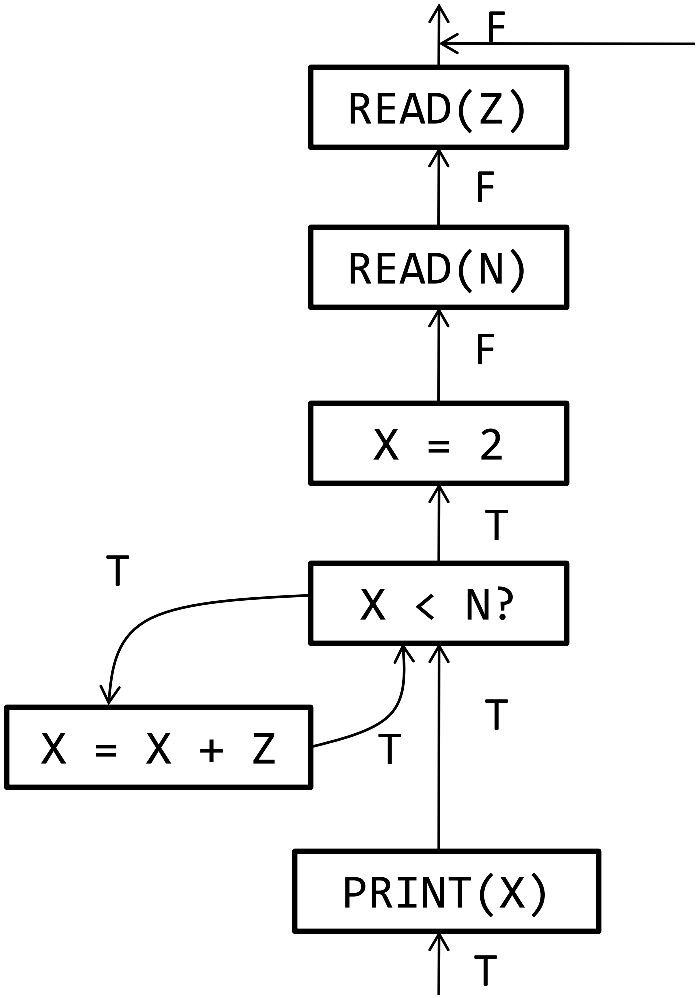
X must be live here
(applying X-fer function slide 11)



X dead here (slide 6) (nothing changes).

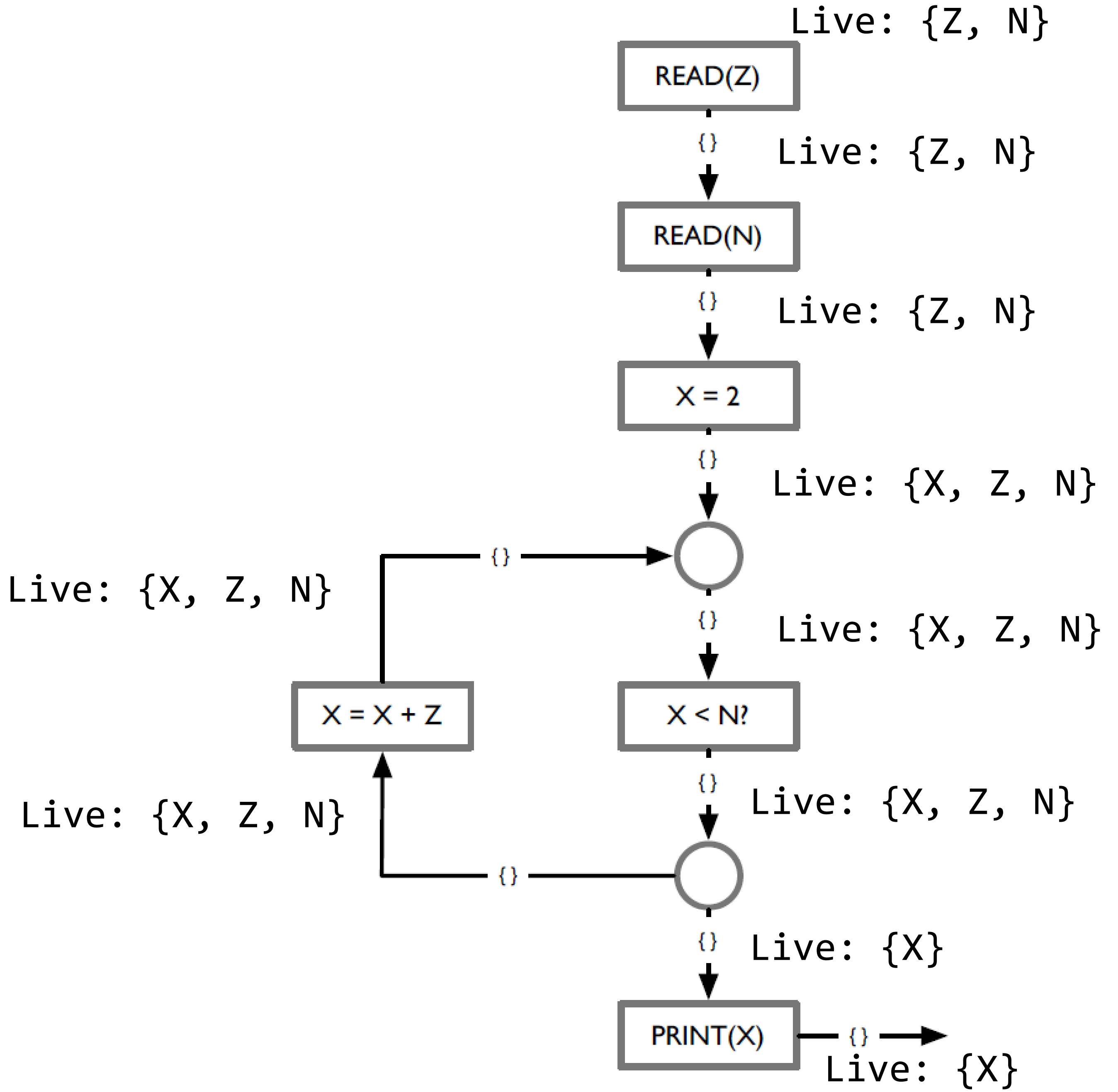

F
READ(Z)
X dead here (slide 6) (nothing changes).
F
READ(N)
F
X = 2
T
T
X < N?
X = X + Z
T
T
PRINT(X)
T

X dead here (slide 6) (nothing changes).



Exercise

Repeat liveness for variables Z and N



Live: {Z, N}
READ(Z)
{}
Live: {Z, N}
READ(N)
{}
Live: {Z, N}
X = 2
{}
Live: {X, Z, N}
{}
Live: {X, Z, N}
{}
Live: {X, Z, N}
X = X + Z
X < N?
{}
Live: {X, Z, N}
Live: {X, Z, N}
{}
{}
Live: {X}
PRINT(X)
{}
Live: {X}