Preliminary Transformations
Chapter 4

Vivek Pai
Comp 515
March 2, 1995

Purpose

Transform code to make dependence testing workable

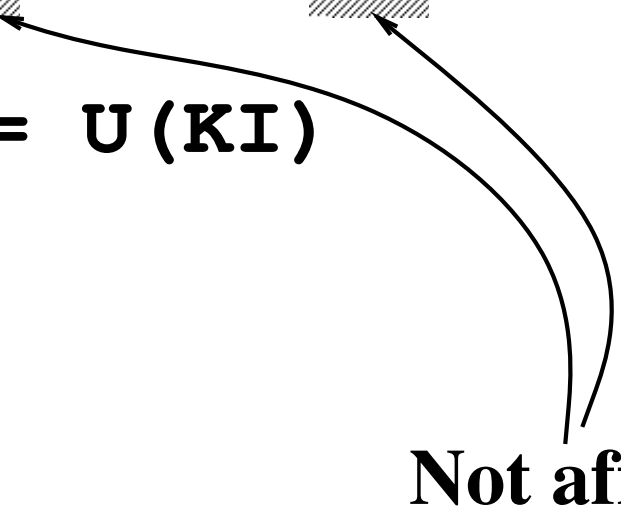
Transformations

Induction Variable Substitution

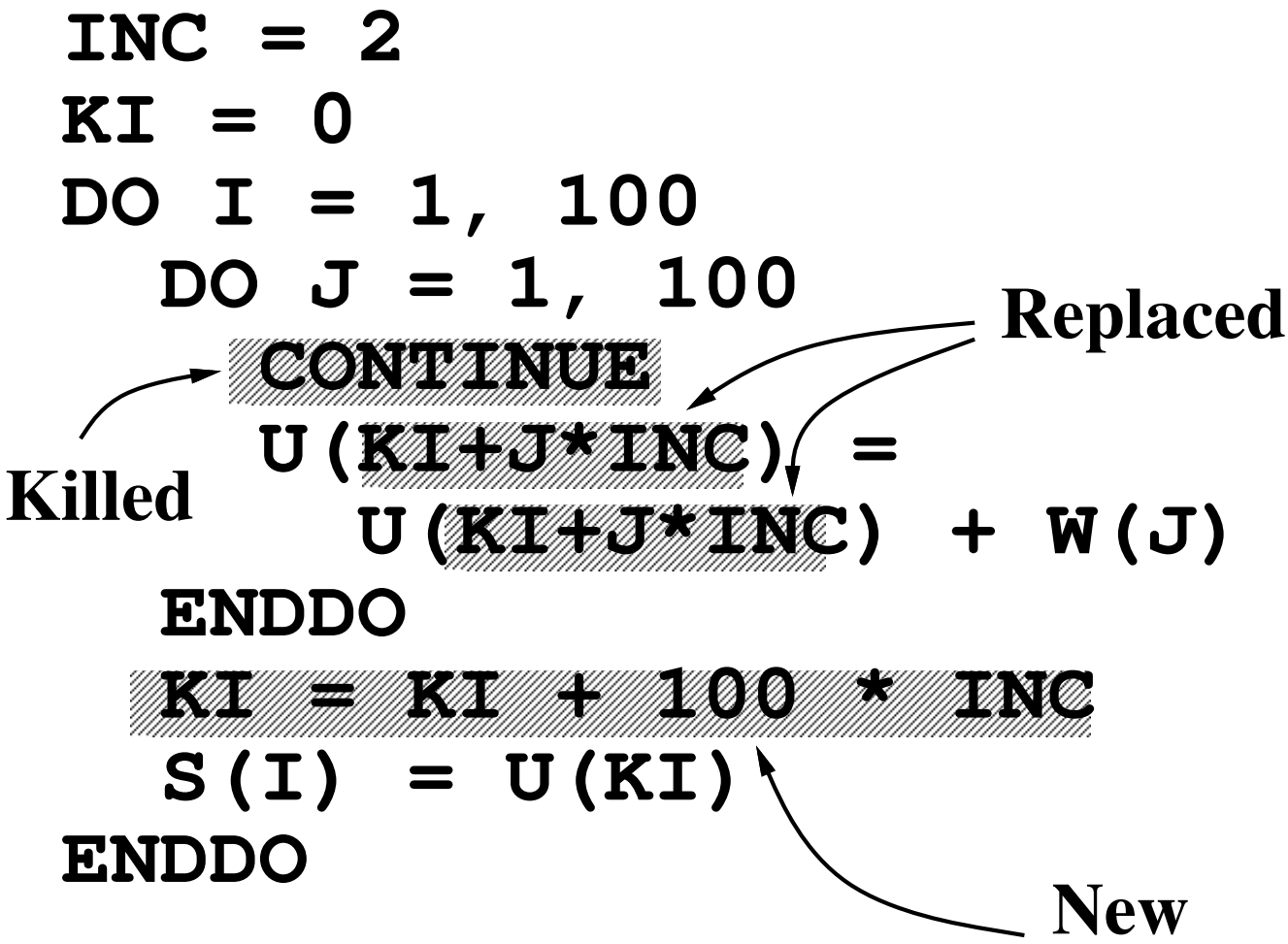
Variable that tracks with loop index → **Direction function of loop index**

Before

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      KI = KI + INC
      U(KI) = U(KI) + W(J)
   ENDDO
   S(I) = U(KI)
ENDDO
```



After IVS


INC = 2
KI = 0
DO I = 1, 100
DO J = 1, 100
CONTINUE
U(KI+J*INC) =
U(KI+J*INC) + W(J)
ENDDO
KI = KI + 100 * INC
S(I) = U(KI)
ENDDO
Replaced
Killed
New

After IVS

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(KI+J*INC) =
         U(KI+J*INC) + W(J)
   ENDDO
   KI = KI + 100 * INC
   S(I) = U(KI)
ENDDO
```

But wait

Second IVS

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(KI+J*INC) =
         U(KI+J*INC) + W(J)
   ENDDO
   CONTINUE
   S(I) = U(KI+ I*(100*INC))
ENDDO
KI = KI + 100*100*INC
```

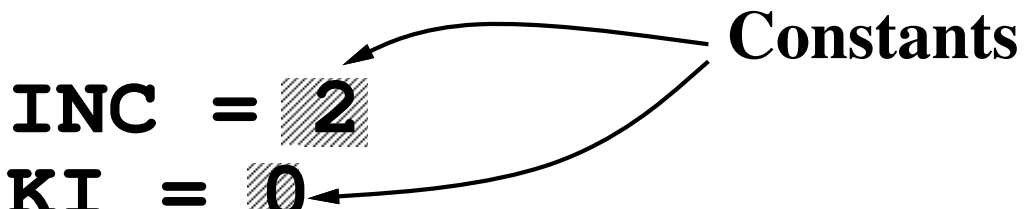
Killed

Replaced

New

All subscripts affine

Second IVS




```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(KI+J*INC) =
         U(KI+J*INC) + W(J)
   ENDDO
   CONTINUE
   S(I) = U(KI+ I*(100*INC))
ENDDO
KI = KI + 100*100*INC
```

Constant Propagation

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(0 +J*2) =
         U(0+J*2) + W(J)
   ENDDO
   CONTINUE
   S(I) = U(0 + I*(100*2))
ENDDO
KI = 0 + 100*100*2
```

Constant Propagation

Simplify All Replaced Values

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(0 +J*2) =
         U(0+J*2) + W(J)
   ENDDO
   CONTINUE
   S(I) = U(0 + I*(100*2))
ENDDO
KI = 0 + 100*100*2
```

Constant Propagation

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(J*2) = U(J*2) + W(J)
   ENDDO
   CONTINUE
   S(I) = U(I*200)
ENDDO
KI = 20000
```

Useless Redundancy

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(J*2) = U(J*2) + W(J)
   ENDDO
   CONTINUE
   S(I) = U(I*200)
ENDDO
KI = 20000
```

Dead Code Elimination

```
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(J*2) = U(J*2) + W(J)
   ENDDO
   CONTINUE
   S(I) = U(I*200)
ENDDO
```

Summary

Induction Variable Substitution

Constant Propagation

Dead Code Elimination

Dataflow Analysis

Loop invariants

Constant detection

Def–Use chains

Keith says: "The compile–time reasoning about the run time ..."

Loop Normalization

Lower, Upper, Step

↓ ↓

1, Iterations

Easier dependence testing

Detection of loop induction vars

Program transformations (IVS)

Drawbacks

```
DO I = 1, M
   DO J = I, N
      A(J,I) = A(J,I-1) + 5
   ENDDO
ENDDO
```

```
DO J = 1, ...
```

Direction vector: (<,=)

Drawbacks

```
DO I = 1, M
  DO J = 1, N-I+1
    A(J+I-1,I) =
      A(J+I-1,I-1) + 5
  ENDDO
ENDDO
```

```
I = I'- 1
I'= I + 1

J+I = J'+ I'
    = J'+ I + 1
J = J'+ 1
J'= J - 1
```

Direction vector: (<,>)

LN Procedure

Replace loop bounds (contract)

```
DO I = L, U, S
```

becomes

```
DO i = 1, (U-L+S)/S
```

Replace usage (expand)

```
I => i*S - S + L
```

Final assignment (equalize)

```
I = i*S - S + L
```

Def–Use Chains

Edges from variable definitions

to variable uses

Within a Block

Go through block in order

Each use gets pointer from last definition

Across Blocks

uses(b)

Variables used in b,
with no prior definition in b

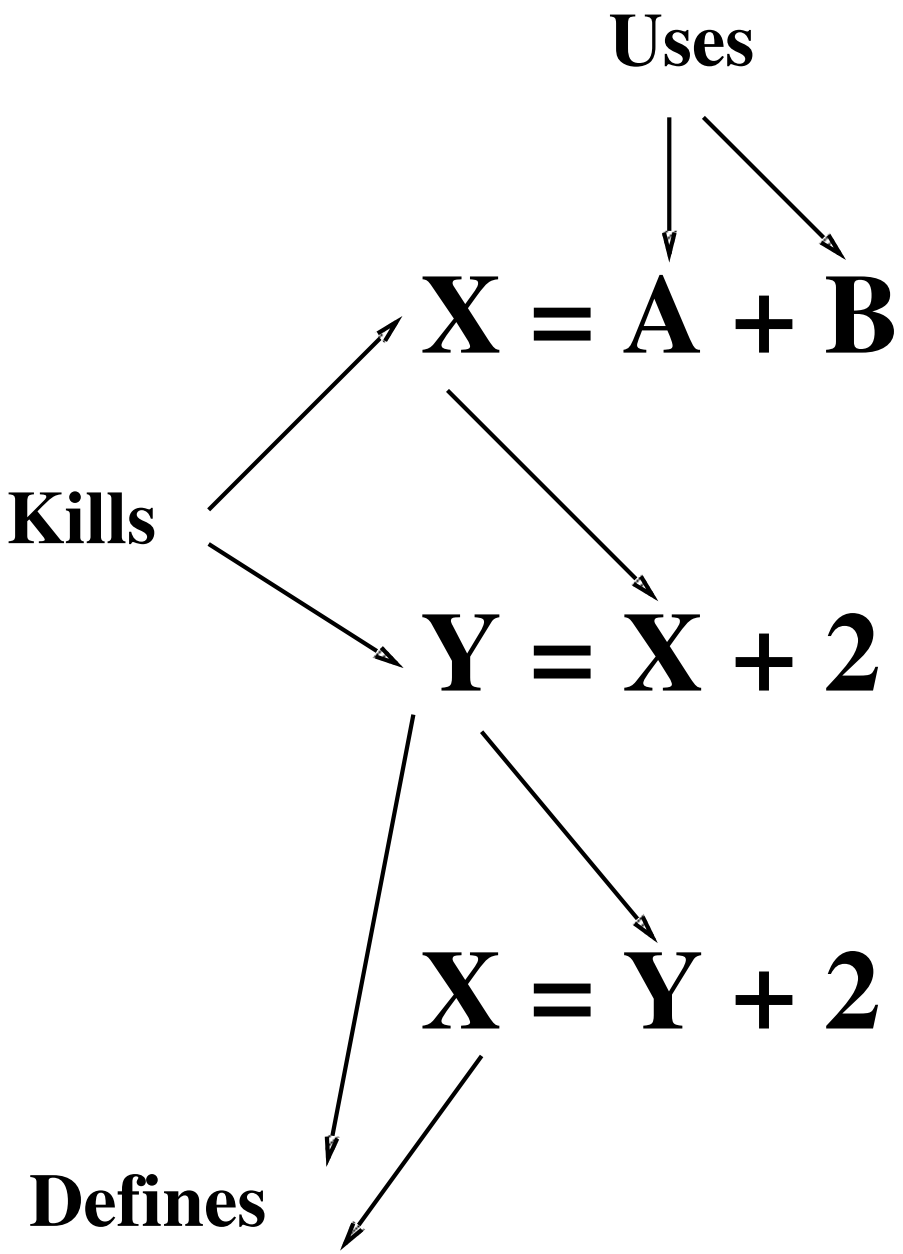
defines(b)

all definitions in b
not killed in b

kills(b)

all variables killed in b
due to definitions

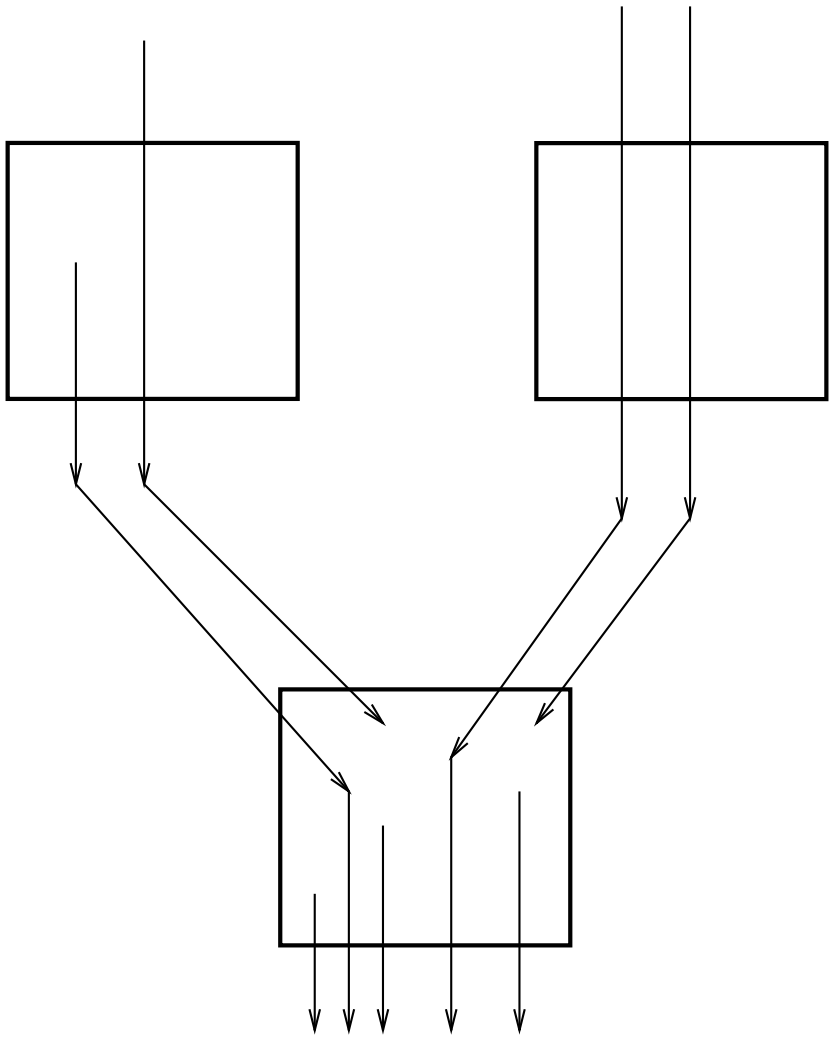
Example



Reaches

The set of all definitions from all blocks that can reach b

Global



Reaches Defined

Reaches(b) =

union (over predecessors p)

defines(p) union

(reaches(b) intersect notkilled(p))

Iterative Method

Initially, reaches(b) = null

defines(b), killed(b) are local sets

compute new reaches(b) for all blocks, iterate until stable (no changes)

Dead Code Elimination

Output is useful

Anything that doesn't get used by the output isn't useful, sort of

Finding Useful

Output, Input, Control Flow all in worklist

Remove element from worklist

Tag it as useful

For element, add defs of all uses to worklist

Repeat until worklist empty

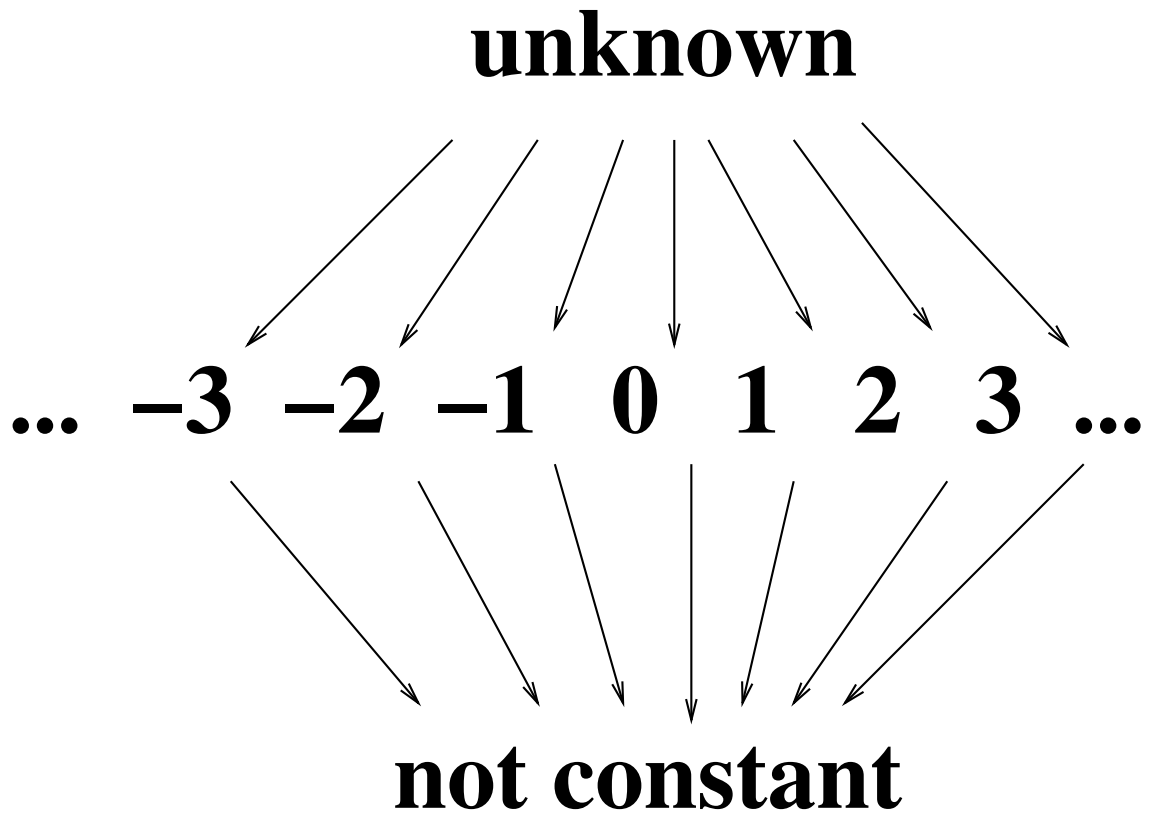
Constant Propagation

If we can prove a variable

is constant, replace all of

its uses with the constant

The Lattice



Algorithm (simplified)

Start at either unknown or constant

Worklist gets everything that is constant

Take element from worklist

For all uses of this def

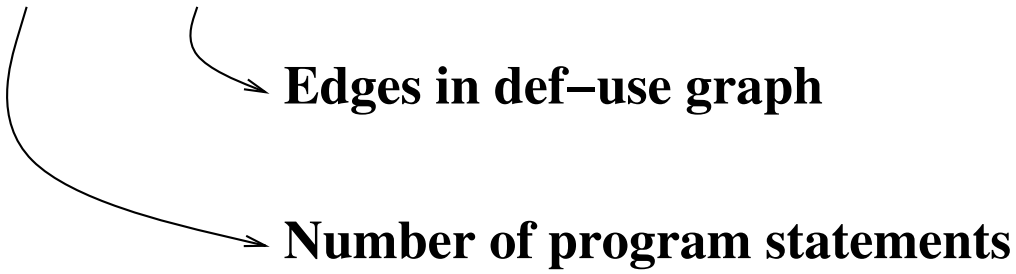
Find all defs for each use

If defs agree, replace the use

If a new constant emerges, add to worklist

Complexity

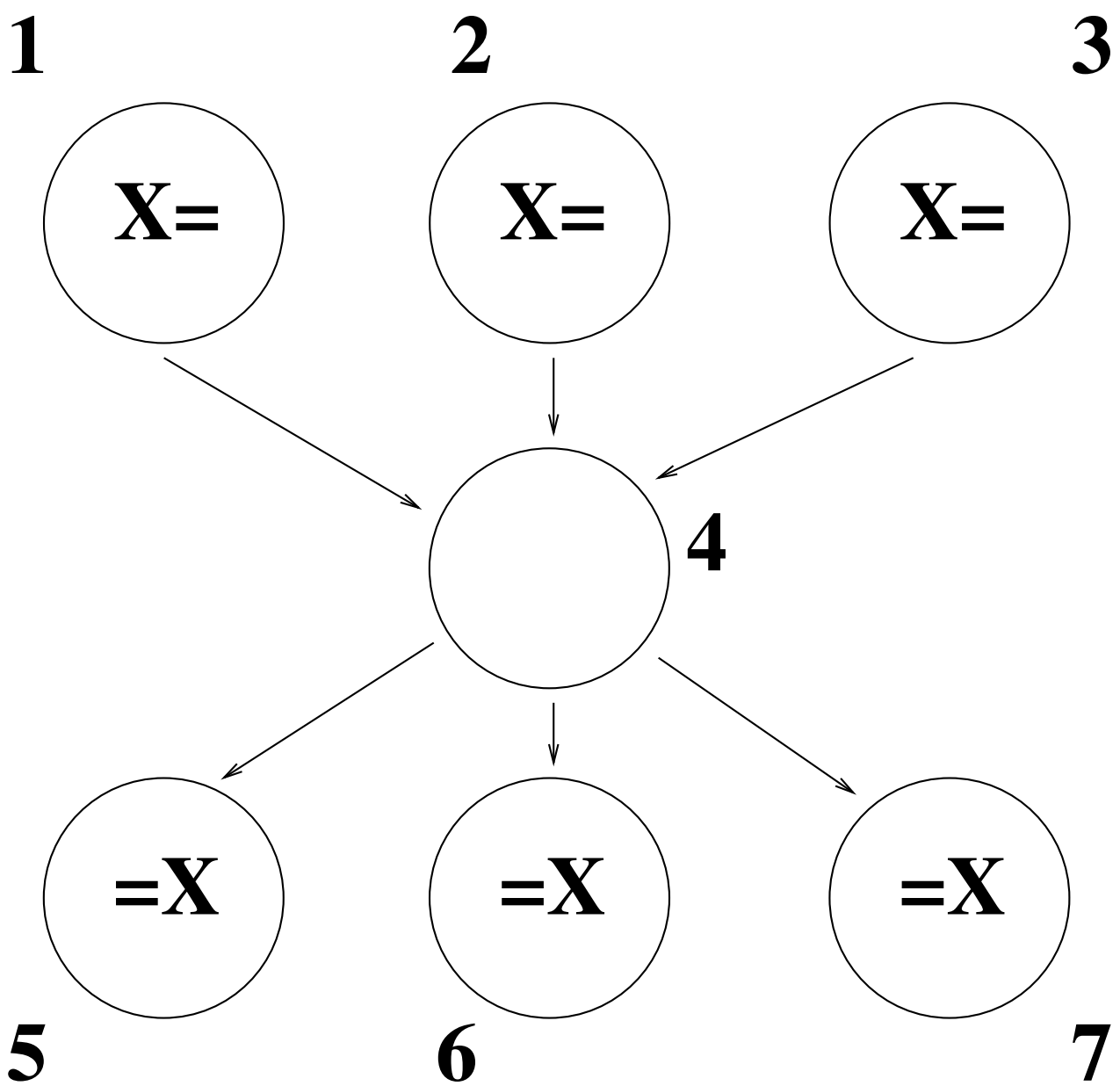
O(N+E)



Each element "lowered" at most twice

Each def–use edge examined at most twice

Worst Case

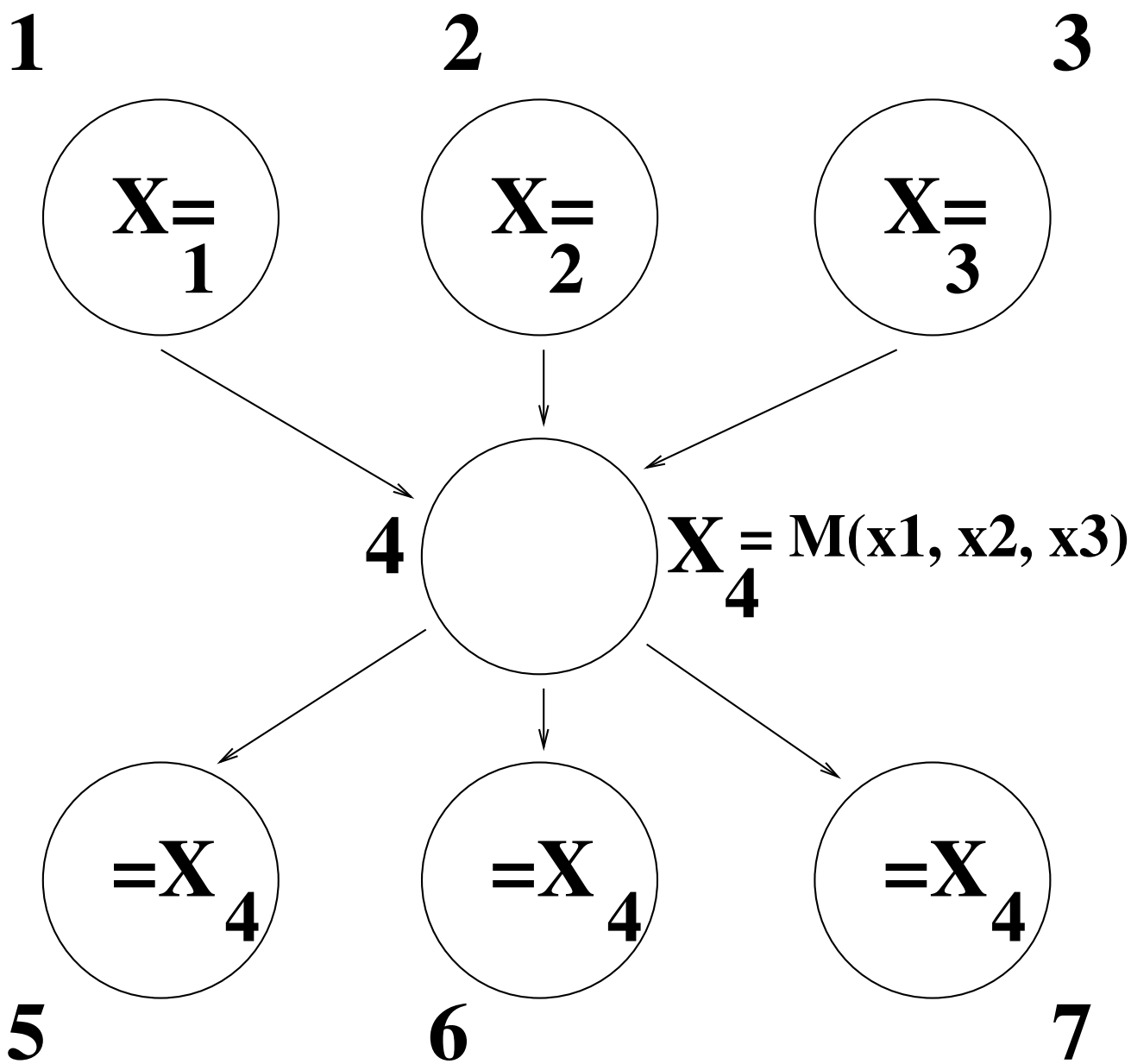


Static Single Assignment

Each assignment creates a different variable name

Merge variables at control flow joins

New Worst Case



What SSA Gets You

Every use has only 1 def

Constant propagation searched all defs before to verify constant

Merge nodes now check for non–constant value

Next Time

Induction Variable Substitution

the joys, the heartaches