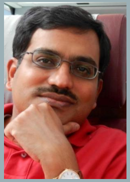
STATIC ANALYSIS PART 1 - MOTIVATION

CS3213 FSE

Prof. Abhik Roychoudhury

National University of Singapore



WHAT WE DID EARLIER

UML as modeling notation

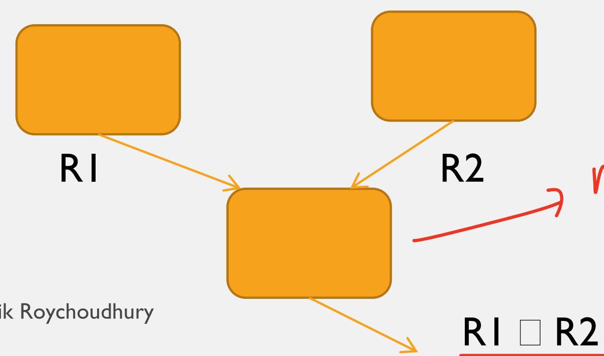
- System Requirements: Use-cases, Scenarios, Sequence Diagrams
- System structure: Class diagrams
- Discussion on semantics
- System behavior: State diagrams
- Discussion of the thinking behind your course project

- Today
  - Start discussion on software engineering practices for code instead of models
  - **Static analysis and vulnerability detection: also touches upon Secure SE**

STATIC ANALYSIS

- Do not try to generate tests which show vulnerabilities.
- Do not try to explore paths in the program
  - Analysis is path insensitive.
  - Instead treat the source code as an artifact, and analyze the source-code directly.
  - Since analysis results from different paths get merged at control flow merge points – analysis output is approximate.
  - Lot of false alarms !

don't care execution



merged at control flow merge points.

output is approx. of R1, R2.

SIMPLE EXAMPLE

```
1.  n = 0;
2.  while (n < large_number){
3.      n = n + 1
4.  }
5.  // exit code
```

all possible executions of program.

✗ just one specific execution.

Iteration 1: $Val_{n,2} = [0,0]$
Iteration 2: $Val_{n,2} = [0,1]$
Iteration 3: $Val_{n,2} = [0,2]$
…

Concrete execution:

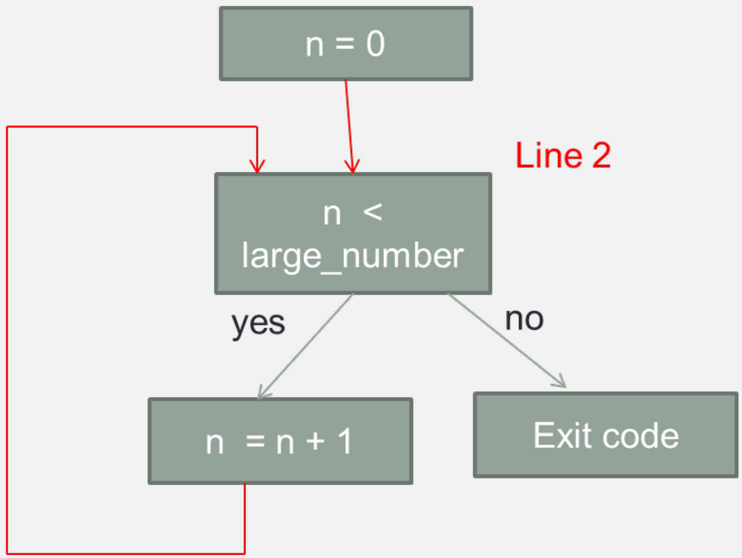
Value of a variable at a program point

Abstract execution

Approximate value of a variable at a program point

[An example approximation is via *intervals of possible variable values*]

WHAT IS GOING ON?



Newer and newer values are possible by going through the loop.
As a result, the interval gets expanded.
We should approximate the **set of all possible values** in abstract execution.

ANOTHER EXAMPLE

```
input x;
while (isEven(x)){
      x = x / 2;
}
x = 4*x;
… // exit code
```

at the end of loop. x is even.

Abstract execution

Just keep track in each location whether

x is even or odd

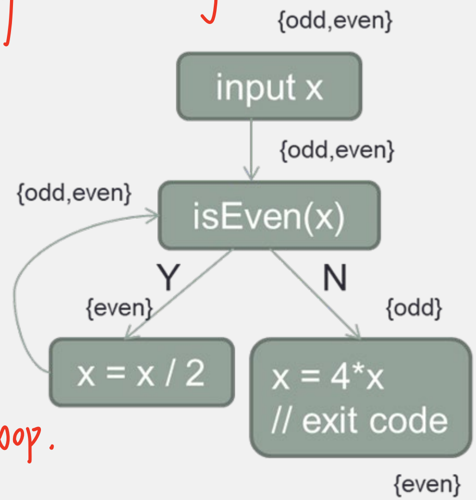
This is different from the interval representation.

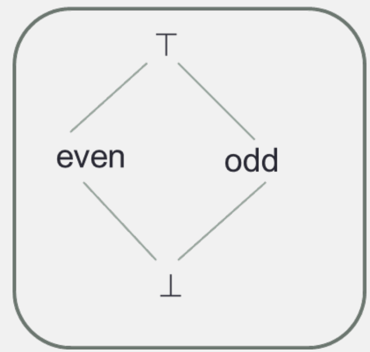
ABSTRACT EXECUTION

get as much information of x, in each line.

```
1.  input x;
2.  while (isEven(x)){
3.        x = x / 2;
4.  }
5.  x = 4*x;
6.  … // exit code
```

Not unrolling the loop.





Can abstract execution ensure that the value of x in line 6 is even?

You can only keep track of whether x is odd or even.

INFERENCE ACHIEVED

```
input x;
while (isEven(x)){
      x = x / 2;
}
x = 4*x;
… // exit code
```

Repeated propagation of sets of abstract values until the estimates stabilize at each program point.

Continue the estimation of abstract values until they do not change any more in any program point. This is when the computation has reached a **fixed-point**.

This provides the final "inference".

We can infer that the end value of x is even, provided exit code does not touch the value of x.

WHY STATIC ANALYSIS?

- Sample vulnerable code

```
void foo(){
	char buf[80];
	strcpy(buf, gethostbyaddr(…)->hp_hname);
}
```

Could write past the end of buf.

Typically allows the attacker to execute arbitrary code.

WHY STATIC ANALYSIS?

- Sample Application
  - Detect **buffer Overruns**: Concentrate on string variables in the program.
  - If s is a string variable, define
  - Alloc(s) == Number of bytes allocated for the string s
  - Len(s) == Number of bytes used by string s

  - Both Alloc(s) and Len(s) are sets
  - Alloc(s) captures possible values of allocated bytes to s
  - Len(s) captures possible values of length of s

  - Captures the set of values of Len(s) and Alloc(s) at any program point – over-approximation!

CONSTRAINTS

- Capture Len(s) and Alloc(s) by ranges
  - Ranges of the form [m,n]
- Constraints of the form
  - $X \subseteq Y$, where X and Y are range variables.
- Example constraint
  - strcpy(dst, src) $\Rightarrow$ len(src) $\subseteq$ len(dst)

copy src to dest.

EXAMPLES

char s[n]	$\{n\} \subseteq \text{Alloc}(s)$
s = “foo”	$\{4\} \subseteq \text{Len}(s)$ $\{4\} \subseteq \text{Alloc}(s)$
fgets(s,n,...);	$[1, n] \subseteq \text{Len}(s)$
sprintf(dst,"%d",n);	$[1, 20] \subseteq \text{Len}(dst)$

Checking $\text{Len}(s) \leq \text{Alloc}(s)$ for all string s at the end of analysis

Suppose $\text{Len}(s) = [a,b]$ and $\text{Alloc}(s) = [c,d]$

- If $b \leq c$, s never overflows the buffer
- If $a > d$, buffer over-run always occurs
- If the two ranges overlap, there is a possibility of buffer over-run.

The C library function char *fgets(char *str, int n, FILE *stream) reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

```
char buf[128];
while (fgets(buf,128,stdin)){
   if (!strchr(buf,'\n')){
      char error[128];
      sprintf(error,"Line long %s\n", buf);
      die(error);
   }
}
```

$[128,128] \subseteq Alloc(buf)$
$[1, 128] \subseteq Len(buf)$

$[128, 128] \subseteq Alloc(error)$

$Len(buf) + 11 \subseteq Len(error)$

Collect such constraints from the lines of the program.

Solve the constraint system and check Len(s) $\leq$ Alloc(s)

You could also keep track of ranges of buffers and over-approximate these ranges using abstract execution.

PART II - PROGRAM REPRESENTATIONS *CS3213 FSE COURSE*

Prof. Abhik Roychoudhury

National University of Singapore

(**Ack**: Xiangyu Zhang & Aditya Mathur, Purdue for some slides)

WHY PROGRAM REPRESENTATIONS

- Original representations generic.
  - Source code (cross languages).
  - Binaries (cross machines and platforms).
  - Source code / binaries + test cases.
- They are hard for machines to analyze.
- Software is translated into certain representations before analyses are applied.

CONTROL FLOW GRAPH

- The most commonly used program representation.

PROGRAM REPRESENTATION: BASIC BLOCKS

A basic block in program P is a sequence of consecutive statements with a single entry and a single exit point. Thus a block has unique entry and exit points.

Control always enters a basic block at its entry point and exits from its exit point. There is no possibility of exit or a halt at any point inside the basic block except at its exit point. The entry and exit points of a basic block coincide when the block contains only one statement.

CONTROL FLOW GRAPH (CFG)

A control flow graph (or flow graph) G is defined as a finite set N of nodes and a finite set E of edges. An edge (i, j) in E connects two nodes n_i and n_j in N. We often write G= (N, E) to denote a flow graph G with nodes given by N and edges by E.

CONTROL FLOW GRAPH (CFG)

In a flow graph of a program, each basic block becomes a node and edges are used to indicate the flow of control between blocks.

An edge (i, j) connecting basic blocks b_i and b_j implies that control can go from block b_i to block b_j.

We also assume that there is a node labeled Start in N that has no incoming edge, and another node labeled End, also in N, that has no outgoing edge.

CONTROL FLOW GRAPH

```
x = 1; y = 0; z = 0;
while (x < 10){
    if (x > 5)
        y = y + x;
    else  z = z + x;
    x = x + 1;
}
printf(…);
```

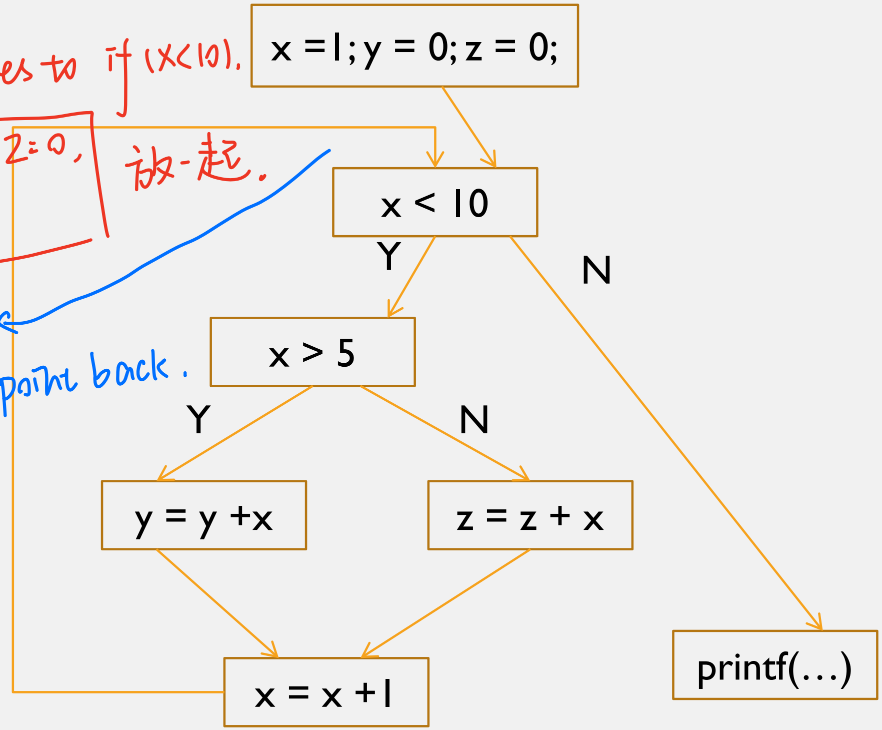
→ if this changes to if (x<10),

$\boxed{x=1,\ y=0,\ z=0,\ x<0}$ 放一起.

∵ 这里 while, $\boxed{x<10}$ 一直被 point back.

∴ 不能合并.

Otherwise. 有 control flow change, 但会改 x, y, z 值.



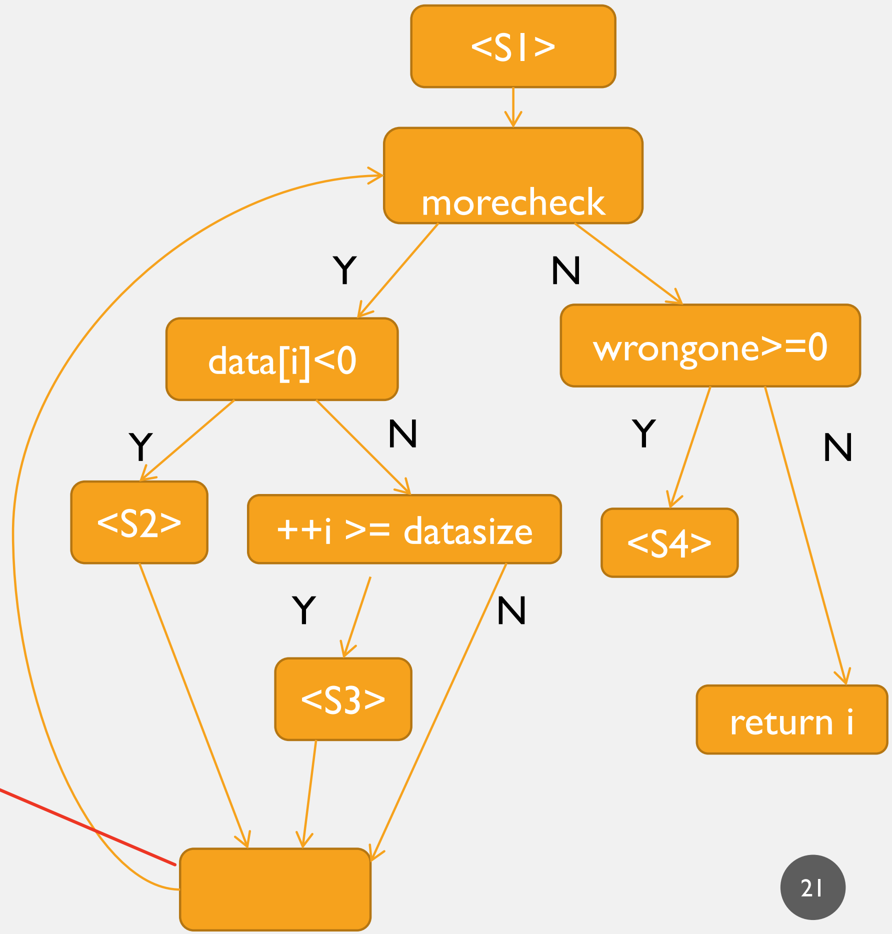
Nodes of the graph, basic blocks, are maximal code fragments executed without control transfer. The edges denote control transfer.

Only 1 feasible path.

CFG CONTINUED

```
procedure Check_data()
{          <S1>
   L:      while (morecheck)
   LB:    {
                  if  (data[i] < 0)
   A:                 {  <S2> }
                  else
   B:             if  (++i >= datasize)
                                  <S3>;
               }
              if (wrongone >= 0)
    C:                {  <S4> }
    C':      else  return i;
 }
```

什么都没有，
只用来 merge



PATHS

Consider a flow graph G= (N, E).
A sequence of k edges, k>0, (e_1, e_2, … e_k) ,
denotes a path of length k through the flow graph if the following sequence condition holds.

Given that n_p, n_q, n_r, and n_s are nodes belonging to N, and
$0< i<k$, if $e_i = (n_p, n_q)$ and $e_{i+1} = (n_r, n_s)$ then $n_q = n_r$.

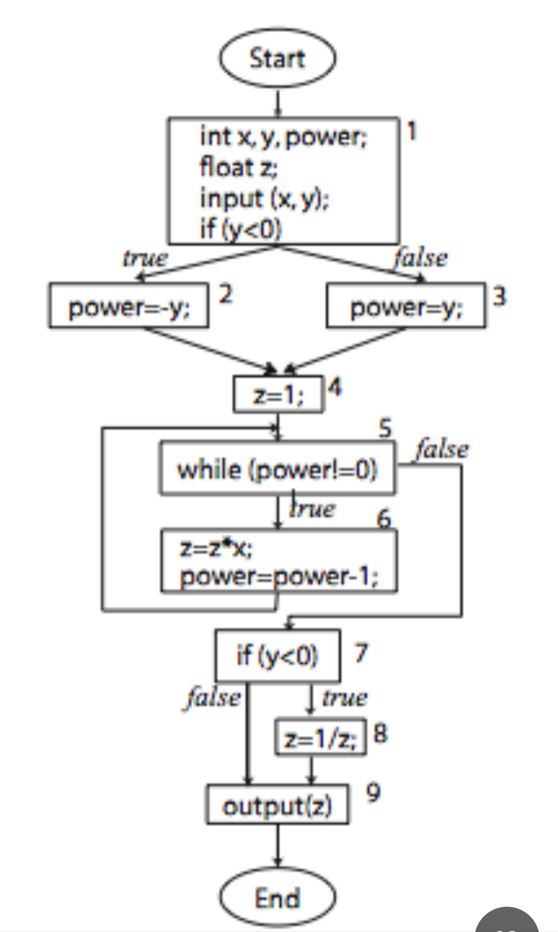
Complete path: a path from start to exit
Subpath: a subsequence of a complete path

PATHS: INFEASIBLE PATHS

A path p through a flow graph for program P is considered feasible if there exists at least one test case which when input to P causes p to be traversed.

Infeasible path: Each Node is live. Seq. of node is dead.

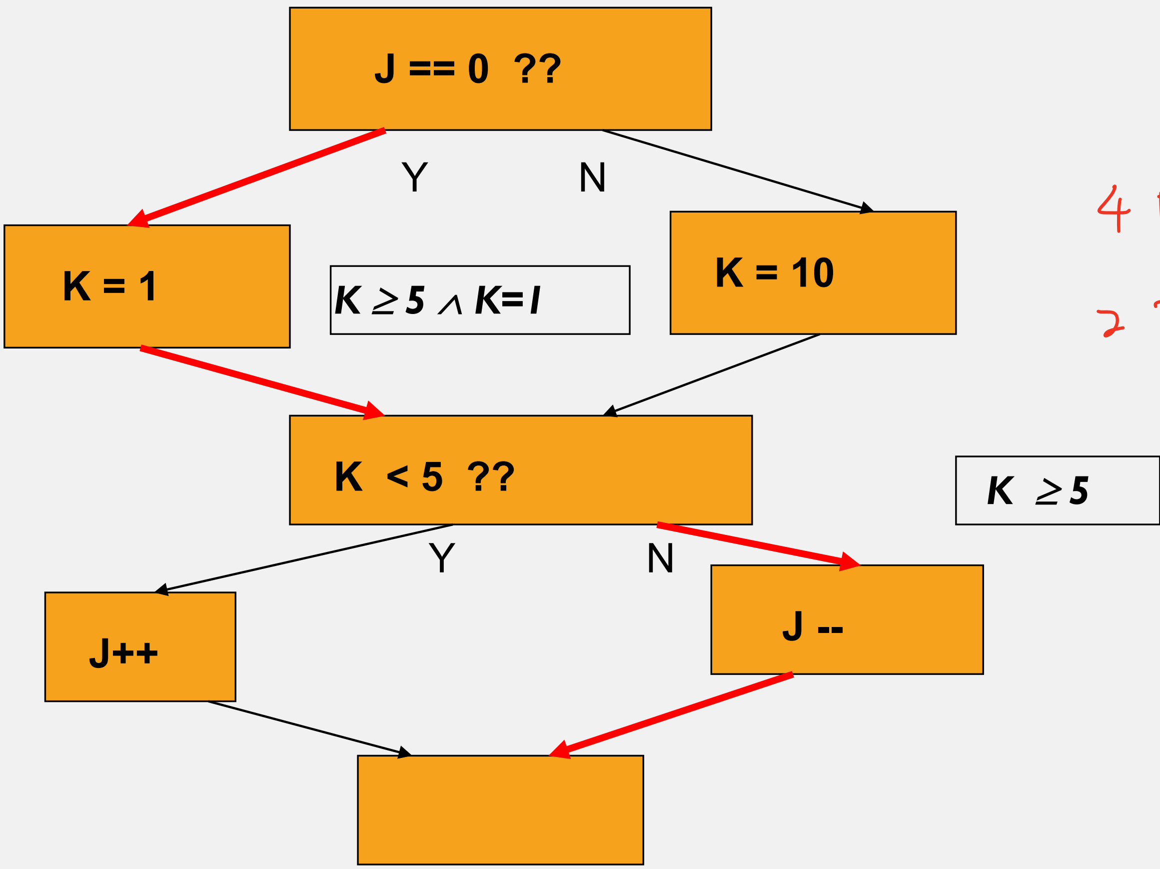
p_1= (Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End)
p_2= (Start, 1, 2, 4, 5, 7, 9, End)



INFEASIBLE PATH DETECTION

- Important problem for reducing test suite size.
- Can also be useful for accurate analysis results, or getting an accurate understanding of program behavior
- Useful to find out smallest infeasible path patterns.
- But, first how do we even test that a given path is infeasible.

TESTING FOR INFEASIBILITY



4 paths.

2 feasible paths.

COMMON MISTAKE AND WAY FORWARD

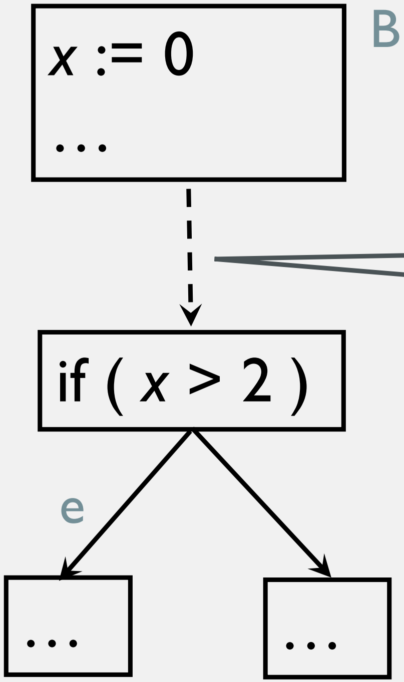
Infeasible path is different from dead code.
See the example in previous slide.

We need an automated mechanism to check whether a given path is infeasible.
We will do that later in this module.

We can always have an incomplete detection of infeasible paths using patterns
How ?

Find conflicting pairs ??
- (Assignment, Branch) or AB conflict
- (Branch, Branch) or BB conflict

CONFLICT RELATIONS

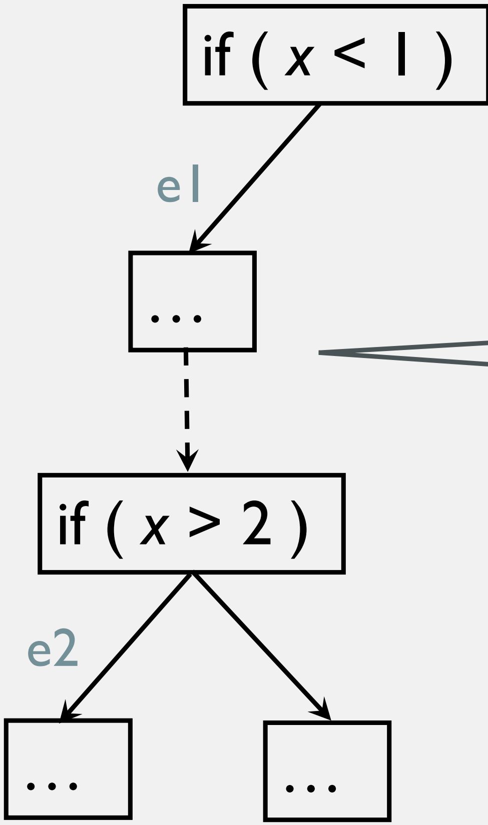


x == 0 AND x > 2

no assignment to x

Assignment-Branch (AB) conflict

CONFLICT RELATIONS



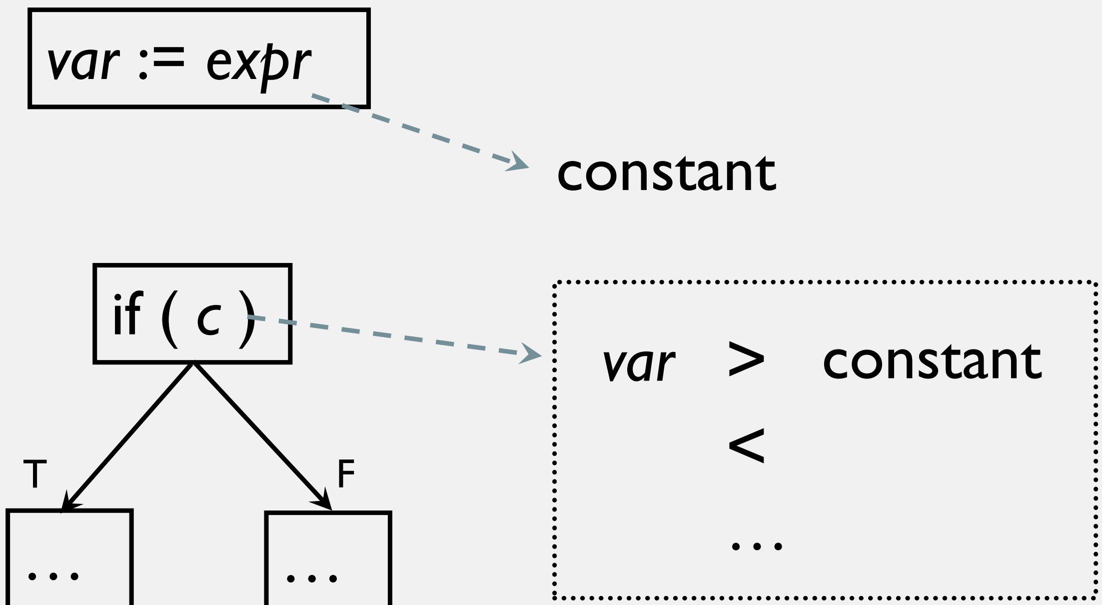
$x < 1$ AND $x > 2$

no assignment to x

Branch-Branch (BB) conflict

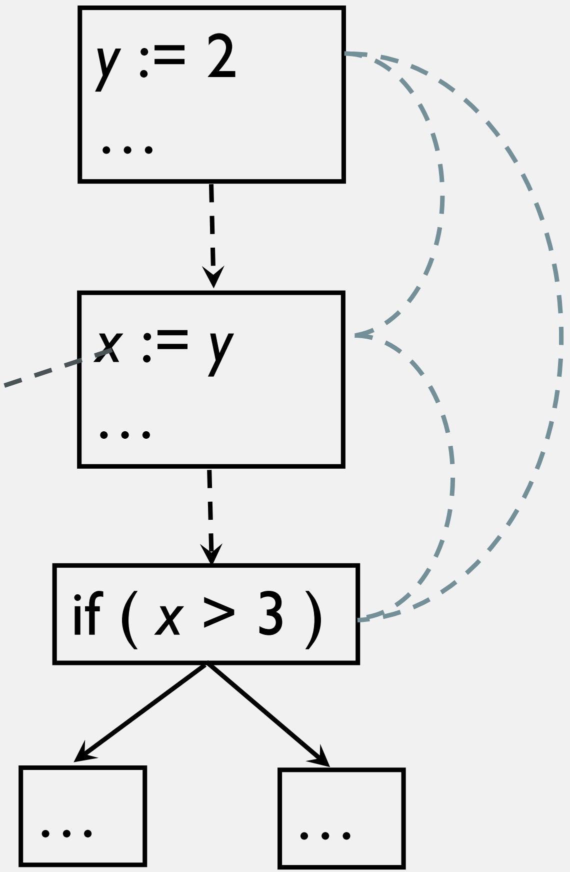
LIMITATION

- Constant-valued RHS only
  - Complex expressions not checked for feasibility



LIMITATION

- Pairwise conflicts



Also, RHS
not a constant

AB Conflict:

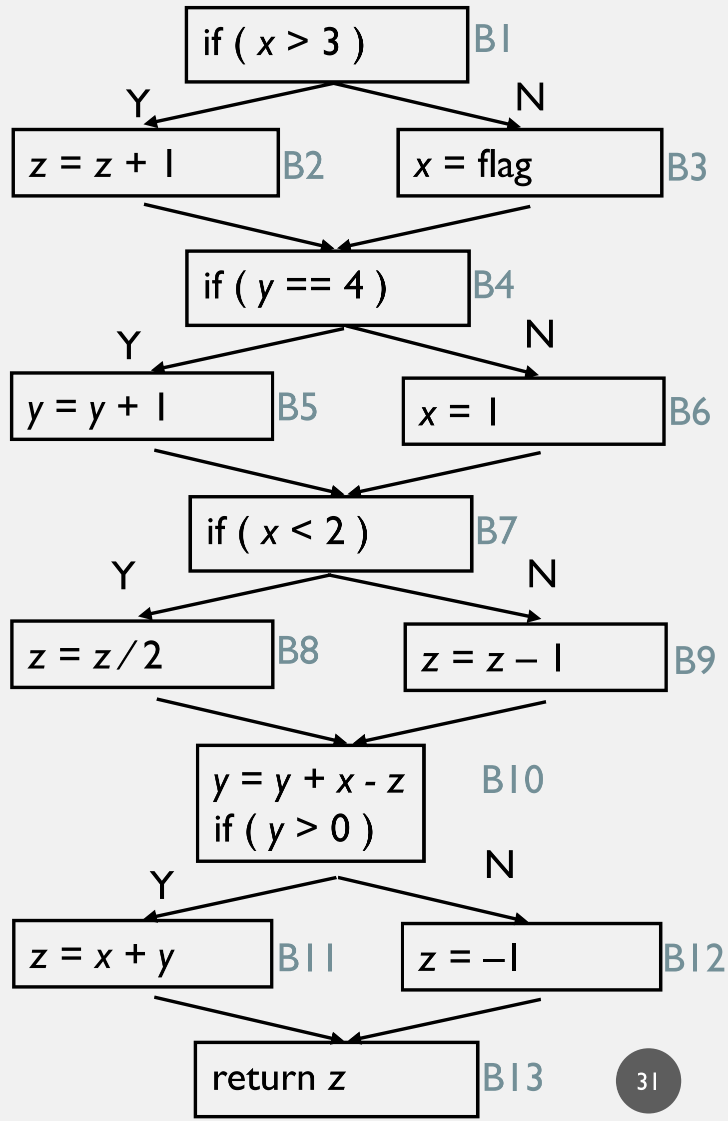
(B6, B7 → B9)

BB Conflict:

(B1 → B2, B7 → B8)

16 paths. $= 2^4$

Even utilizing such infeasible path information for static analysis is useful, even if the infeasible path detection is not fully automated.



inside procedural

(INTRA-PROCEDURAL) CFG

- nodes = regions of source code (basic blocks)
  - Basic block = maximal program region with a single entry and single exit point
  - Often statements are grouped in single regions to get a compact model
  - Sometime single statements are broken into more than one node to model control flow within the statement
- directed edges = possibility that program execution proceeds from the end of one region directly to the beginning of another

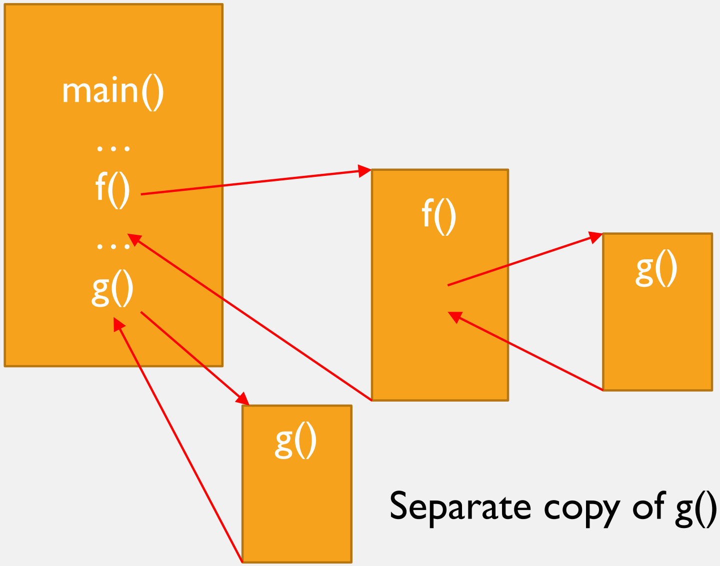
INTER-PROCEDURAL CFG

You can create a separate copy of each procedure f, for each call site of f

This is only to make sure that for each copy, we know the site to return.

```
main(){
   …
   f();
   …
   g();
    …
}
```

```
f(){
      …
     g();
      …
    }
```



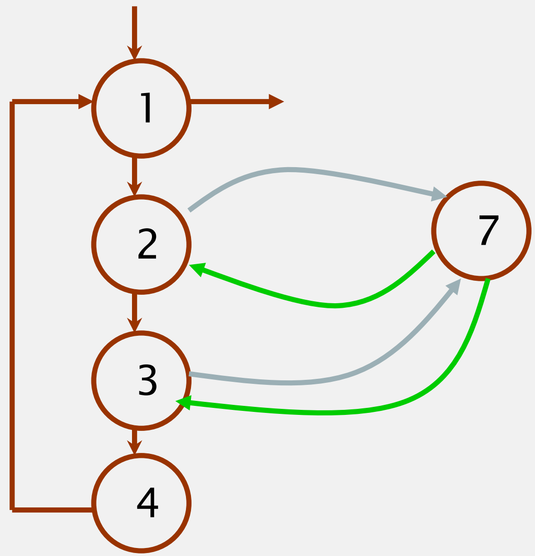
Separate copy of g()

INTER-PROCEDURAL CONTROL FLOW GRAPH (ICFG)

between procedurals.

- Besides the normal intraprocedural control flow graph, additional edges are added connecting
  - Each call site to the beginning of the procedure it calls.
  - The return statement back to the call site.

```
1:    for (i=0; i<n; i++) {
2:        t1= f(0);
3:        t2 = f(243);
4:        x[i] = t1 + t2 + t3;
5:    }
6:  int  f (int v) {
7:    return (v+1);
8:  }
```



CALL GRAPH (CG)

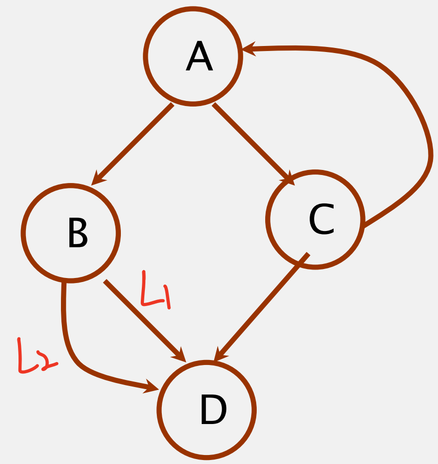
- Each node represents a function; each edge represents a function invocation

```
void A( ) {
   B( );
   C( );
}

void B( ) {
L1:   D( );
L2:   D( );
}

void C ( ) {
   D( );
   A( );
}

void D ( ) {
}
```



TOOLS

- C/C++: LLVM, CIL
- Java: SOOT, Wala
- Binary: Valgrind, Pin

PART III - DATAFLOW ANALYSIS *CS3213 FSE COURSE*

Prof. Abhik Roychoudhury

National University of Singapore

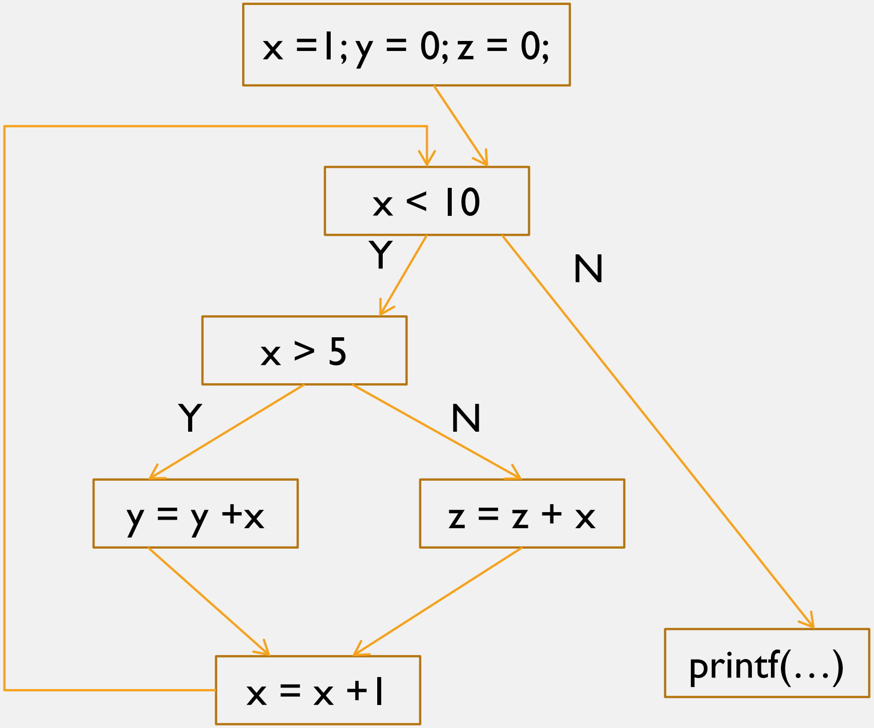
(**Ack**: *Mauro Pezze, University of Lugano, for a couple of slides, and Ilya Sergey, NUS, for one example).*

CONCEPTS LEARNT

- Understand basics of data-flow in programs and the related concepts (def-use pairs, dominators…)
- Understand some analyses that can be performed with the data-flow model of a program
  - The data flow analyses to build models
  - Analyses that use the data flow models
- Use of fixed-point analysis: Static analysis of source code

CONTROL FLOW GRAPH

```
x = 1; y = 0; z = 0;
while (x < 10){
    if (x > 5)
            y = y + x;
    else  z = z + x;
    x = x + 1;
}
printf(…);
```



Nodes of the graph, basic blocks, are maximal code fragments executed without control transfer. The edges denote control transfer.

USE OF CFG

All of the subsequent analysis discussed is applied on the Control flow graph of a program.

The nodes of the graph are basic blocks, and the edges denote control transfer.

So the computation of the data flows will propagate along the edges of the control flow graph.

As a shorthand, while examining the examples, we may show it statement by statement, even though the equations are for nodes in CFG.

DEF-USE PAIR

A **def-use (du) pair** associates a point in a program where a value is produced with a point where it is used

Definition: where a variable gets a value

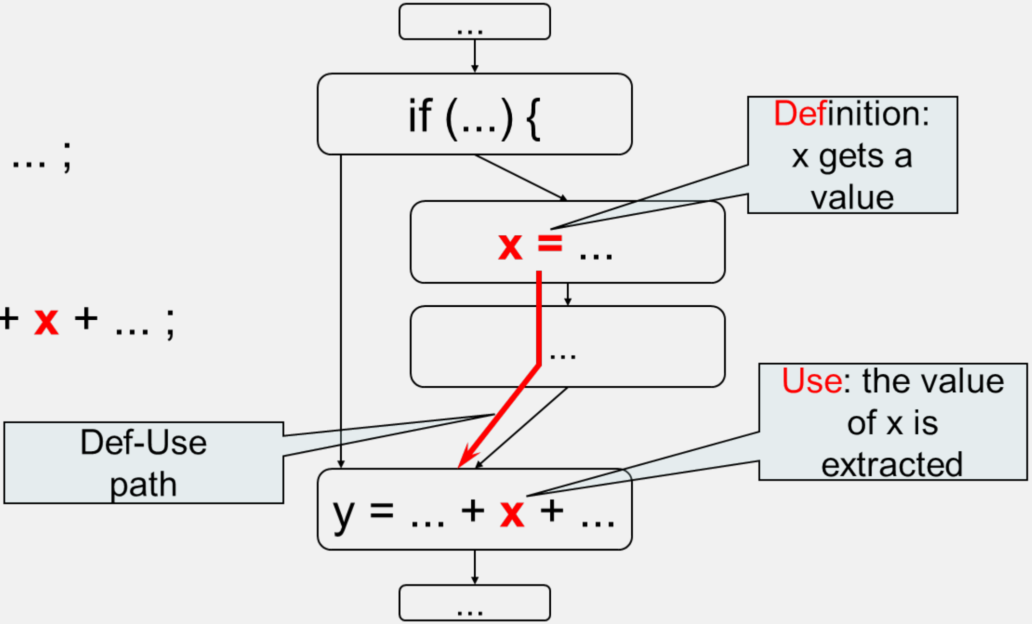
- Variable declaration (often the special value "uninitialized")
- Variable initialization
- Assignment
- Values received by a parameter

Use: extraction of a value from a variable

- Expressions
- Conditional statements
- Parameter passing
- Returns

DEF-USE PAIR

```
...
if (...) {
    x = ... ;
...
}
y = ... + x + ... ;
```



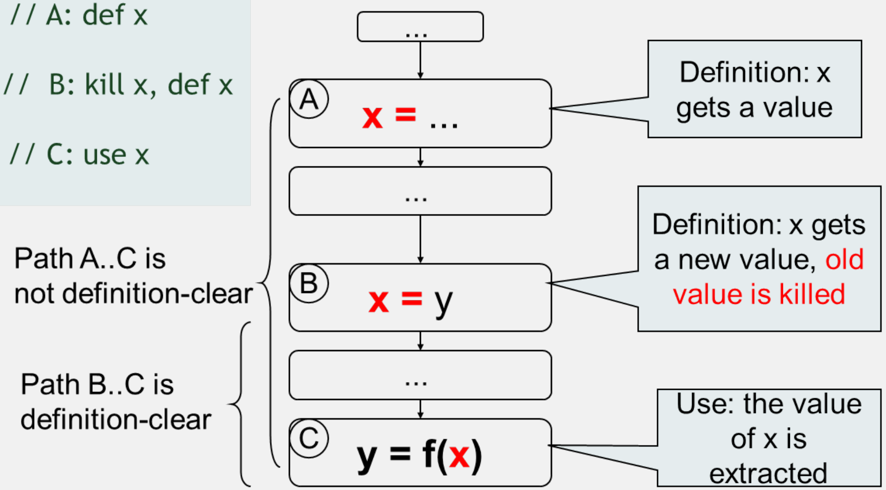
DEF-USE PAIR

- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without another definition of the variable between
  - If, instead, another definition is present on the path, then the latter definition **kills** the former
- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use

and not infeasible path.

DEFINITION-CLEAR OR KILLING

```
x = …      // A: def x
q = …
x = y;     //  B: kill x, def x
z = …
y = f(x);  // C: use x
```

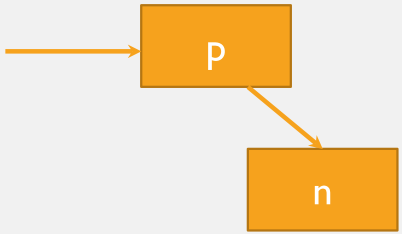


CALCULATING DEF-USE PAIRS

- Definition-use pairs can be defined in terms of paths in the program control flow graph:
  - There is an association (d,u) between a definition of variable v at d and a use of variable v at u iff
    - 1 there is at least one control flow path from d to u
    - 2 with no intervening definition of v.
  - v_d **reaches** u (v_d is a **reaching definition** at u).
  - If a control flow path passes through another definition e of the same variable v, v_e **kills** v_d at that point.
- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges.
- Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

COMPUTING DATAFLOW

- An efficient algorithm for computing reaching definitions (and several other properties) is based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node.
- Suppose we are calculating the reaching definitions of node n, and there is an edge (p,n) from an immediate predecessor node p.
  - If the predecessor node p can assign a value to variable v, then the definition v_p reaches n. We say the definition v_p is generated at p.
  - If a definition v_p of variable v reaches a predecessor node p, and if v is not redefined at that node (in which case we say the v_p is killed at that point), then the definition is propagated on from p to n.



DATAFLOW EQUATIONS

Calculate reaching definitions at E in terms of its immediate predecessor D

```
public class GCD  {
public int gcd(int x, int y) {
    int tmp;              // A: def x, y, tmp
    while (y != 0) {      // B: use y
       tmp = x % y;       // C: def tmp; use x, y
       x = y;             // D: def x; use y
       y = tmp;           // E: def y; use tmp
    }
    return x;             // F: use x
   }
```

$$\text{Reach}(E) = \text{ReachOut}(D)$$

$$\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$$

DATAFLOW EQUATIONS - MERGING OF FLOWS

This line has two predecessors: Before the loop, end of the loop

```
public class GCD  {
public int gcd(int x, int y) {
    int tmp;              // A: def x, y, tmp
    while (y != 0) {      // B: use y
        tmp = x % y;      // C: def tmp; use x, y
        x = y;            // D: def x; use y
        y = tmp;          // E: def y; use tmp
    }
    return x;             // F: use x
   }
```

- $\text{Reach}(B) = \text{ReachOut}(A) \cup \text{ReachOut}(E)$
- $\text{ReachOut}(A) = \text{gen}(A) = \{x_A, y_A, tmp_A\}$
- $\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$

REACHING DEFINITIONS: **RECURSIVE** EQUATIONS

static slicing includes infeasible paths.

All paths.

$$\text{Reach}(n) = \bigcup_{m \in pred(n)} \text{ReachOut}(m)$$

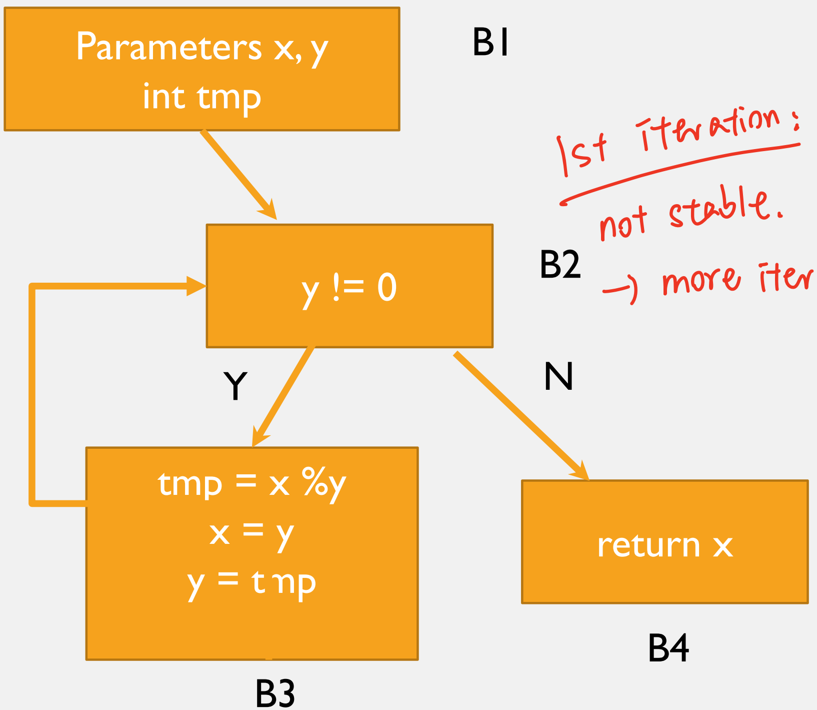
$$\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v_n \mid v \text{ is defined or modified at } n \}$$

$$\text{kill}(n) = \{ v_x \mid v \text{ is defined or modified at } x, x \neq n \}$$

ILLUSTRATION

each iteration: every node.



1st iteration;
not stable.
→ more iteration.

$\text{Reach}_{B1} = \{\}$
$\text{ReachOut}_{B1} = \{\} - \{\} \cup \{x1, y1, tmp1\}$

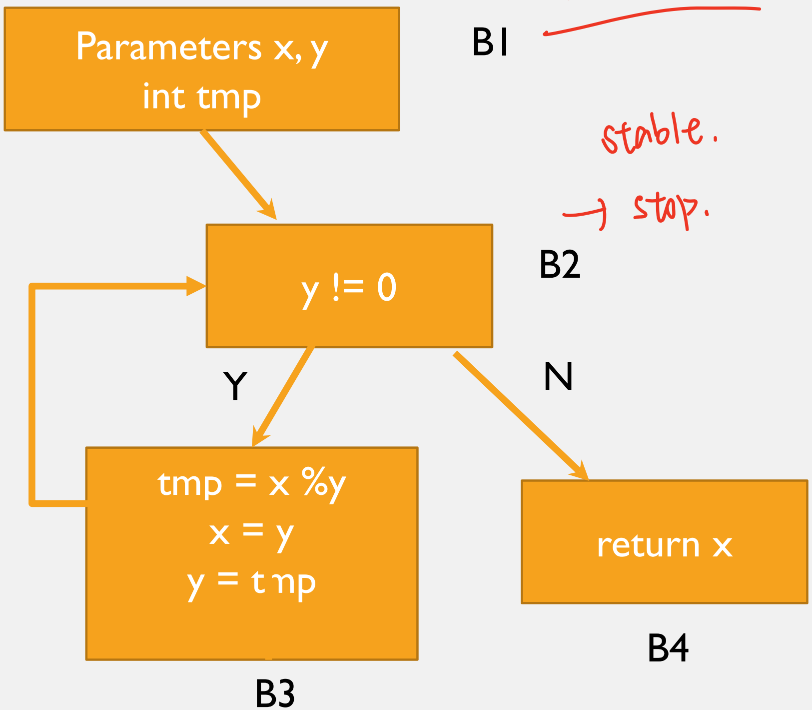
Haven't initialized.

$\text{Reach}_{B2} = \text{ReachOut}_{B1} \cup \underline{\text{ReachOut}_{B3}}$
$= \{x1, y1, tmp1\} \cup \{\}$
$= \{x1, y1, tmp1\}$
$\text{ReachOut}_{B2} = \text{Reach}_{B2} - \{\} \cup \{\}$
$= \text{Reach}_{B2}$

$\text{Reach}_{B3} = \text{ReachOut}_{B2}$
$\text{ReachOut}_{B3} = \text{Reach}_{B3} - \{x1, y1, tmp1\}$
$\cup \{x3, y3, tmp3\}. = \{x3, y3, tmp3\}$

$\text{Reach}_{B4} = \text{ReachOut}_{B2}$
$\text{ReachOut}_{B4} = \text{Reach}_{B4}$

ILLUSTRATION



2nd iteration:

stable.

→ stop.

Static: approx. Inefficient.

Reach_{B1} = {}
ReachOut_{B1} = {} –{} U {x1,y1,tmp1}

iteration and approx.

Reach_{B2} = ReachOut_{B1} U ReachOut_{B3}
={x1, y1, tmp1} U {x3,y3,tmp3}
= {x1, y1, tmp1,x3,y3,tmp3}
ReachOut_{B2} = Reach_{B2} –{} U{}
= Reach_{B2}

Reach_{B3} = ReachOut_{B2}
ReachOut_{B3} = Reach_{B3} – {x1,y1,tmp1}
U{x3,y3,tmp3}. = {x3, y3, tmp3}

Reach_{B4} = ReachOut_{B2}
ReachOut_{B4} = Reach_{B4}

AVAILABLE EXPRESSIONS

All incoming paths.

- An expression e = x op y is *available* at a program point p, if
  - on every path from the entry node of the graph to node p, e is computed at least once, and
  - And there are no definitions of x or y since the most recent occurance of e on the path

DATA FLOW FACTS

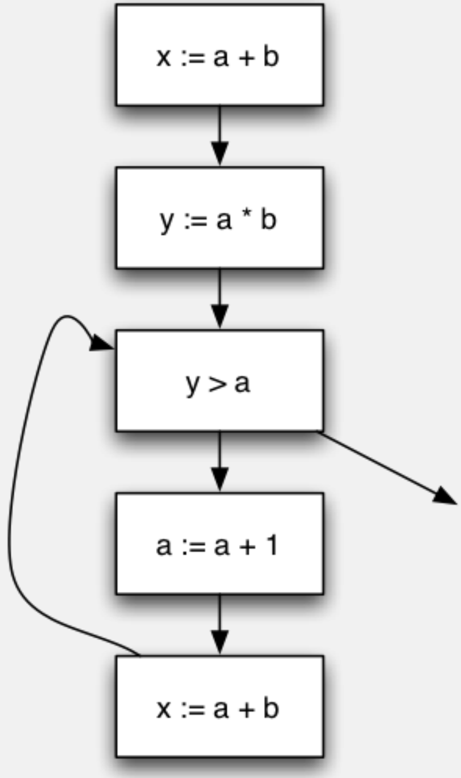
Is expression e available?

Facts:

a + b is available?

a * b is available?

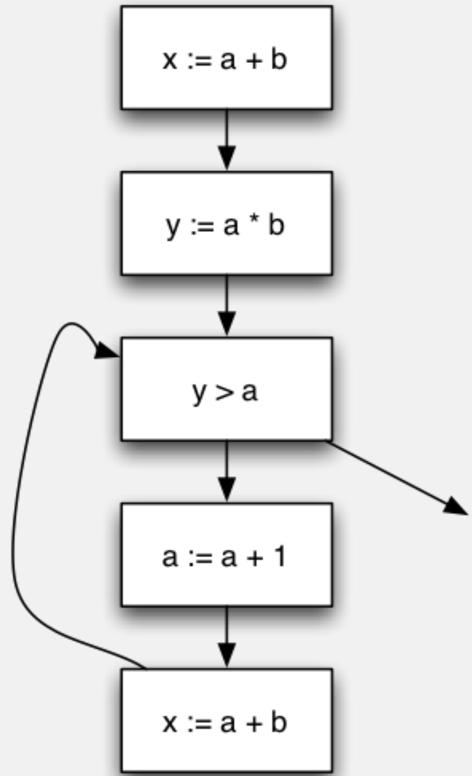
a + 1 is available?



GEN AND KILL

- What is the effect of each statement on the set of facts?

stmt	gen	kill
x = a + b	a + b	
y = a * b	a * b	
a = a + 1		a + b a * b a + 1

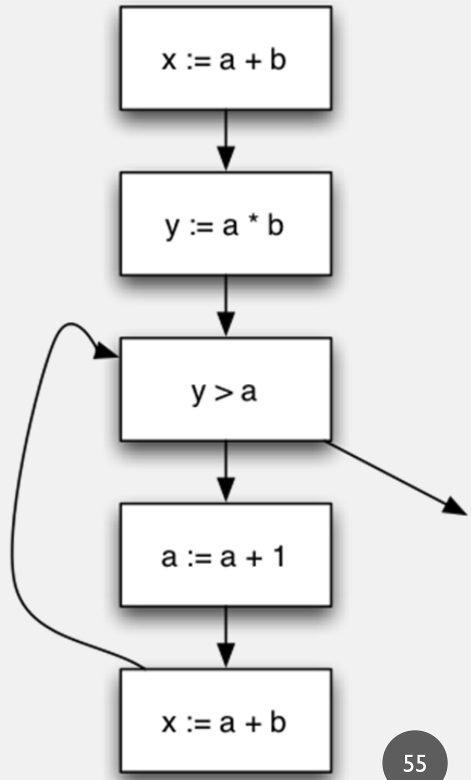


GEN AND KILL

statement on the set of facts?

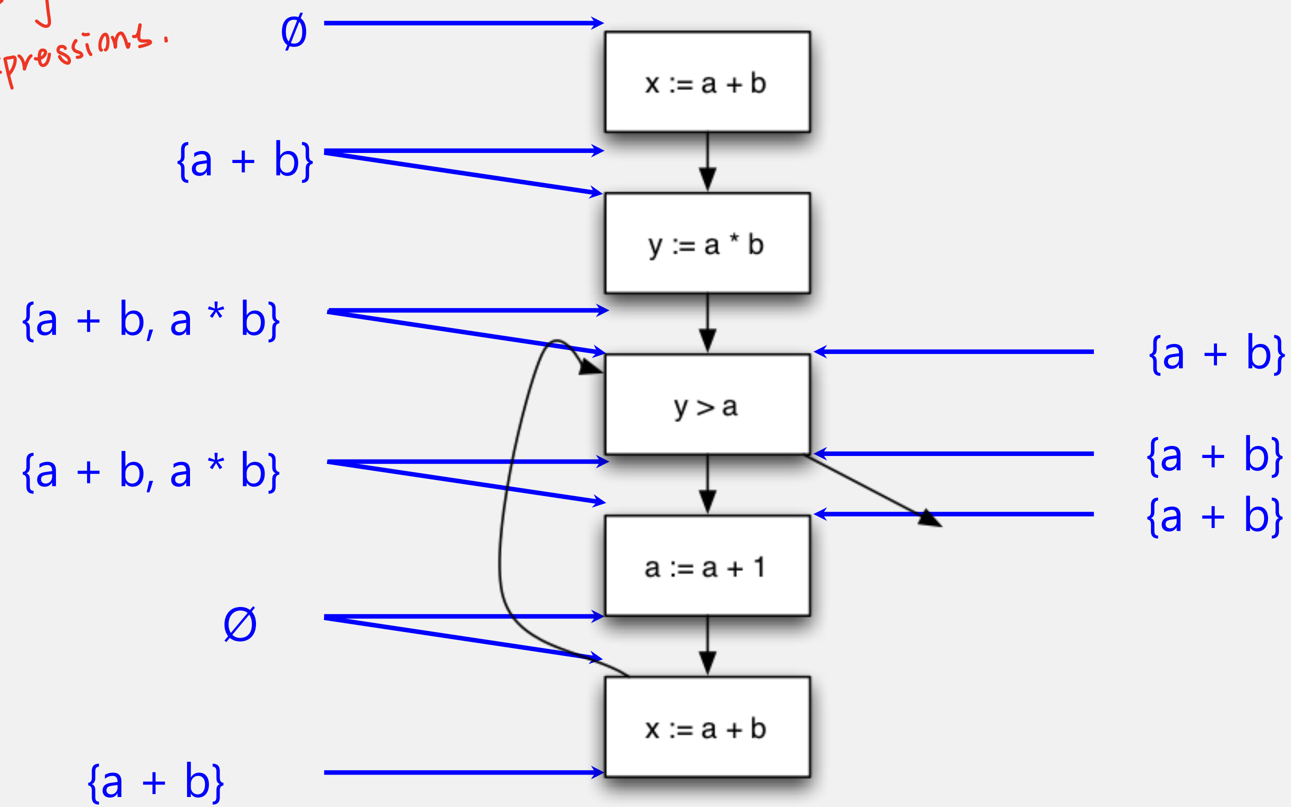
stmt	gen	kill
x = a + b	x a + b	
y = a * b	y a * b	
a = a + 1	a	a + b a * b a + 1

Variables on LHS are also "expressions"
Not shown in these computations



COMPUTING AVAILABLE EXPRESSIONS

Set of available expressions.



TERMINOLOGY

- A *join point* is a program point where two branches meet
- Available expressions is a *forward, must problem*
  - *Forward* = Data Flow from in to out Incoming paths.
  - *Must* = At joint point, property must hold on all paths that are joined.

AVAILABLE EXPRESSIONS: EQUATIONS

$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

gen(n) = { exp | exp is computed at n }

kill(n) = { exp | exp has variables assigned at n }

LIVENESS ANALYSIS

- A variable v is *live at a program* point p if
  - v will be used on some execution path originating from p before v is overwritten

may

Outgoing paths.

从 p 出去的 v is used before overwritten

LIVENESS ANALYSIS: EQUATIONS

- A variable v is *live at a program* point p if
  - v will be used on some execution path originating from p before v is overwritten

$$out(n) = \bigcup_{m \in succ(n)} in(m)$$

m是n的下家.

$$in(n) = (out(n) \setminus kill(n)) \cup gen(n)$$

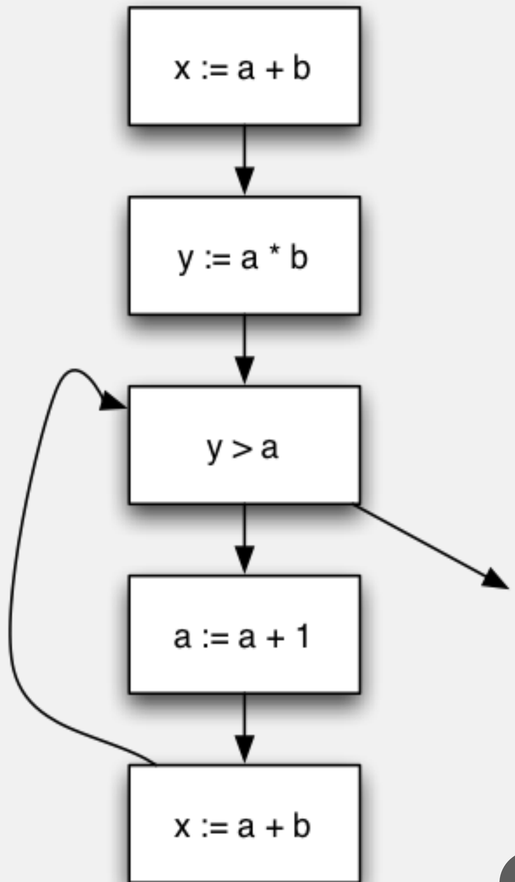
$$gen(n) = \{ v \mid v \text{ is used at } n \}$$

$$kill(n) = \{ v \mid v \text{ is modified at } n \}$$

GEN AND KILL

- What is the effect of each statement on the set of facts?

stmt	gen	kill
x = a + b	a, b	x
y = a * b	a, b	y
y > a	a, y	
a = a + 1	a	a



LIVENESS ANALYSIS: EQUATION TO ALGORITHM

$$out(n) = \bigcup_{m \in succ(n)} in(m)$$

$$in(n) = (out(n) \setminus kill(n)) \cup gen(n)$$

$$gen(n) = \{ v \mid v \text{ is used at } n \}$$

$$kill(n) = \{ v \mid v \text{ is modified at } n \}$$

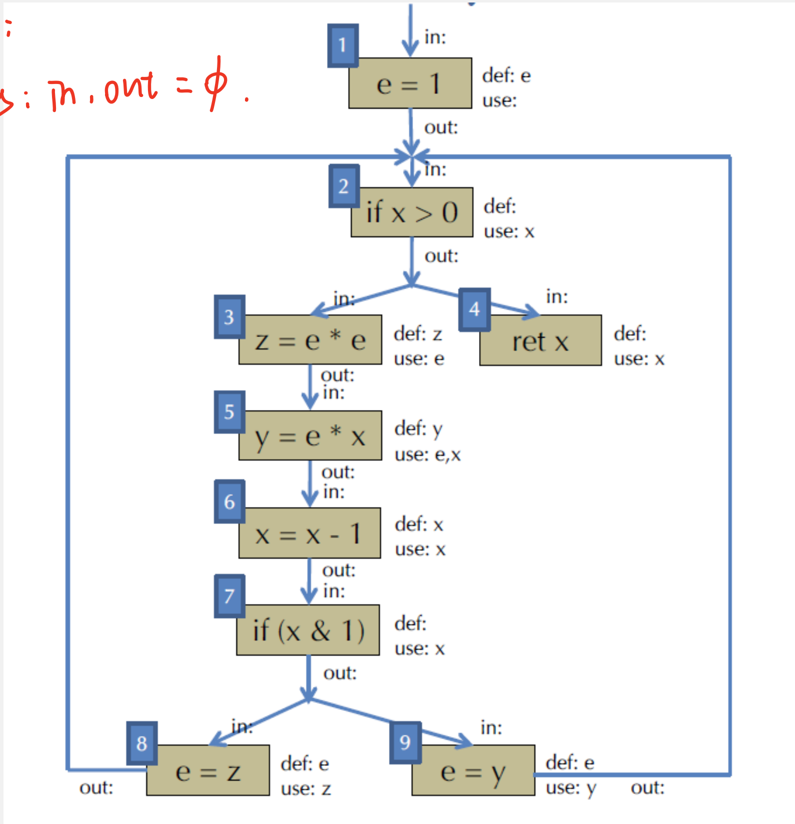
```
for all n{
    in[n] := Ø, out[n] := Ø
}
repeat until no change in 'in' and 'out' {
        for all n{
            out[n] := ∪ m∈succ[n] in[m]
            in[n] := use[n] ∪ (out[n] - def[n])
        }
}
```

EXAMPLE

```
e = 1;
while(x>0) {
  z = e * e;
  y = e * x;
  x = x – 1;
  if (x & 1) {
    e = z;
  } else {
    e = y;
  }
}
return x;
```

Step 0 :

All nodes : in , out = ϕ .



ITERATIVE ANALYSIS: I

first iteration:
out 都是 ф.
in 全 assign 为 use[n]

Each iteration update:

$out[n] := \bigcup_{n' \in succ[n]} in[n']$

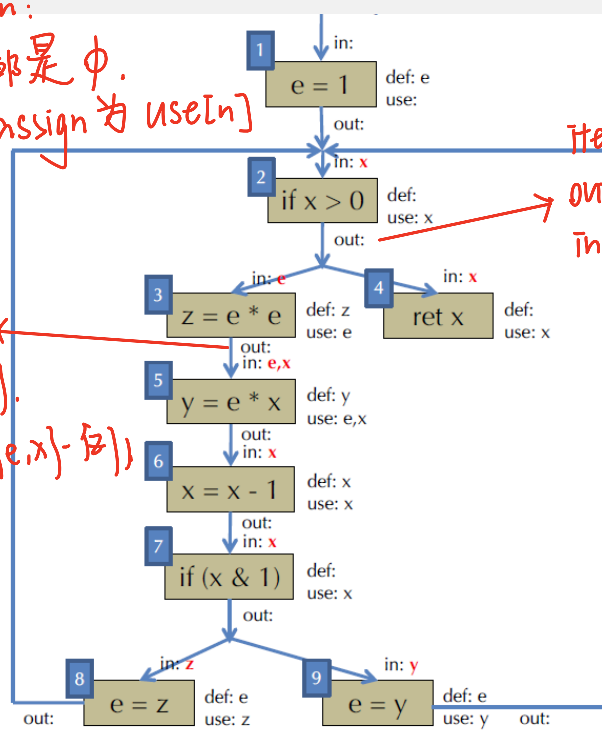
$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 1:

in[2] = x
in[3] = e
in[4] = x
in[5] = e,x
in[6] = x
in[7] = x
in[8] = z
in[9] = y

iteration 2:
out[3] = {e,x}.
in[3] = {e} ∪ ({e,x} - {z})
= {e,x}.

(showing only updates that make a change)



iteration 2:
out[2] = {e,x}.
in[2] = {e,x}.

ITERATIVE ANALYSIS: 2

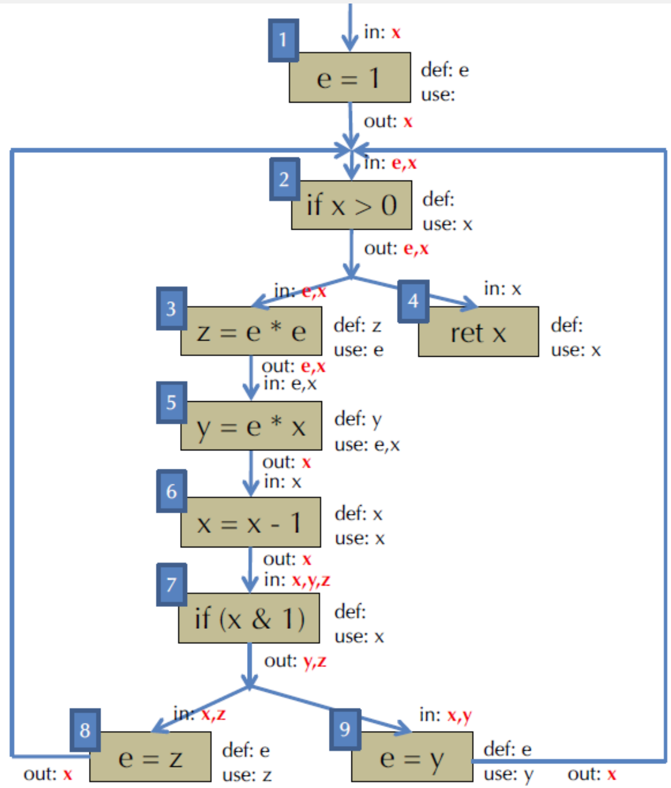
Each iteration update:

$out[n] := \bigcup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 2:

out[1]= x
in[1] = x
out[2] = e,x
in[2] = e,x
out[3] = e,x
in[3] = e,x
out[5] = x
out[6] = x
out[7] = z,y
in[7] = x,z,y
out[8] = x
in[8] = x,z
out[9] = x
in[9] = x,y



ITERATIVE ANALYSIS: 3

Each iteration update:

$out[n] := \cup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 3:

out[1]= e,x

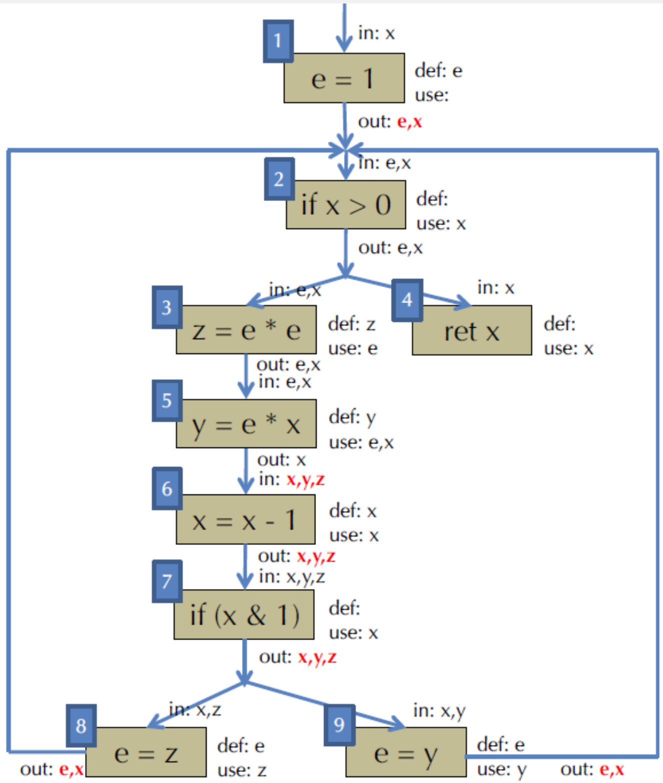
out[6]= x,y,z

in[6]= x,y,z

out[7]= x,y,z

out[8]= e,x

out[9]= e,x



ITERATIVE ANALYSIS: 4

Each iteration update:

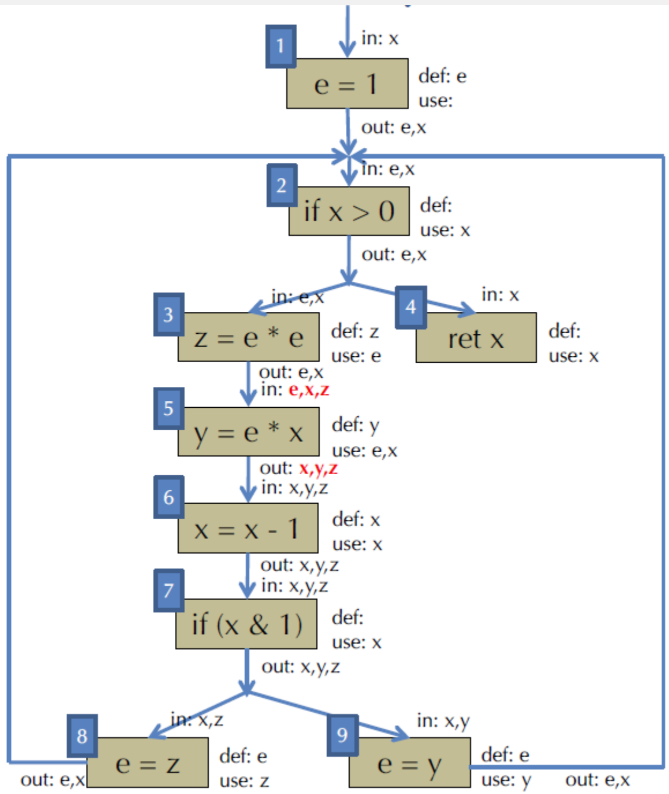
$out[n] := \bigcup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 4:

out[5]= x,y,z

in[5]= e,x,z



ITERATIVE ANALYSIS: 5

Each iteration update:

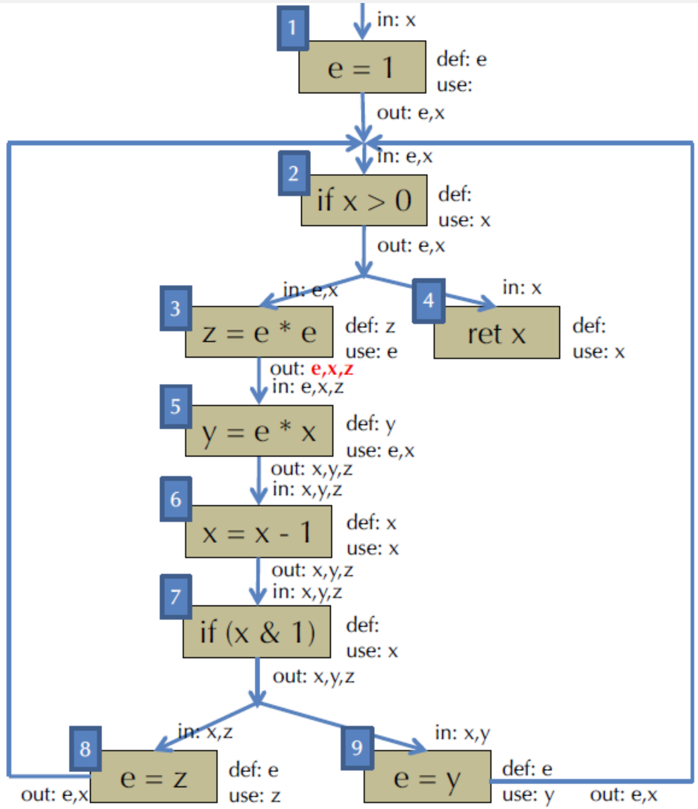
out[n] := $\bigcup_{n' \in succ[n]}$ in[n']

in[n] := use[n] ∪ (out[n] - def[n])

- Iteration 5:

out[3] = e,x,z

Done!



LIVENESS: ALGORITHM EFFICIENCY

```
for all n{
    in[n] := ∅, out[n] := ∅
}
 repeat until no change in 'in' and 'out' {
         for all n{
              out[n] := ⋃ m∈succ[n] in[m]
              in[n] := use[n] ∪ (out[n] - def[n])
          }
}
```

→

```
for all n{  in[n] := ϕ;  out[n] := ϕ; }

w := new queue with all nodes;

Repeat until w is empty{

      n := w.dequeue();

      old_in := in[n];

      out[n] := ∪m∈succ[n]in[m];
      in[n] := use[n] ∪ (out[n] - def[n])
      if  (old_in != in[n]){
            for all m in pred[n], w.enqueue(m);
       }
}
```

WORKLIST ALGORITHM

- Initially all nodes are on the work list, and have default values
  - Default for "any-path" problem is the empty set, default for "all-path" problem is the set of all possibilities (union of all gen sets)
- **While the work list is not empty**
  - Pick any node n on work list; remove it from the list
  - Apply the data flow equations for that node to get new values
  - If the new value is changed (from the old value at that node), then
    - Add successors (for forward analysis) or predecessors (for backward analysis) on the work list
- **Eventually the work list will be empty (because new computed values = old values for each node) and the algorithm stops.**

CLASSIFICATION OF ANALYSES

- Forward/backward: a node's set depends on that of its predecessors/successors
- Any-path/all-path: a node's set contains a value iff it is coming from any/all of its inputs

	Any-path ($\cup$)	All-paths ($\cap$)
Forward (pred)	Reach	Avail
Backward (succ)	Live	"inevitable"

REACHING DEFINITIONS

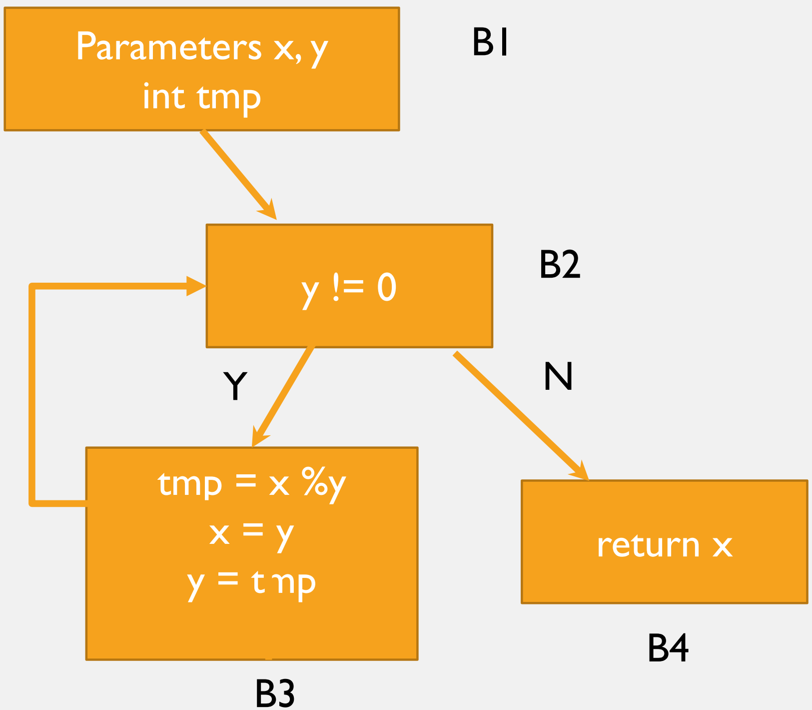
- A *definition of a variable* v is an assignment to v
- A definition of variable v reaches point p if
  - There is no intervening assignment to v
- Also called *def-use* information
- What kind of problem?
  - Forward or backward? Forward
  - May or must? May or any-path

ITERATIVE SOLUTION OF RECURSIVE EQUATIONS

- Initialize values (first estimate of answer)
  - For "any path" problems, first guess is "nothing" (empty set) at each node
  - For "all paths" problems, first guess is "everything" (set of all possible values = union of all "gen" sets)
- Repeat until nothing changes
  - Pick some node and recalculate (new estimate)

This will converge on a "fixed point" solution where every new calculation produces the same value as the previous guess.

ILLUSTRATION: RECAP



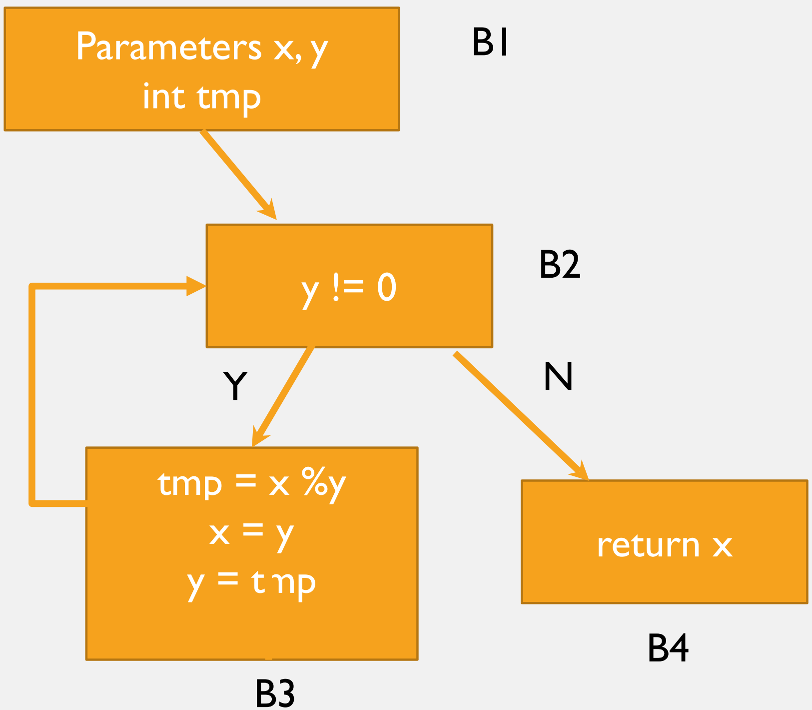
$Reach_{B1} = \{\}$
$ReachOut_{B1} = \{\} - \{\} \cup \{x1, y1, tmp1\}$

$Reach_{B2} = ReachOut_{B1} \cup ReachOut_{B3}$
$= \{x1, y1, tmp1\} \cup \{\}$
$= \{x1, y1, tmp1\}$
$ReachOut_{B2} = Reach_{B2} - \{\} \cup \{\}$
$= Reach_{B2}$

$Reach_{B3} = ReachOut_{B2}$
$ReachOut_{B3} = Reach_{B3} - \{x1, y1, tmp1\}$
$\cup \{x3, y3, tmp3\}. = \{x3, y3, tmp3\}$

$Reach_{B4} = ReachOut_{B2}$
$ReachOut_{B4} = Reach_{B4}$

ILLUSTRATION: RECAP



$Reach_{B1}$ = {}
$ReachOut_{B1}$ = {} –{} U {x1,y1,tmp1}

$Reach_{B2}$ = $ReachOut_{B1}$ U $ReachOut_{B3}$
=\{x1, y1, tmp1\} U \{x3,y3,tmp3\}
= \{x1, y1, tmp1,x3,y3,tmp3\}
$ReachOut_{B2}$ = $Reach_{B2}$ –{} U{}
= $Reach_{B2}$

$Reach_{B3}$ = $ReachOut_{B2}$
$ReachOut_{B3}$ = $Reach_{B3}$ – {x1,y1,tmp1}
U{x3,y3,tmp3}. = {x3, y3, tmp3}

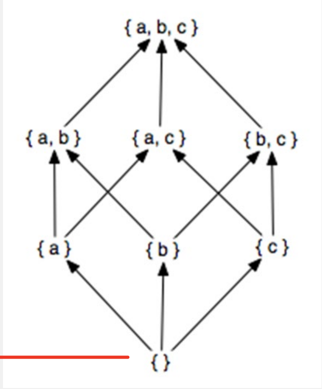
$Reach_{B4}$ = $ReachOut_{B2}$
$ReachOut_{B4}$ = $Reach_{B4}$

ABSTRACT DOMAIN FOR FLOW ANALYSIS

- Flow equations must be monotonic 单调
  - Initialize to the bottom element of a lattice of approximations
  - Each new value that changes must move up the lattice
- Typically: Powerset lattice
  - Bottom is empty set, top is universe
  - Or empty at top for all-paths analysis

Monotonic: y > x implies f(y) ≥ f(x)

(where f is application of the flow equations on values from successor or predecessor nodes, and ">" is movement up the lattice)



any-path starts from empty.

ALIASING IN ANALYSIS

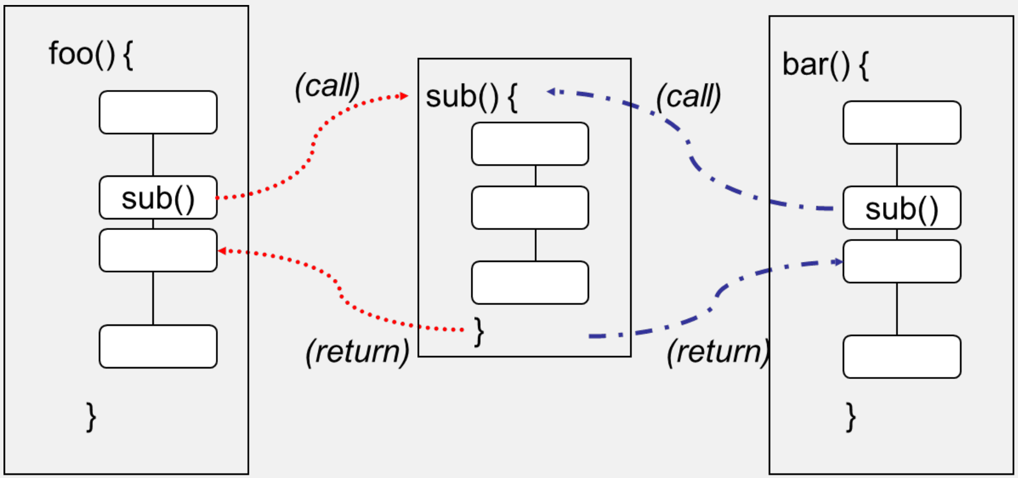
- Arrays and pointers introduce uncertainty:
  Do different expressions access the same storage?
  - a[i] same as a[k] when i = k
  - a[i] same as b[i] when a = b (**aliasing**)
- The uncertainty is accomodated depending to the kind of analysis
  - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
  - All-path: vice versa

NATURE OF ANALYSES

- Intraprocedural
  - Within a single method or procedure
    - as described so far
- … or Inter-procedural
  - Across several methods (and classes) or procedures

- Cost/Precision trade-offs for inter-procedural analysis are critical, and difficult
  - context sensitivity
  - flow-sensitivity

CONTEXT-SENSITIVE ANALYSIS



A **context-sensitive** (interprocedural) analysis distinguishes sub() called from foo() from sub() called from bar();
A **context-insensitive** (interprocedural) analysis does not separate them, as if foo() could call sub() and sub() could then return to bar()

FLOW-SENSITIVE ANALYSIS

- Reach, Avail, etc. were flow-sensitive, intraprocedural analyses
  - They considered ordering and control flow decisions
  - Within a single procedure or method, this is (fairly) cheap — $O(n^3)$ for n CFG nodes
- Many interprocedural flow analyses are flow-insensitive
  - $O(n^3)$ would not be acceptable for all the statements in a program!
    - Though $O(n^3)$ on each individual procedure might be ok
  - Often flow-insensitive analysis is good enough ... consider type checking as an example

SUMMARY

- Data flow analysis detect patterns on programs (and their Control Flow Graph)
  - Nodes initiating the pattern
  - Nodes terminating it
  - Nodes that may interrupt it
- Often, but not always, about flow of information (dependence)
- Pros:
  - Can be implemented by efficient iterative algorithms
  - Widely applicable (not just for classic "data flow" properties)
- Limitations:
  - Unable to distinguish feasible from **infeasible paths**
  - Merging of estimates from paths: approximation in reporting, **false alarms** ...
    - Key concern for industrial usage, though widely used in programming environments.

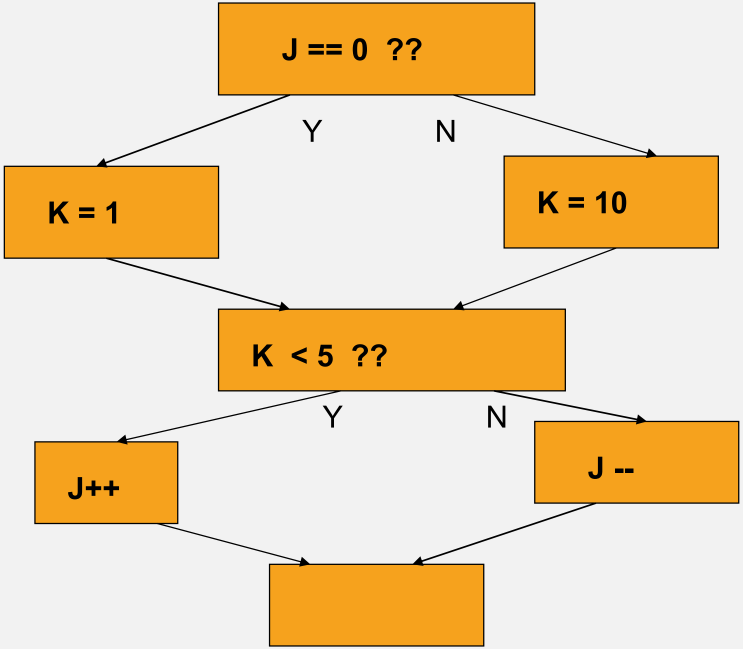
STATIC ANALYSIS PRACTICE

CS3213

Abhik Roychoudhury

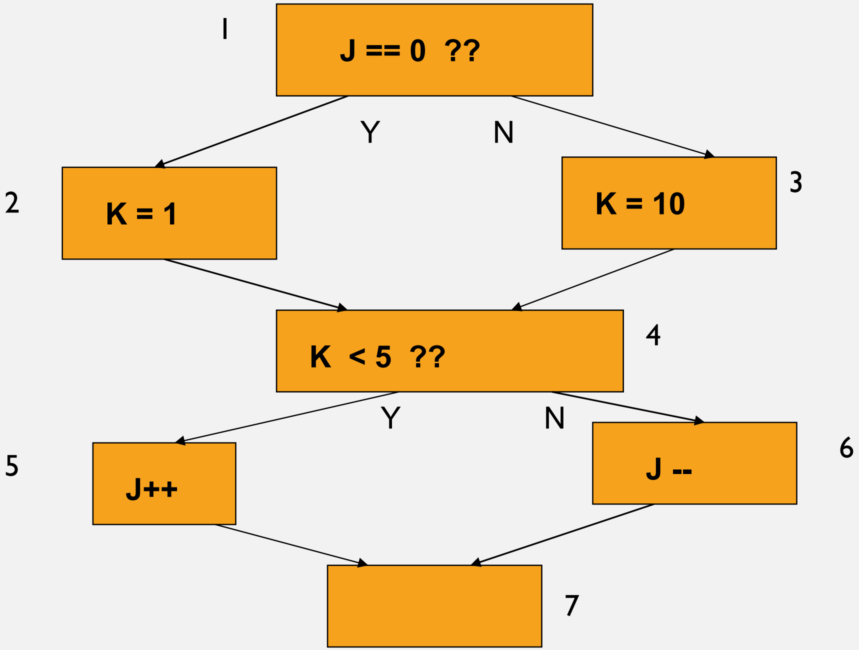
National University of Singapore

TESTING FOR INFEASIBILITY



1. How many paths in total?
2. How many feasible paths?
3. How many infeasible paths?

TESTING FOR INFEASIBILITY



4 paths in total

2 feasible paths

2 infeasible paths

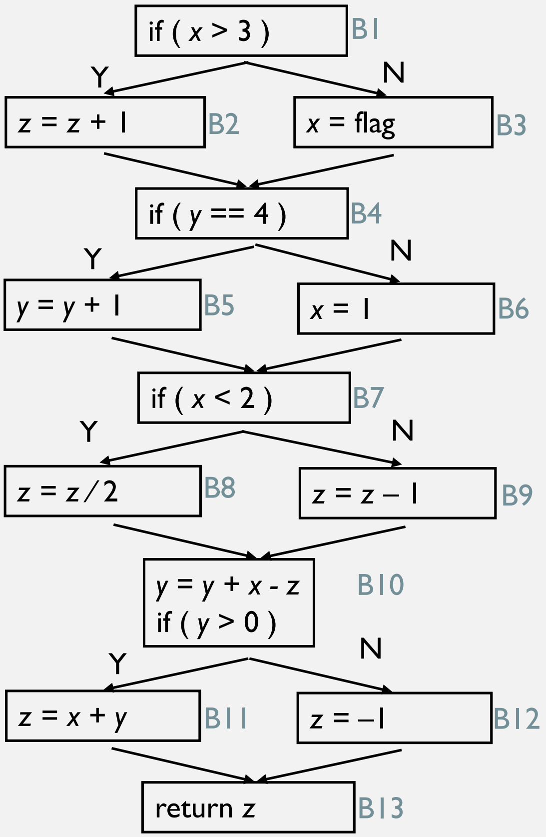
1-2-4-5-7 feasible
1-2-4-6-7 infeasible
1-3-4-5-7 infeasible
1-3-4-6-7 feasible

CONFLICT PAIRS

AB Conflict:
(B6, B7 → B9)

BB Conflict:
(B1 → B2, B7 → B8)

How many paths?
How many feasible paths?
How many infeasible paths?



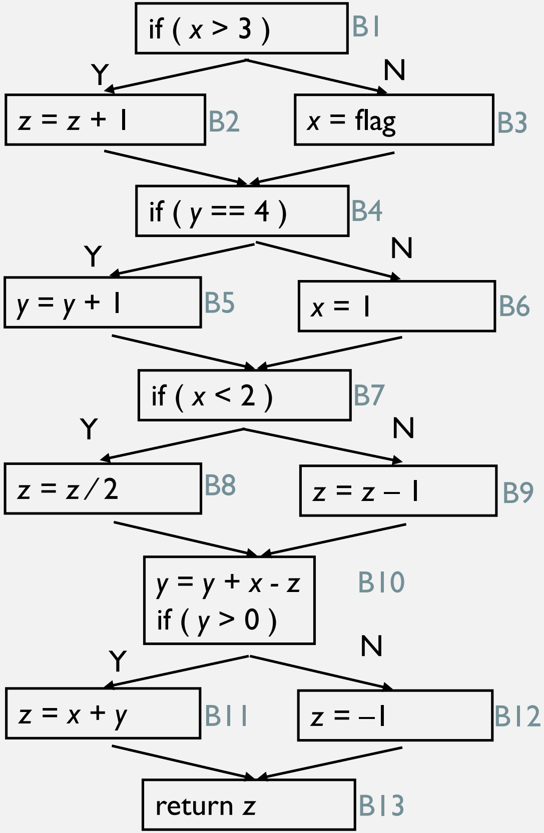
CONFLICT PAIRS

AB Conflict:
(B6, B7 → B9)

BB Conflict:
(B1 → B2, B7 → B8)

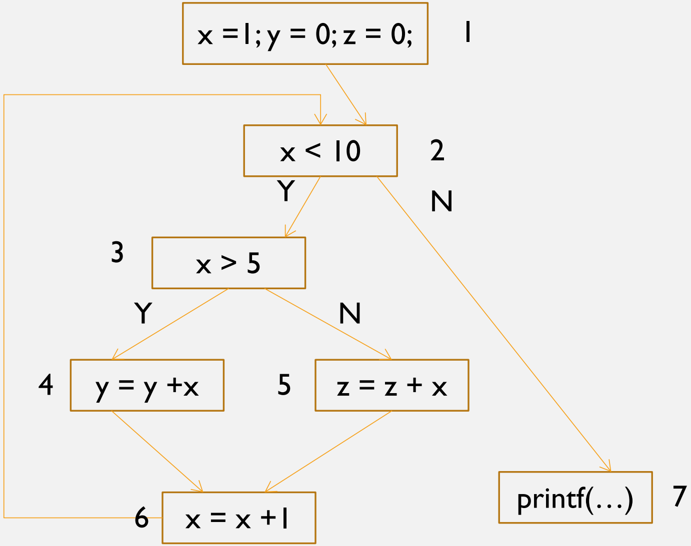
How many paths? *16*
How many feasible paths?
How many infeasible paths?

Please check the suggested answers and see if they are correct.



CONTROL FLOW GRAPH

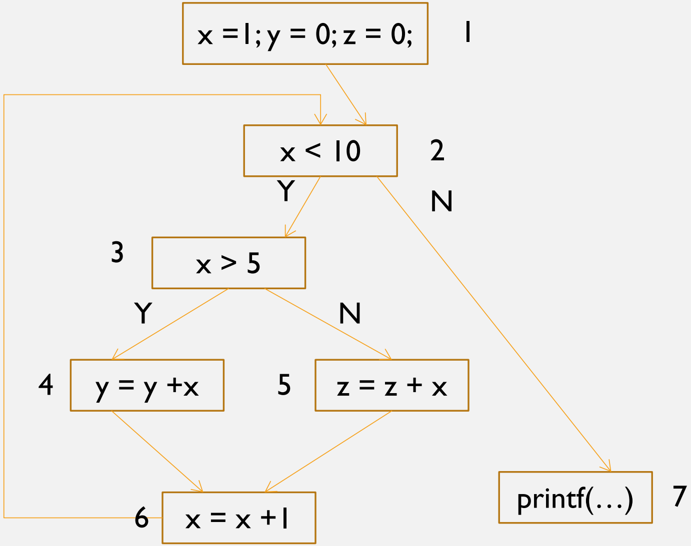
```
x = 1; y = 0; z = 0;
while (x < 10){
    if (x > 5)
            y = y + x;
    else  z = z + x;
    x = x + 1;
}
printf(…);
```



1. Describe sample infeasible path patterns. 2. How many feasible paths are there?

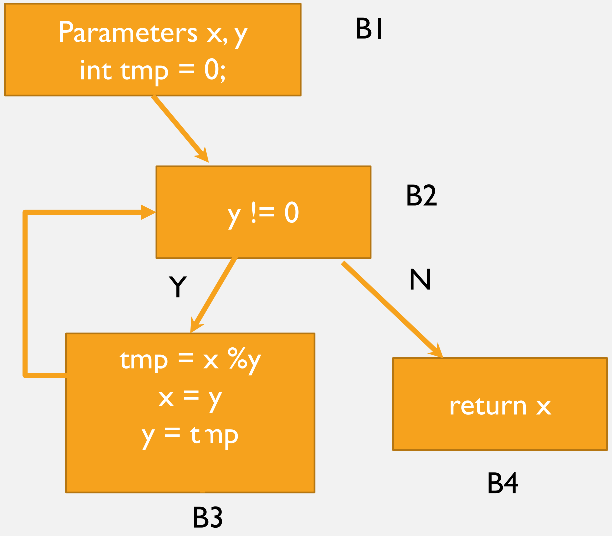
CONTROL FLOW GRAPH

```
x = 1; y = 0; z = 0;
while (x < 10){
    if (x > 5)
            y = y + x;
    else  z = z + x;
    x = x + 1;
}
printf(…);
```



3-4-6-2-3-5 is an infeasible path. Any path containing it is also an infeasible path.
There is only 1 feasible path.

ILLUSTRATION IN CLASS



First iteration Only shown

$Reach_{B1}$ = {}
$ReachOut_{B1}$ = {} –{} U {x1,y1,tmp1}

$Reach_{B2}$ = $ReachOut_{B1}$ U $ReachOut_{B3}$
={x1, y1, tmp1} U {}
= {x1, y1, tmp1}
$ReachOut_{B2}$ = $Reach_{B2}$ –{} U{}
= $Reach_{B2}$

$Reach_{B3}$ = $ReachOut_{B2}$
$ReachOut_{B3}$ = $Reach_{B3}$ – {x1,y1,tmp1}
U{x3,y3,tmp3}. = {x3, y3, tmp3}

$Reach_{B4}$ = $ReachOut_{B2}$
$ReachOut_{B4}$ = $Reach_{B4}$

AVAILABLE EXPRESSIONS: EQUATIONS

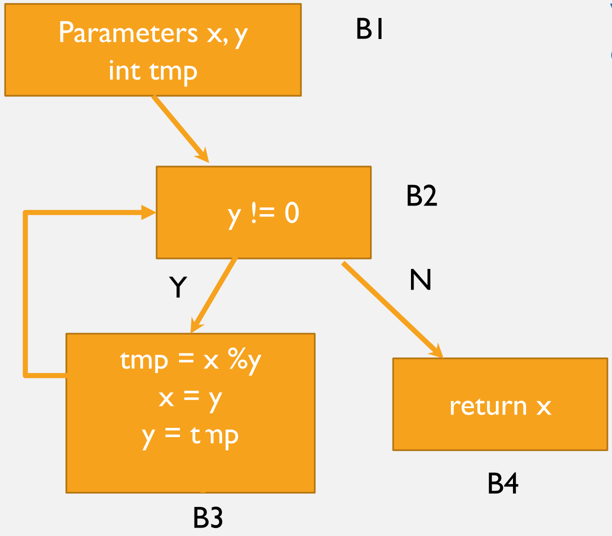
$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

gen(n) = { exp | exp is computed at n }

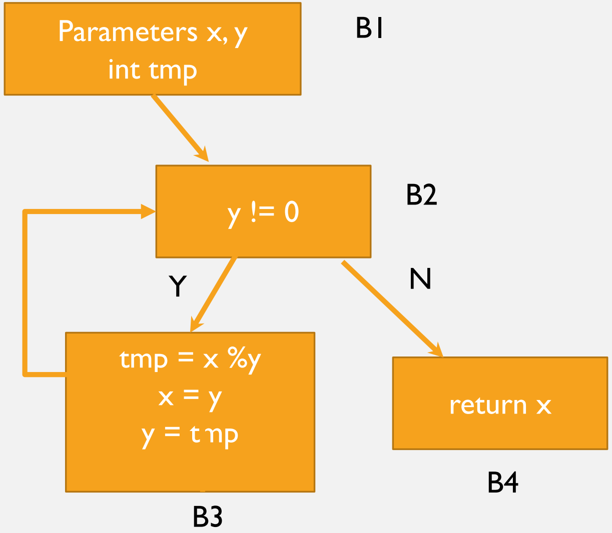
kill(n) = { exp | exp has variables assigned at n }

EXERCISE



Work out the available expressions computation on this control flow graph

ILLUSTRATION OF EXERCISE



$Avail_{B1} = \{\}$
$AvailOut_{B1} = \{\} - \{\} \cup \{\}$

$Avail_{B2} = AvailOut_{B1} \cap AvailOut_{B3}$
$= \{\} \cap \{x\%y, tmp\}$
$= \{\}$
$AvailOut_{B2} = Avail_{B2} - \{\} \cup \{\}$
$= \{\}$

$Avail_{B3} = AvailOut_{B2}$
$AvailOut_{B3} = Avail_{B3} - \{y, x\%y\} \cup \{x\%y, y, tmp\} = \{x\%y, tmp\}$

$Avail_{B4} = AvailOut_{B2}$
$AvailOut_{B4} = Avail_{B4} - \{\} \cup \{x\} = \{x\}$

Does not show all the iterations of the computation

LIVENESS ANALYSIS: EQUATIONS

- A variable v is *live at a program* point p if
  - v will be used on some execution path originating from p before v is overwritten

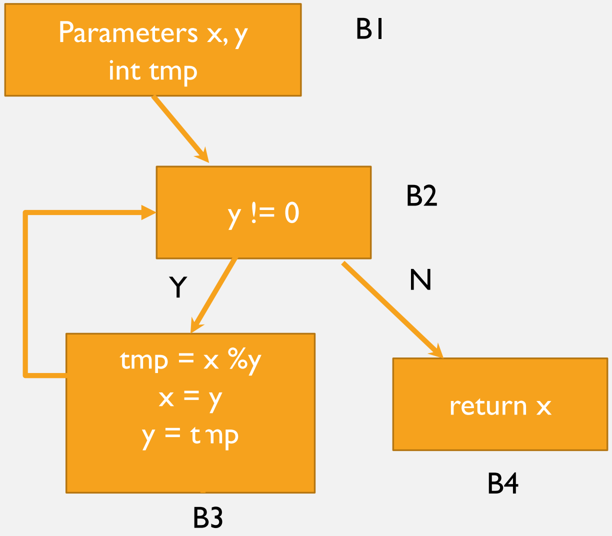
$$out(n) = \bigcup_{m \in succ(n)} in(m)$$

$$in(n) = (out(n) \setminus kill\ (n)) \cup gen(n)$$

$$gen(n) = \{\ v \mid v \text{ is used at } n\ \}$$

$$kill(n) = \{\ v \mid v \text{ is modified at } n\ \}$$

ANOTHER EXERCISE: LIVENESS ANALYSIS



out_{B1} = in_{B2} = {}
in_{B1} = {} –{} U {x,y,tmp} = {x,y,tmp}

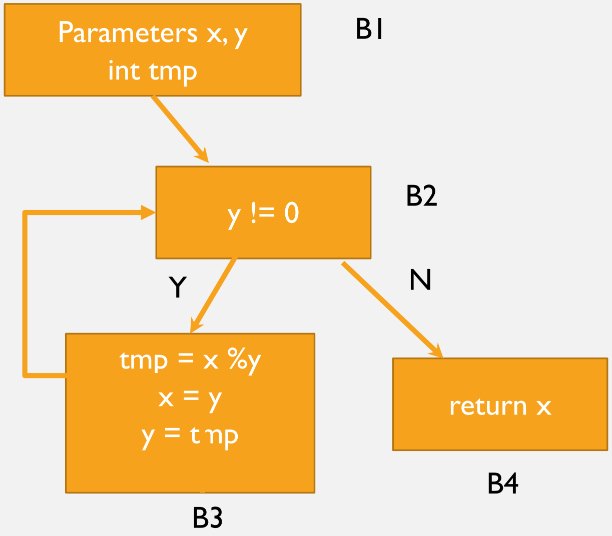
out_{B2} = in_{B3} U in_{B4}
={} U {} = {}
in_{B2} = out_{B2} – {} U{y} = {y}

out_{B3} = in_{B2} = {}
in_{B3} = out_{B3} – {x,y,tmp} U{x,y,tmp}

This is showing the first iteration of the fixed-point computation.

Try to complete the next iterations !!

ANOTHER EXERCISE: LIVENESS ANALYSIS



out_{B1} = in_{B2} = {}
in_{B1} = {} –{} U {x,y,tmp} = {x,y,tmp}

out_{B2} = in_{B3} U in_{B4}
= **{x,y,tmp}** U {} = **{x,y,tmp}**
in_{B2} = out_{B2} – {} U{y} = **{x,y,tmp}**

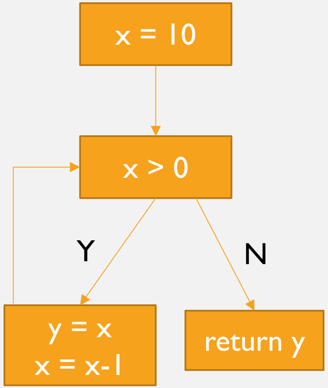
out_{B3} = in_{B2} = **{x,y,tmp}**
in_{B3} = out_{B3} – {x,y,tmp} U{x,y,tmp}
= {x,y,tmp}

This is showing the first iteration of the fixed-point computation.

Try to complete the next iterations !!

INFEASIBLE PATHS & ANALYSIS

```
x = 10;
while  (x > 0){
        y = x;
        x = x – 1;
}
return y;
```



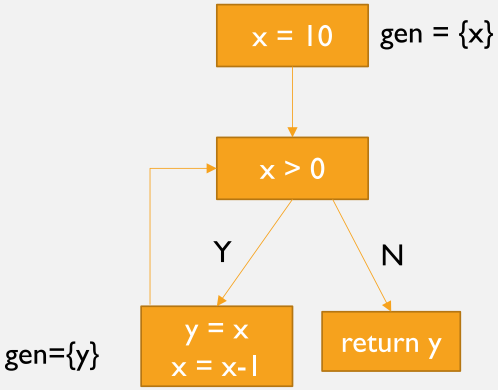
$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

gen(n) = { exp | exp is computed at n }

kill(n) = { exp | exp has variables assigned at n }

INFEASIBLE PATHS & ANALYSIS



$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

gen(n) = { exp | exp is computed at n }

kill(n) = { exp | exp has variables assigned at n }

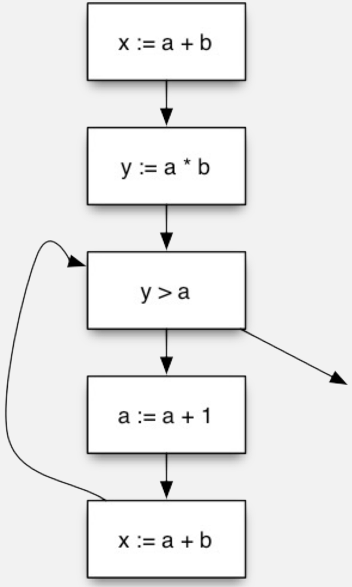
gen(n) is expressions that become available at n
kill(n) is expressions that cease to be available

GEN AND KILL

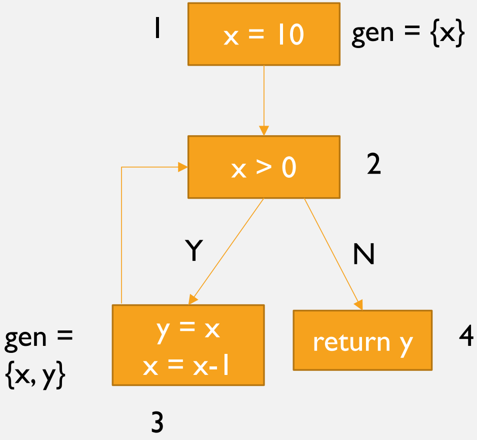
- What is the effect of each statement on the set of facts?

stmt	gen	kill
x = a + b	a + b	
y = a * b	a * b	
a = a + 1		a + b a * b a + 1

Leaves out the variables as expressions



INFEASIBLE PATHS & ANALYSIS



$$\text{Avail}(n) = \bigcap_{m \in pred(n)} \text{AvailOut}(m)$$

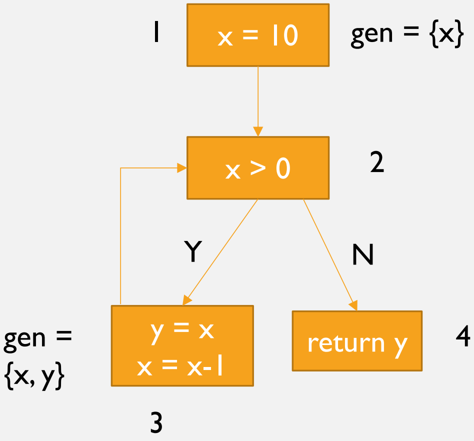
$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

gen(n) = { exp | exp is computed at n }

kill(n) = { exp | exp has variables assigned at n }

Is y available at the return node?
Why or why not? Are there infeasible paths?

INFEASIBLE PATHS & ANALYSIS



$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

gen(n) = { exp | exp is computed at n }

kill(n) = { exp | exp has variables assigned at n }

y is not available at the return node

1-2-4 is an incoming path to y. In this path y is not initialized and not available.

Such infeasible paths also get considered in static analysis computations, leading to over-approximation in static analysis results.