White Box Testing (II)

Data Flow Testing

Data flow testing

- Testing based on data flow characteristics and **d**ata **d**ependency **a**nalysis (DDA): test the correct handling of data dependencies during program execution

- Result-oriented testing (as opposed to control in control oriented testing)

Data Flow Testing

- Data flow testing as a powerful tool to detect improper use of data values due to coding errors.

```
main() {
          int x;
          if (x==42){ ...}
    }
```

Data-Flow Testing

- **Data-flow testing** uses the control flow graph to explore the unreasonable things that can happen to data (*i.e.*, anomalies).
- Consideration of data-flow anomalies leads to test path selection strategies that fill the gap between *complete* path testing and branch testing.

Data flow testing

- **Static data flow testing**: by analyzing source code. Reveal potential anomalies through data flow anomaly analysis

- **Dynamic data flow testing**: involves identifying program paths from source code based on a class of data flow testing criteria

Data flow anomaly

anomaly – abnormal way concerning the generation and usage of data. Could be manifestations of potential programming errors. They may not lead to failures.

defined and then defined again (type 1)

```
x=f1(y)
x=f2(z)
```

- Computation is redundant
- The first statement has a fault , e.g., w=f1(y)
- The second statement has a fault, e.g., v = f1(z)
- Missing statement between the two, e.g., v =f3(x)

Data flow anomaly

Undefined but referenced (used) (type 2)

```
x=x-y-w
```

variable `w` has not been initialized

defined but not referenced (used) (type 3)

```
x=f(x,y)
```

the value `x` is not use in any subsequent computing

Data definition, data use, data dependencies

Data (D) definition

Data creation, initialization, assignment (explicitly and implicitly). **D operation**; D(d)

D-operation is *destructive* (whatever was stored in the data item could be destroyed)

Data use (U operation)

Data use in *computation* or in *predicate* [data referenced]

C-use or **P-use** (**U-use**)

U (u) operation is n*on-destructive*

(d) Defined Objects

- An object (*e.g.*, variable) is **defined** when it:
  - appears in a data declaration
  - is assigned a new value
  - is a file that has been opened
  - is dynamically allocated

(u) Used Objects

- An object is **used** when it is part of a computation or a predicate.
- A variable is used for a computation **(c)** when it appears on the RHS (sometimes even the LHS in case of array indices) of an assignment statement.
- A variable is used in a predicate **(p)** when it appears directly in that predicate.

Example: Definition and Uses

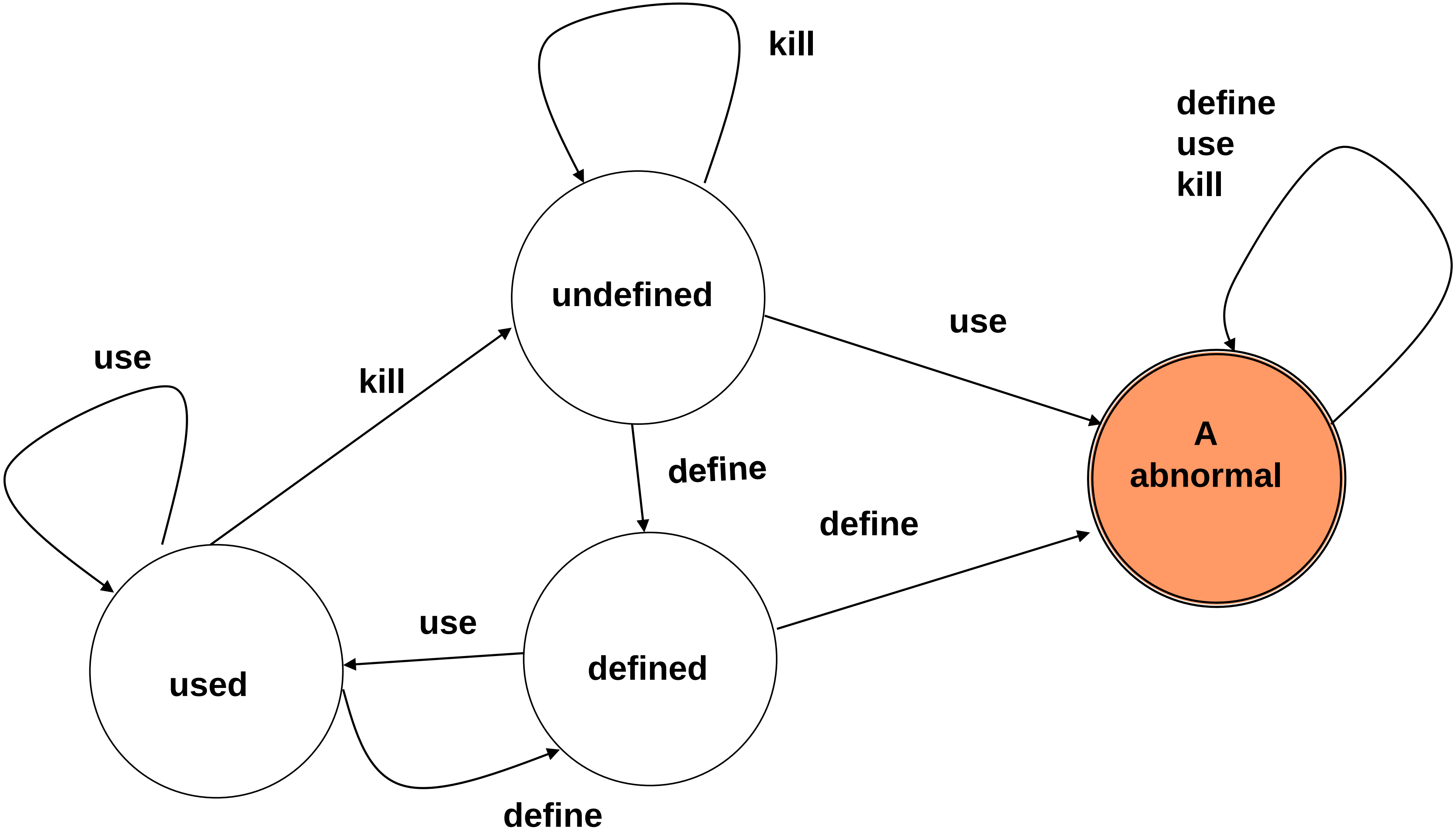
```
1.   read (x, y);
2.   z = x + 2;
3.   if (z < y)
4            w = x + 1;
     else
5.           y = y + 1;
6.   print (x, y, w, z);
```

Example: Definition and Uses

```
1.   read (x, y);
2.   z = x + 2;
3.   if (z < y)
4          w = x + 1;
     else
5.         y = y + 1;
6.   print (x, y, w, z);
```

Def	*C-use*	*P-use*
x, y		
z	x	
		z, y
w	x	
y	y	
	x, y, w, z	

Variables and their states



Relations on data (1)

Examination of time-sequenced pairs of defined (d), used (u), and killed (k)

dd - not invalid but *suspicious*. Likely a programming error

du - perfectly correct --normal case

dk - not invalid but likely a programming error

ud - acceptable

uu - acceptable; ignored given non-destructive nature of the “u” operation

uk - acceptable

Relations on data (2)

kd - acceptable

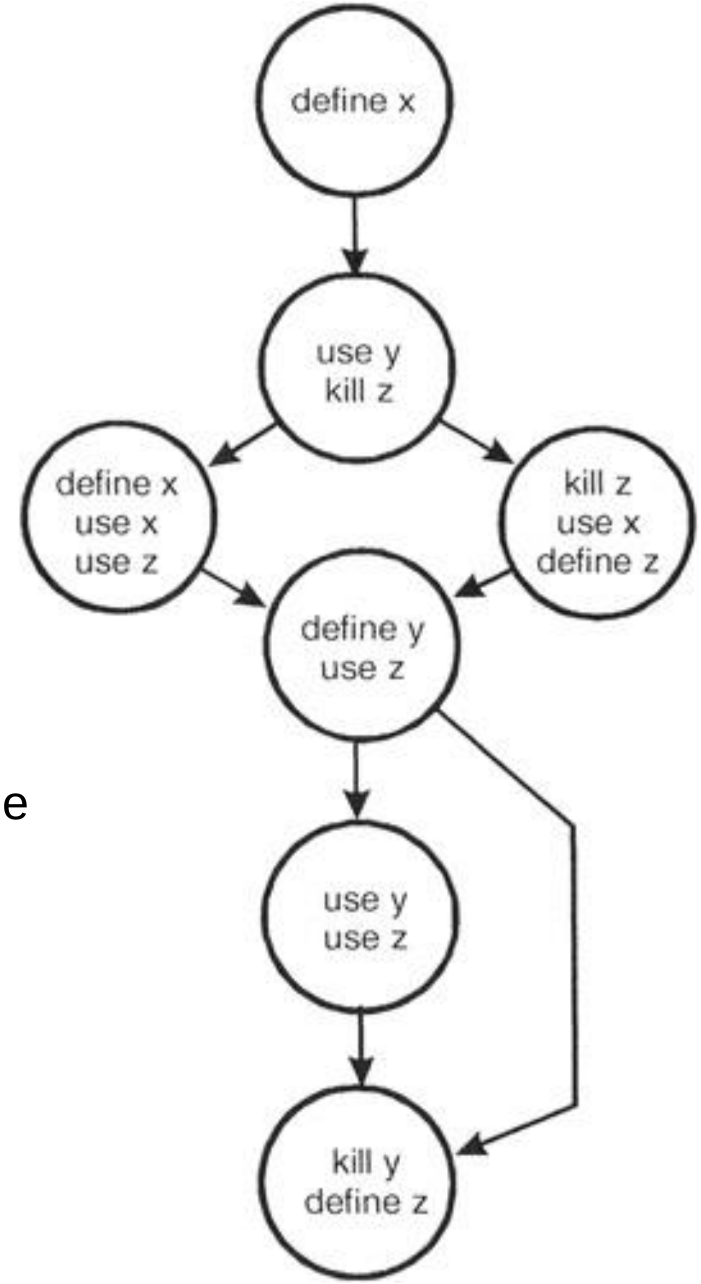
ku - a serious defect. Using a variable that does not exist or is undefined is always an error.

kk - probably a programming error.

Example(1)

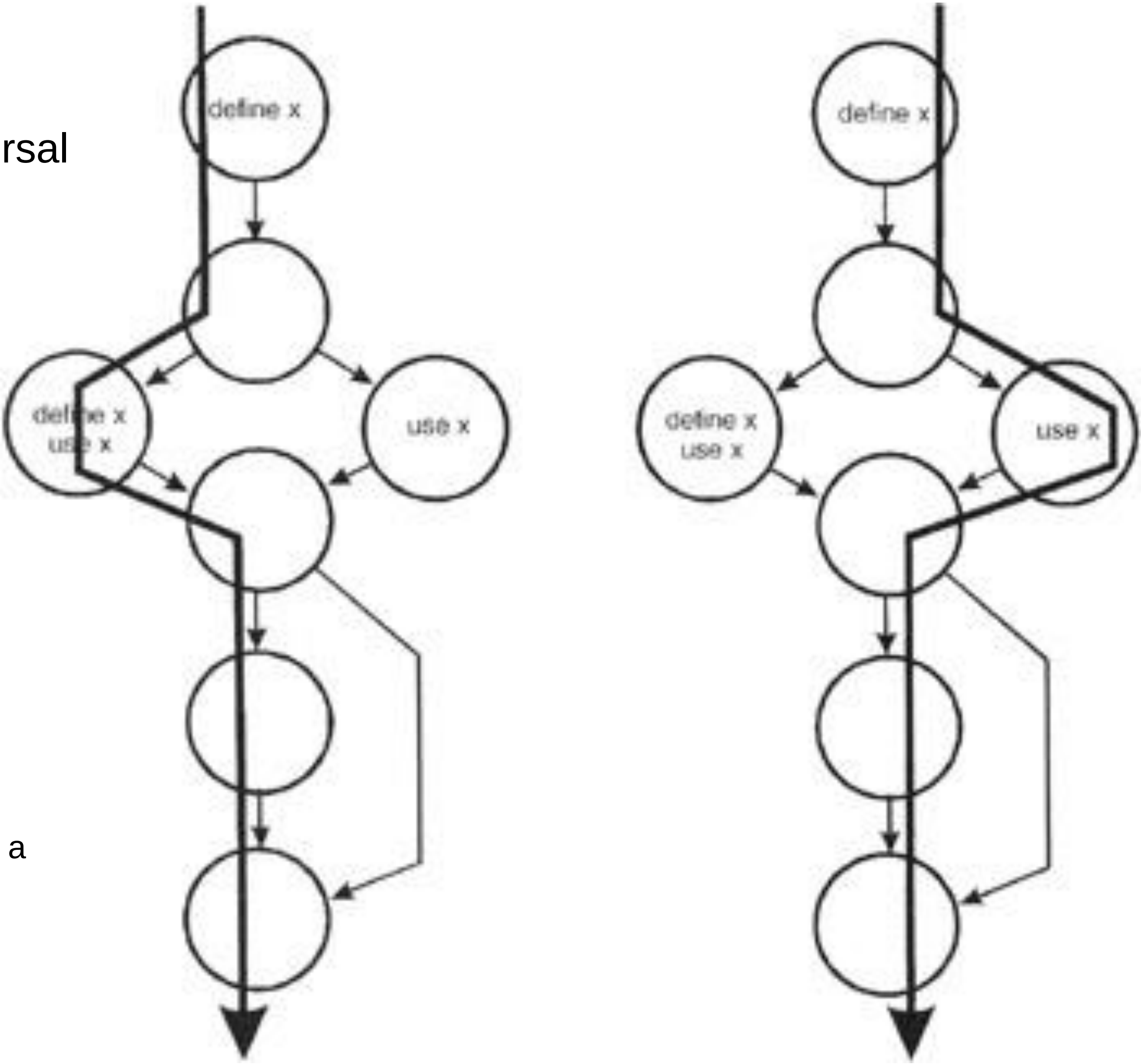
Static data flow testing

For each variable within the module examine define-use-kill patterns along the control flow paths



Example(2)

variable x and path traversal



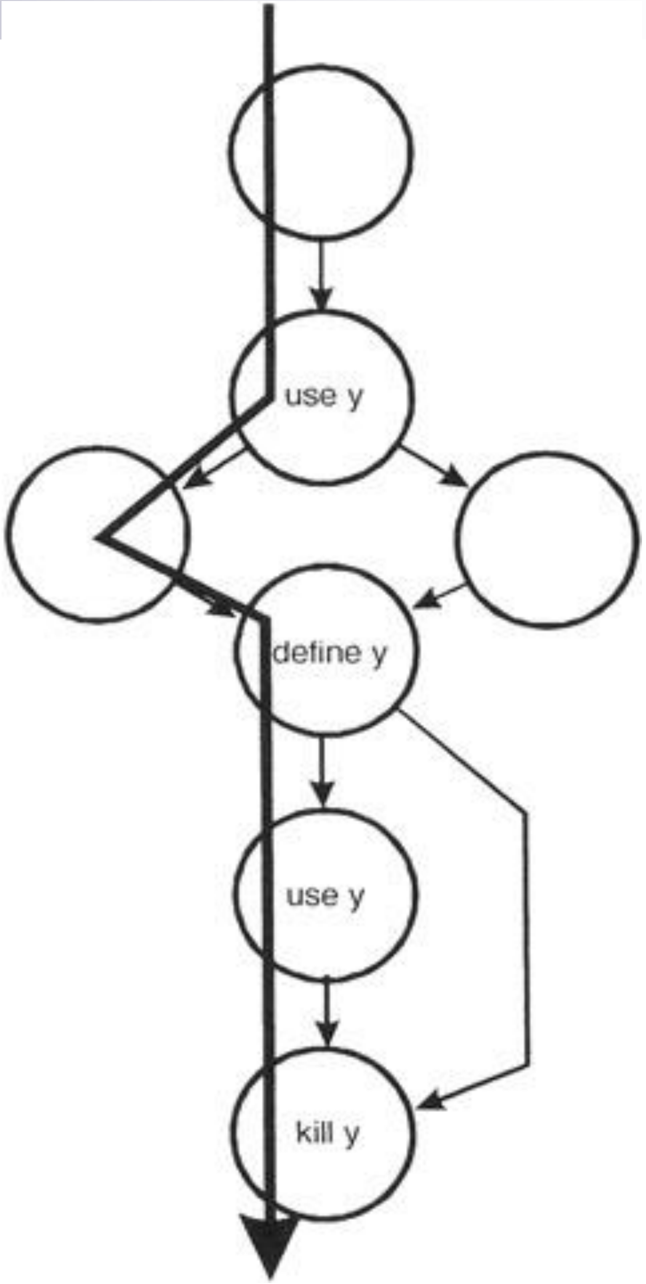
define correct, the normal case

define-define suspicious, perhaps a programming error

define-use correct, the normal case

Example(3)

variable y

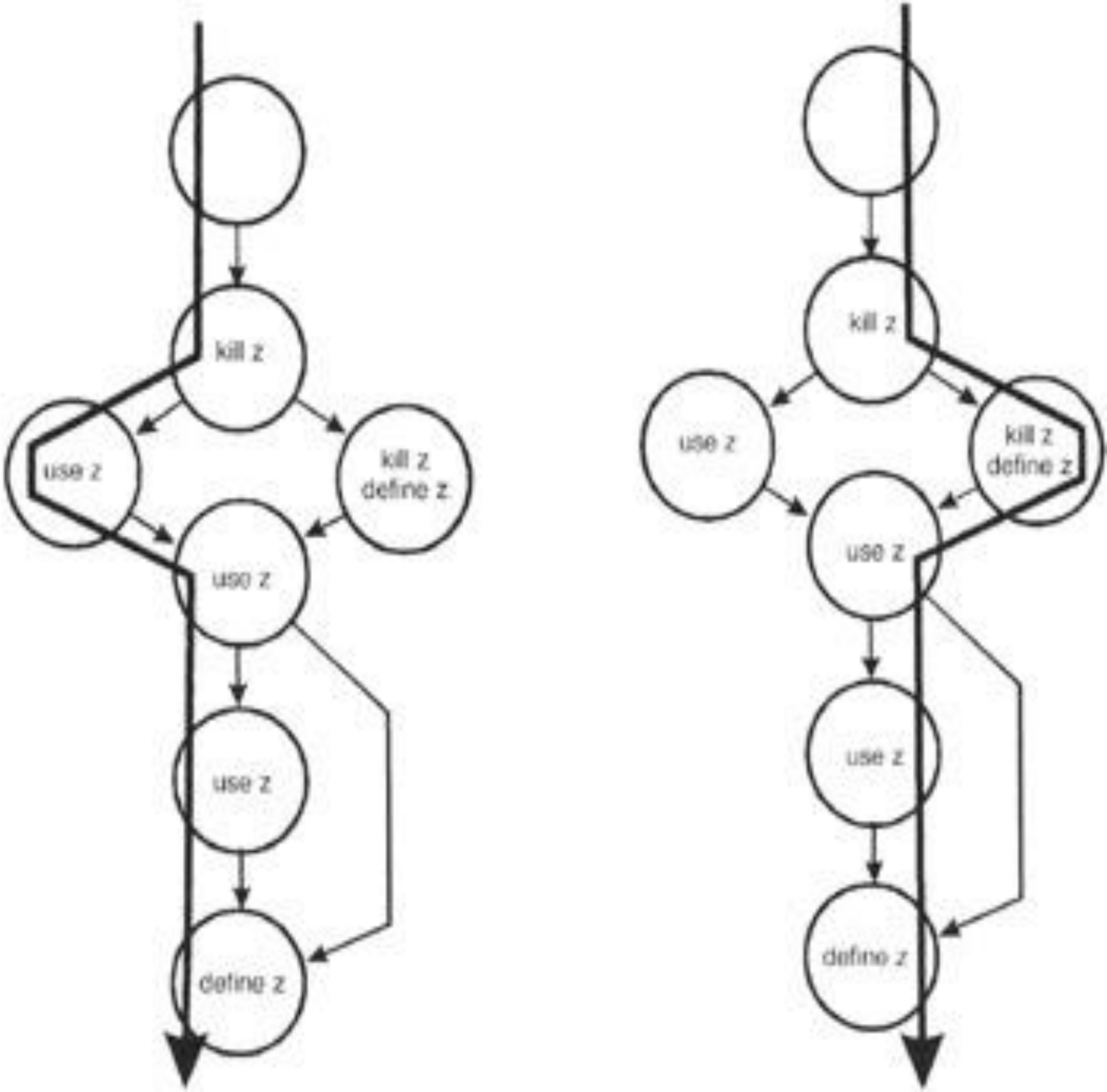


use	major blunder
use-define	acceptable
define-use	correct, the normal case
use-kill	acceptable

Example(4)

variable z

kill	programming error
kill-use	major blunder
use-use	correct, the normal case
use-define	acceptable
kill-kill	probably a programming error
kill-define	acceptable
define-use	correct, the normal case



Example(5)

Problems encountered:

x: define-define

y: use

y: define-kill

z: kill

z: kill-use

z: kill-kill

Static vs Dynamic Anomaly Detection

- **Static Analysis** is analysis done on source code without actually executing it.
  - *e.g.*, Syntax errors are caught by static analysis.

Static Data Flow Testing-limitations (1)

arrays -collections of data elements that share the same name and type

```
int test[100];            //defines an array named test
                         // consisting of 100 integer elements,
                         // named test[0], test[1], etc.
```

Arrays are defined and destroyed as a single entity but specific elements of the array are used *individually*.

Static analysis cannot determine whether the *define-use-kill* rules have been followed properly unless *each* element is considered individually

Static Data Flow Testing – limitations (2)

in complex control flows it is possible that certain path can never be executed.

an improper define-use-kill combination might exist but will never be executed and so is not truly improper.

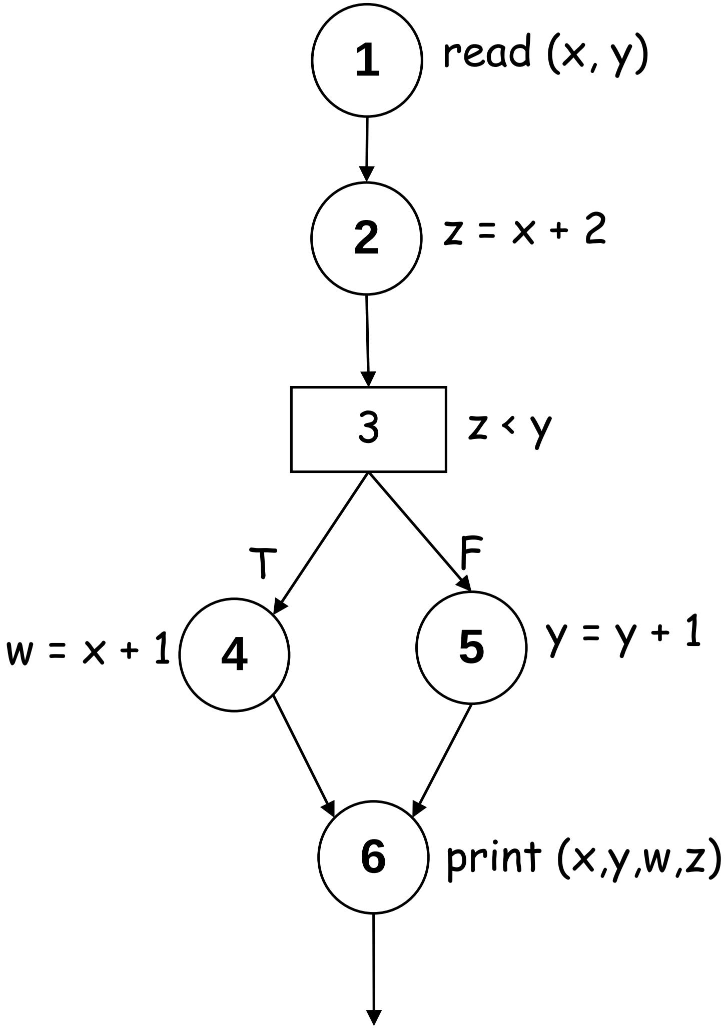
def-use Associations

A def-use association is a triple (x, d, u,), where:

x is a variable,
d is a node containing a definition of x,
u is either a statement or predicate node containing a use of x,

and there is a sub-path in the flow graph from d to u with *no* other definition of x between d and u.

Example



Some Def-Use Associations:

(x, 1, 2), (x, 1, 4), …

(y, 1, (3,t)), (y, 1, (3,f)), (y, 1, 5), …

(z, 2, (3,t)),...

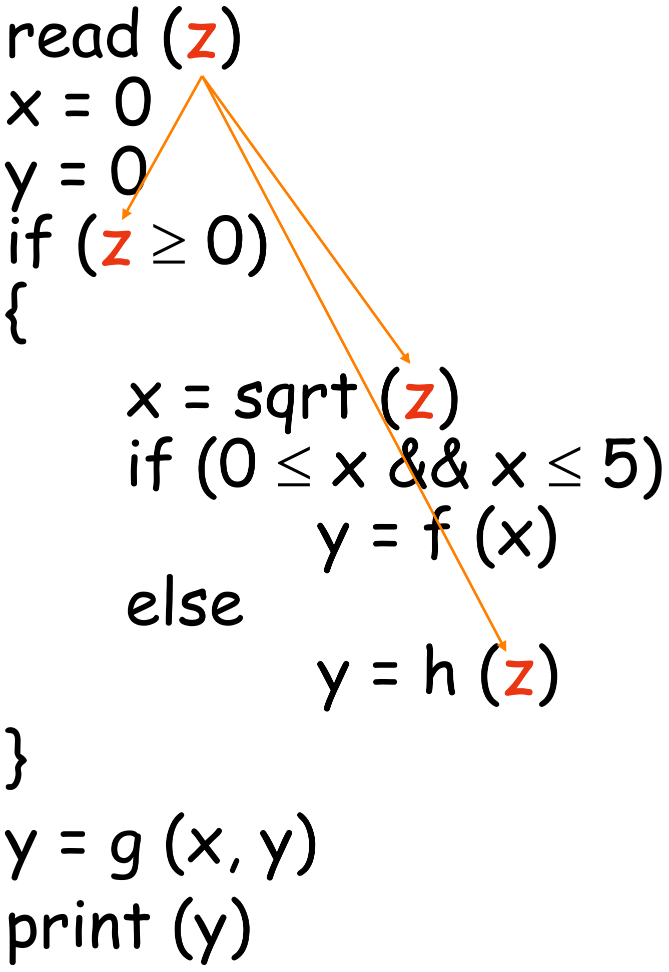
Example

def-use associations for the program below:

```
read (z)
x = 0
y = 0
if (z ≥ 0)
{
        x = sqrt (z)
        if (0 ≤ x && x ≤ 5)
                y = f (x)
        else
                y = h (z)
}
y = g (x, y)
print (y)
```

Example (1)

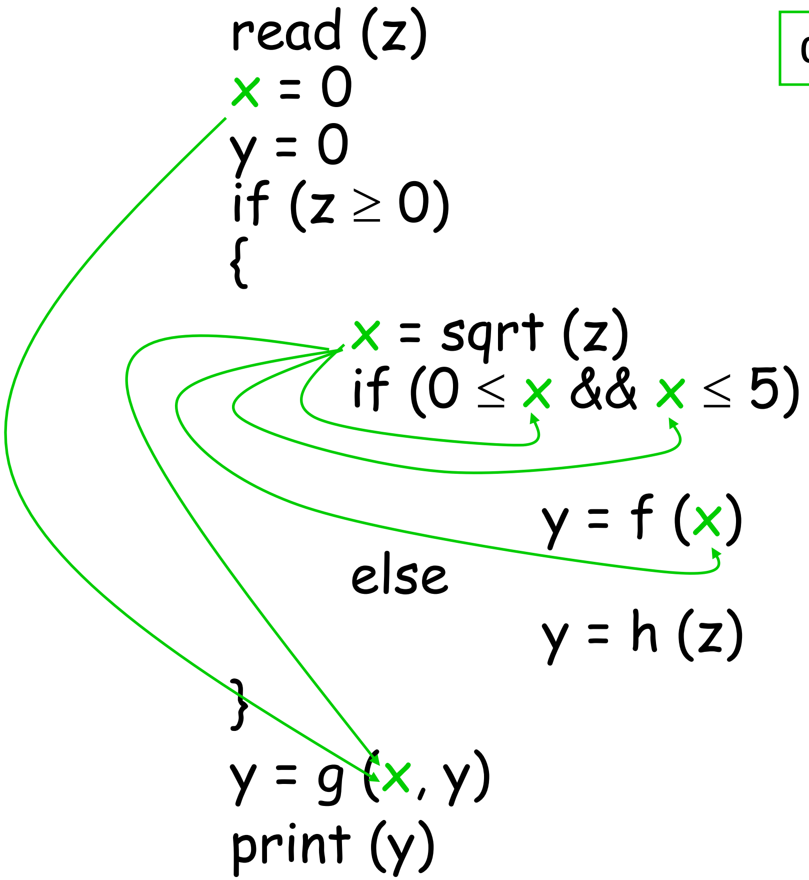
def-use associations for variable z



```
read (z)
x = 0
y = 0
if (z ≥ 0)
{
        x = sqrt (z)
        if (0 ≤ x && x ≤ 5)
                y = f (x)
        else
                y = h (z)
}
y = g (x, y)
print (y)
```

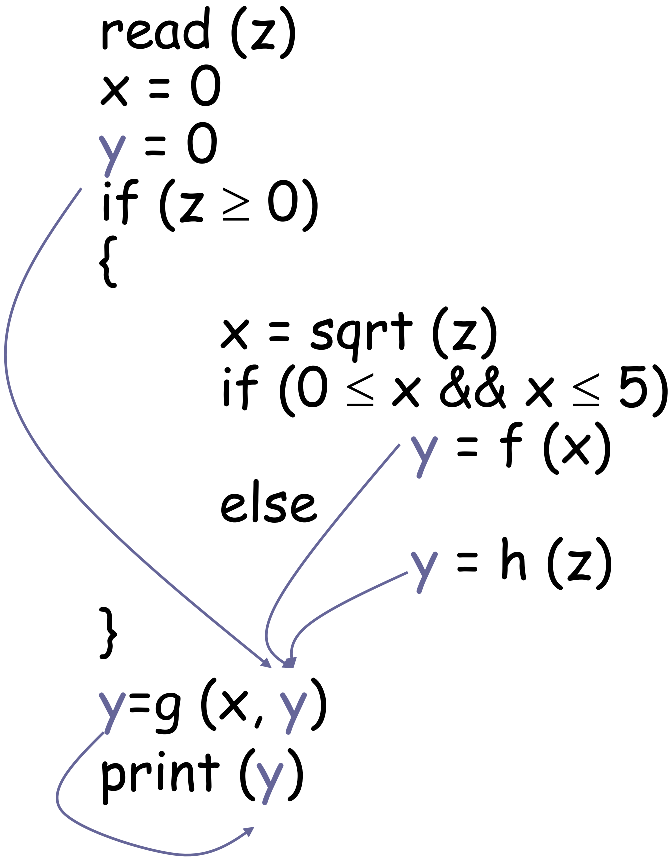
Example (2)

def-use associations for variable x



```
read (z)
x = 0
y = 0
if (z ≥ 0)
{
        x = sqrt (z)
        if (0 ≤ x && x ≤ 5)
                y = f (x)
        else
                y = h (z)
}
y = g (x, y)
print (y)
```

Example (3)



```
read (z)
x = 0
y = 0
if (z ≥ 0)
{
        x = sqrt (z)
        if (0 ≤ x && x ≤ 5)
                y = f (x)
        else
                y = h (z)
}
y=g (x, y)
print (y)
```

def-use associations for variable y

Definition-Clear Paths

A path (i, n_1, ..., n_m, j) is called a *definition-clear path* with respect to x from node i to node j if it contains no definitions of variable x in nodes (n_1, ..., n_m , j) .

Data flow testing coverage

p-use paths

c-use paths

all-use paths

p-use path

A path starts from a definition of a variable and ends in a statement
In which it appears inside a certain predicate

c-use path

A path starts from a definition of a variable and ends in a statement
In which it is involved in computing

all-use path

A path starts from a definition of a variable and ends in a statement
In which it becomes used

Data flow testing coverage (1)

All predicate-uses/ some computational-uses testing

for every variable and every definition of that variable, a test includes at least one def-clear path from the definition to every predicate use;

if there are definitions not covered by that description, then include computational uses so that every definition is covered.

Data flow testing coverage (2)

All computational-uses/ some predicate-uses testing

for every variable and every definition of that variable, a test includes at least one def-clear path from the definition to every computational use;

if there are definitions not covered by that description, then include predicate uses so that every definition is covered.

Data flow testing coverage (3)

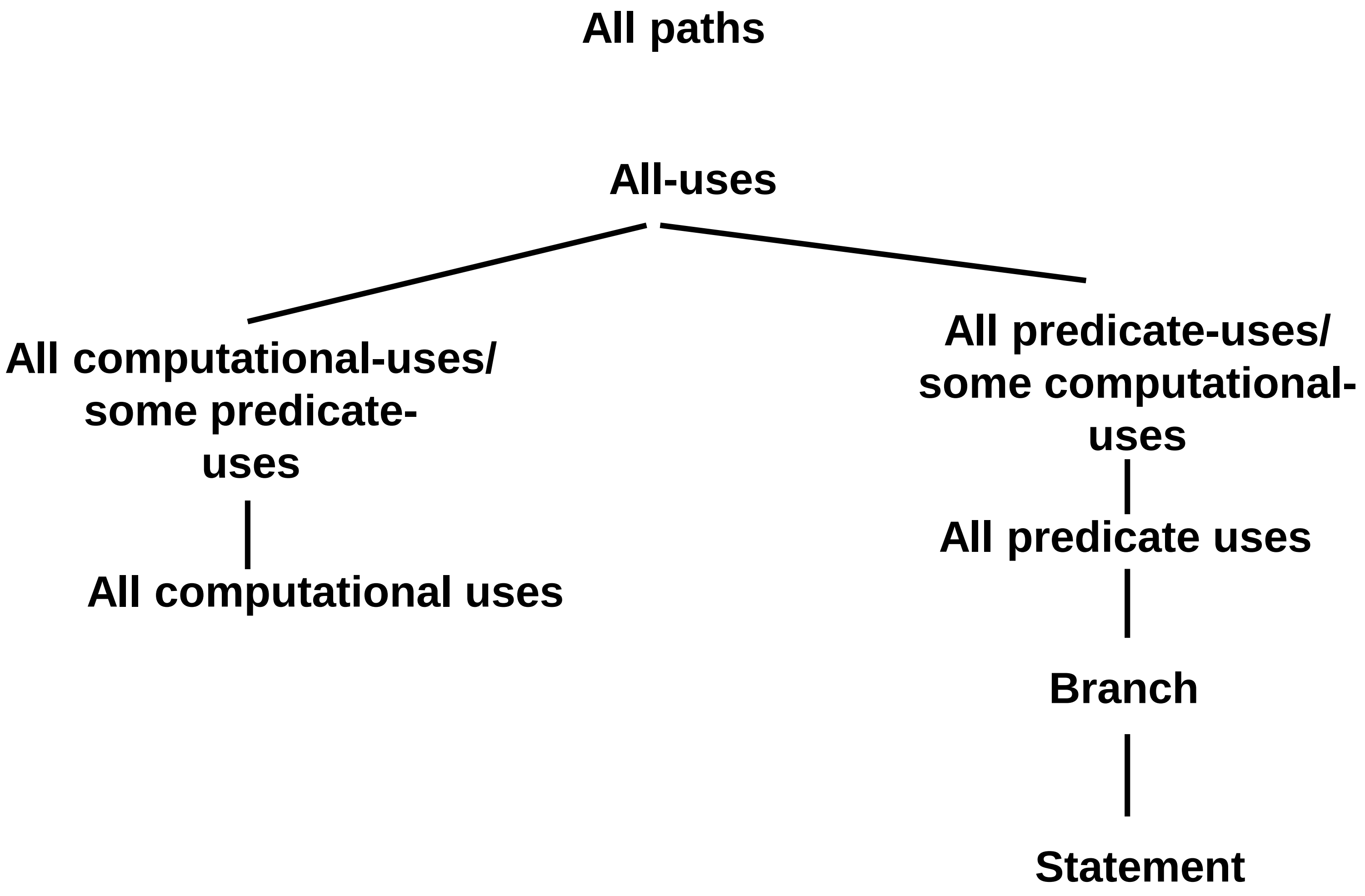
All-uses testing

The test includes at least one def-clear path from every definition to every use that can be reached by that definition

Summary

- Data are as important as code.
- Define what you consider to be a data-flow anomaly.
- Data-flow testing strategies span the gap between **all paths** and **branch** testing.

Test strategies: ordering



All paths

All-uses

All computational-uses/ some predicate-uses

All computational uses

All predicate-uses/ some computational-uses

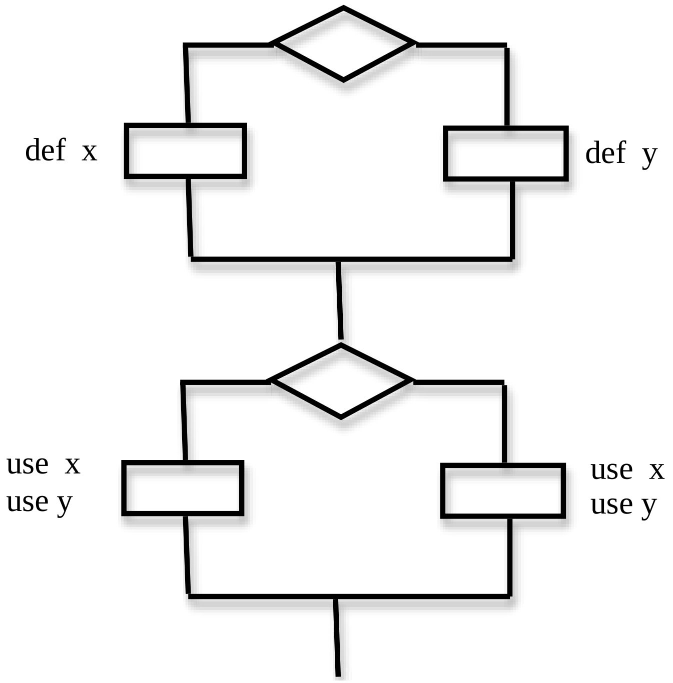
All predicate uses

Branch

Statement

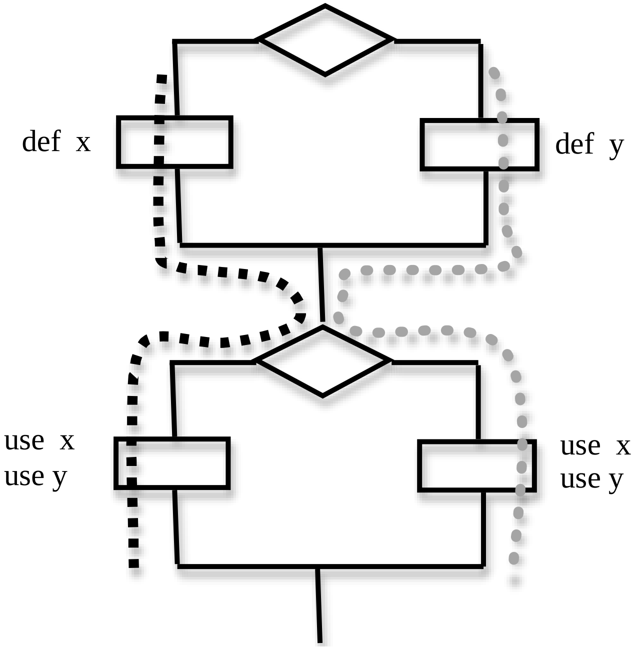
Test strategies: ordering

All-uses *subsumes* branch coverage



Test strategies: ordering(3)

branch coverage



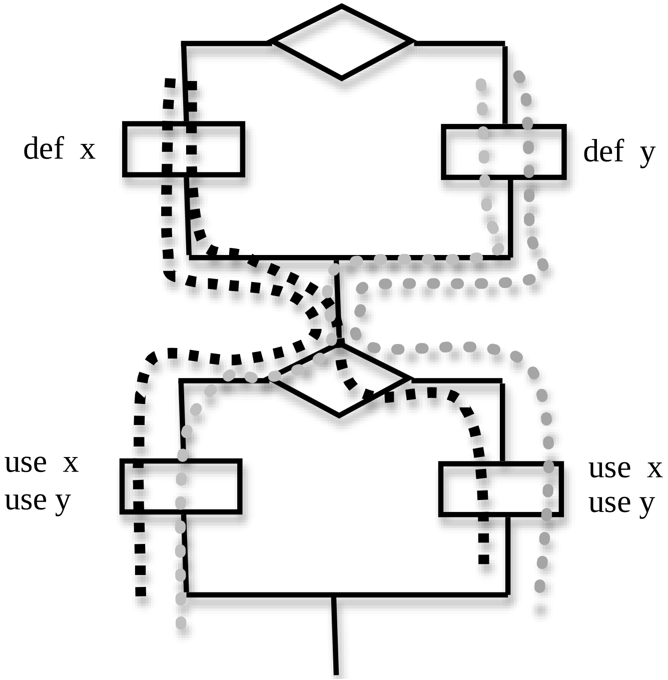
def x

def y

use x
use y

use x
use y

All uses coverage



def x

def y

use x
use y

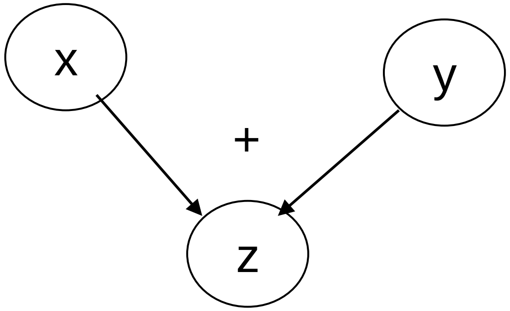
use x
use y

Data Dependency Graphs (DDGs) and Data Slices

Data dependency graph (DDG)

Node: definition of a data item (variables, constants, compound structures)

Link: represents some D-U relation



Assignment z←x+y

Data dependency graph (DDG): categories of nodes

Definition of x , D(x)

- **Output or result nodes** – computational results for the program under testing

- **Input or constant nodes** (user-provided inputs or predefined constants)

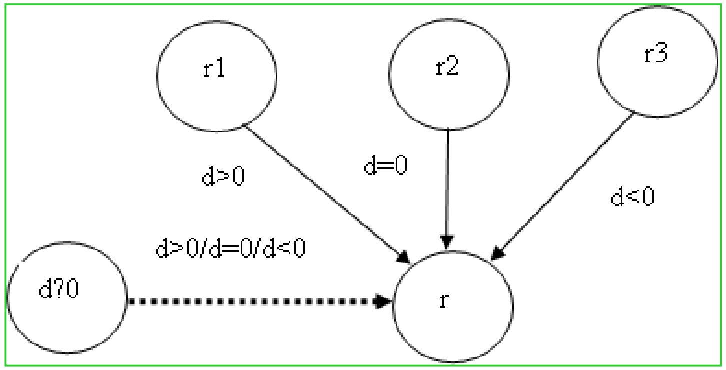
- **Intermediate or storage nodes** (to facilitate computational procedures)

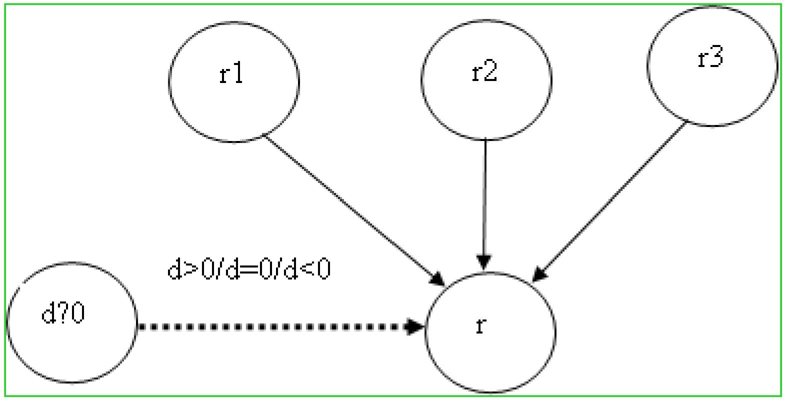
Data dependency graph (DDG): categories of links

Data inlinks

Control inlink

equivalent notation





Real roots of the quadratic equation (discriminant d)

Main features of data dependency graphs (DDGs)

Usually one output data item or a few of them

Typically more input variables

Multiple inlinks are common

“fan” shaped DDGs (flow from top to bottom)

DDG – design process

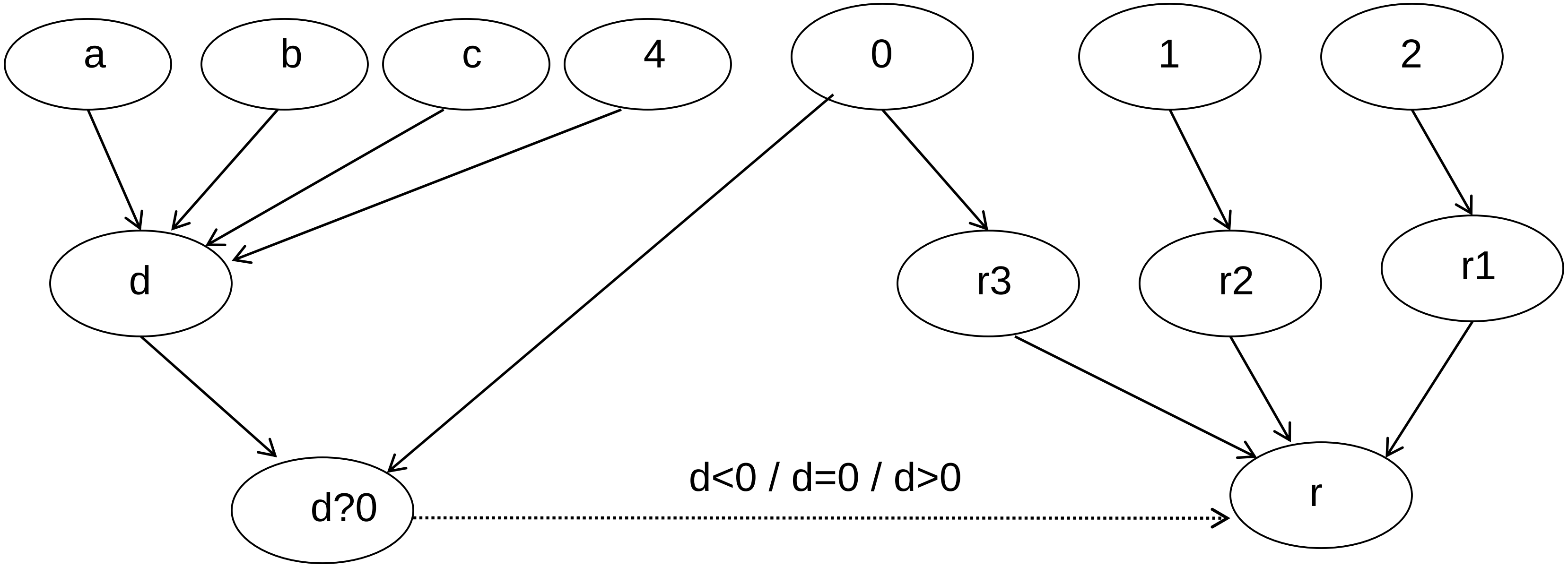
Backward stepwise data resolution from output to input

Identify output variables

Backward chaining to resolve variables using other variables and constants by consulting the specific computation involved

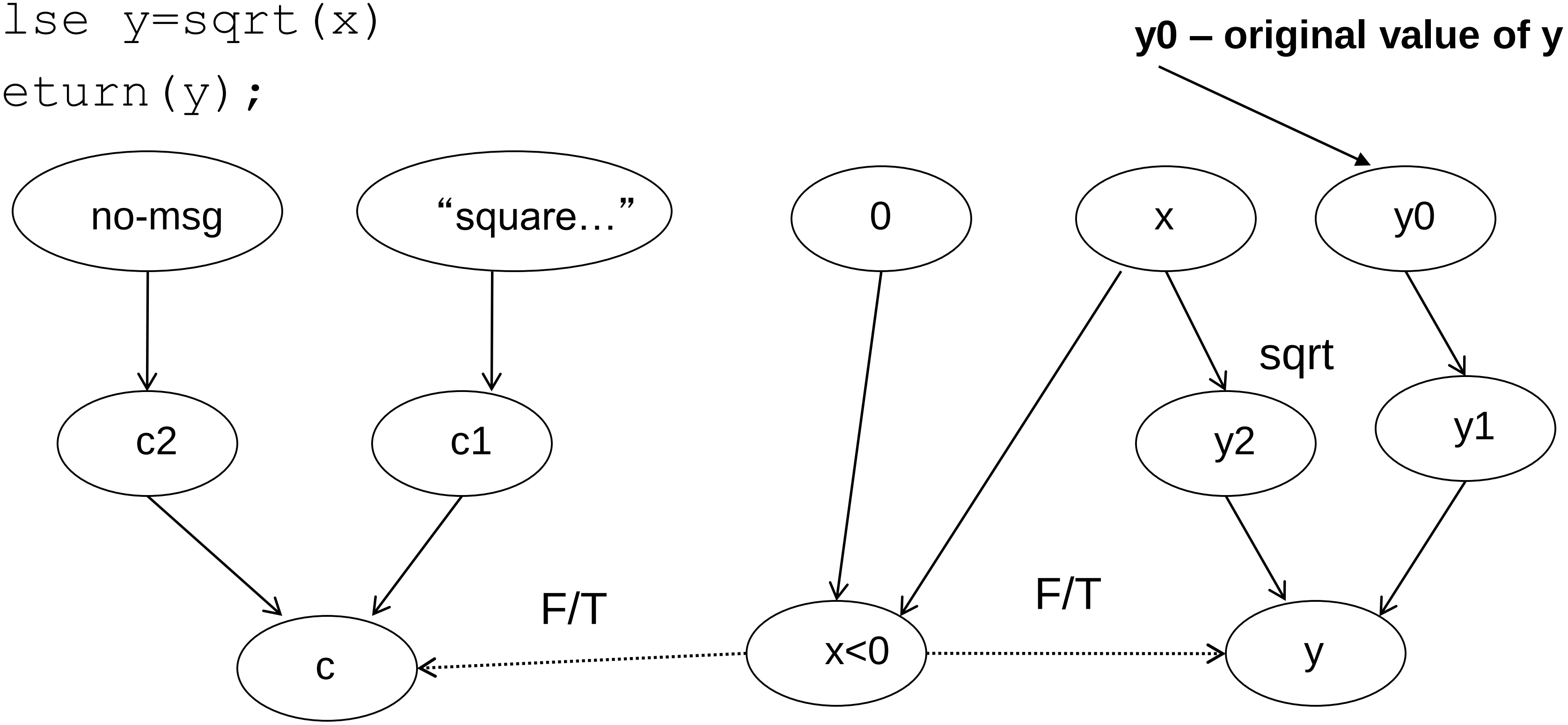
If unresolved variables, repeat the step above until no unresolved variables left

Example (1)



Example (2)

```
Input(x);
If (x<0) then exit ("square root undefined for
   negative number")
else y=sqrt(x)
return(y);
```



Data Flow Testing: data slices

Data flow testing by selecting data slices

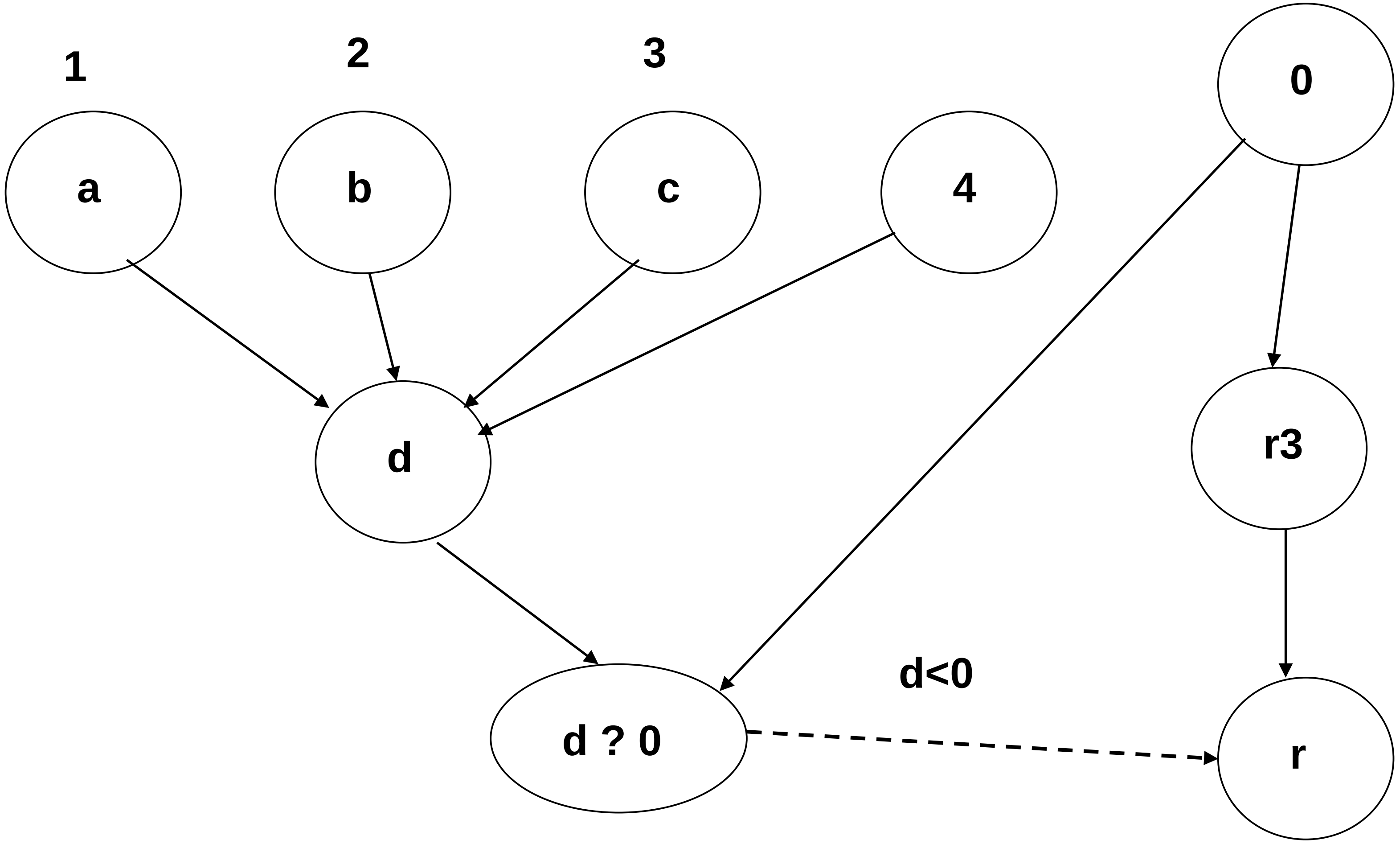
Data slice: a subset of a program; specific realization of a value of the output variable through a specific set of input variables and constant values

Slicing – a way of filtering irrelevant code

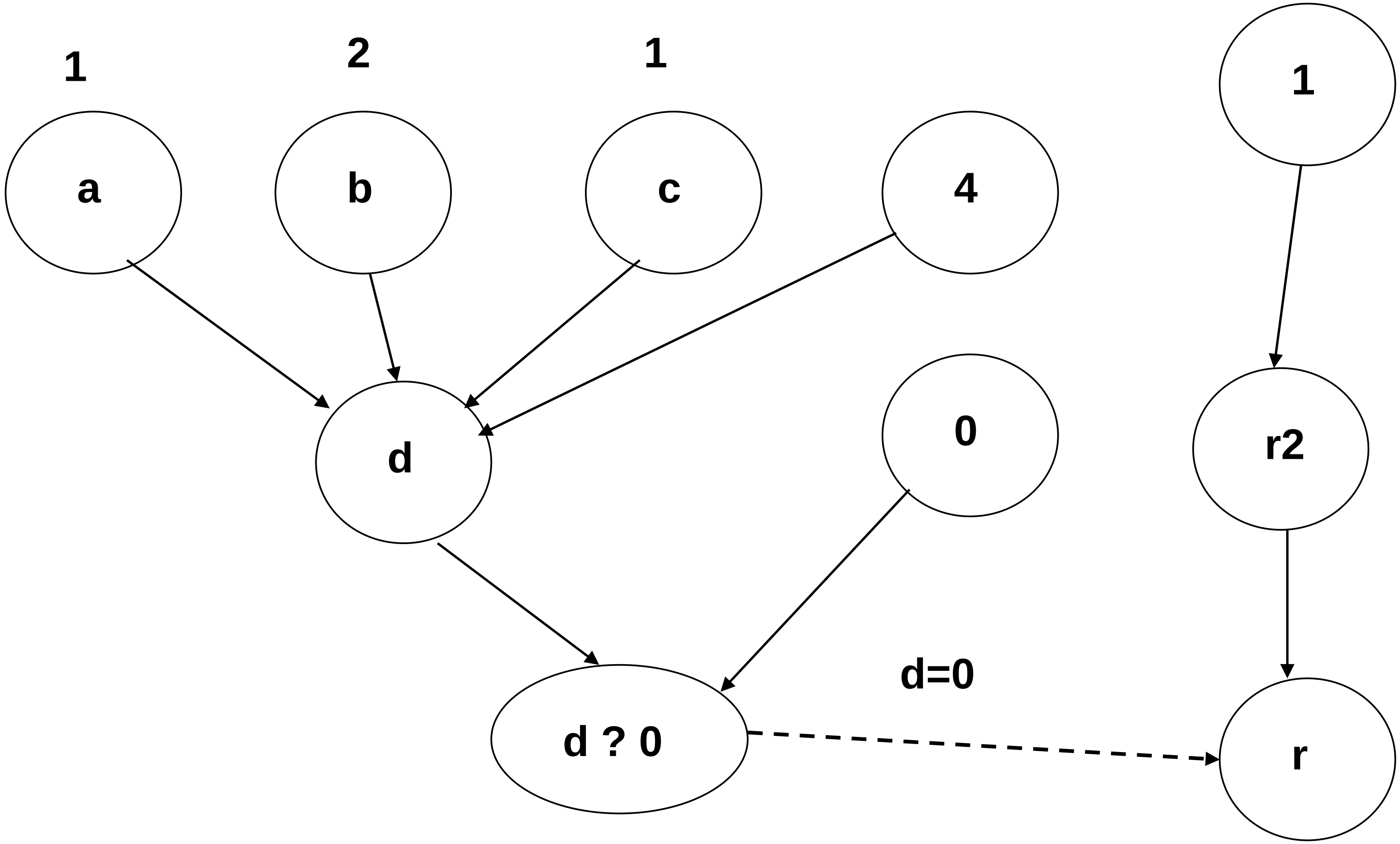
single output variable, no selector nodes – all used input variables and constants are covered in one slice

if there are selectors, multiple slices

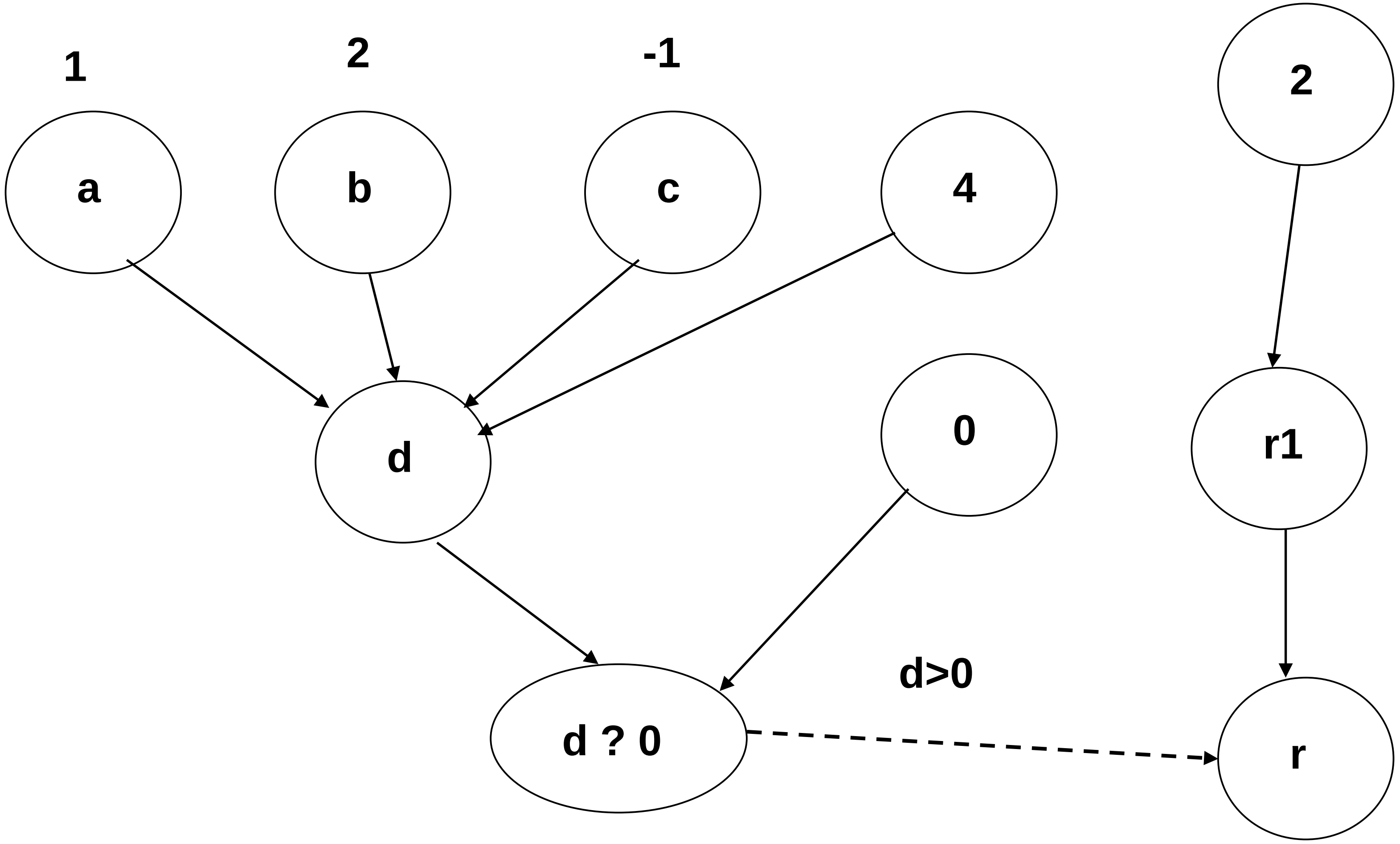
Data slices – sensitization



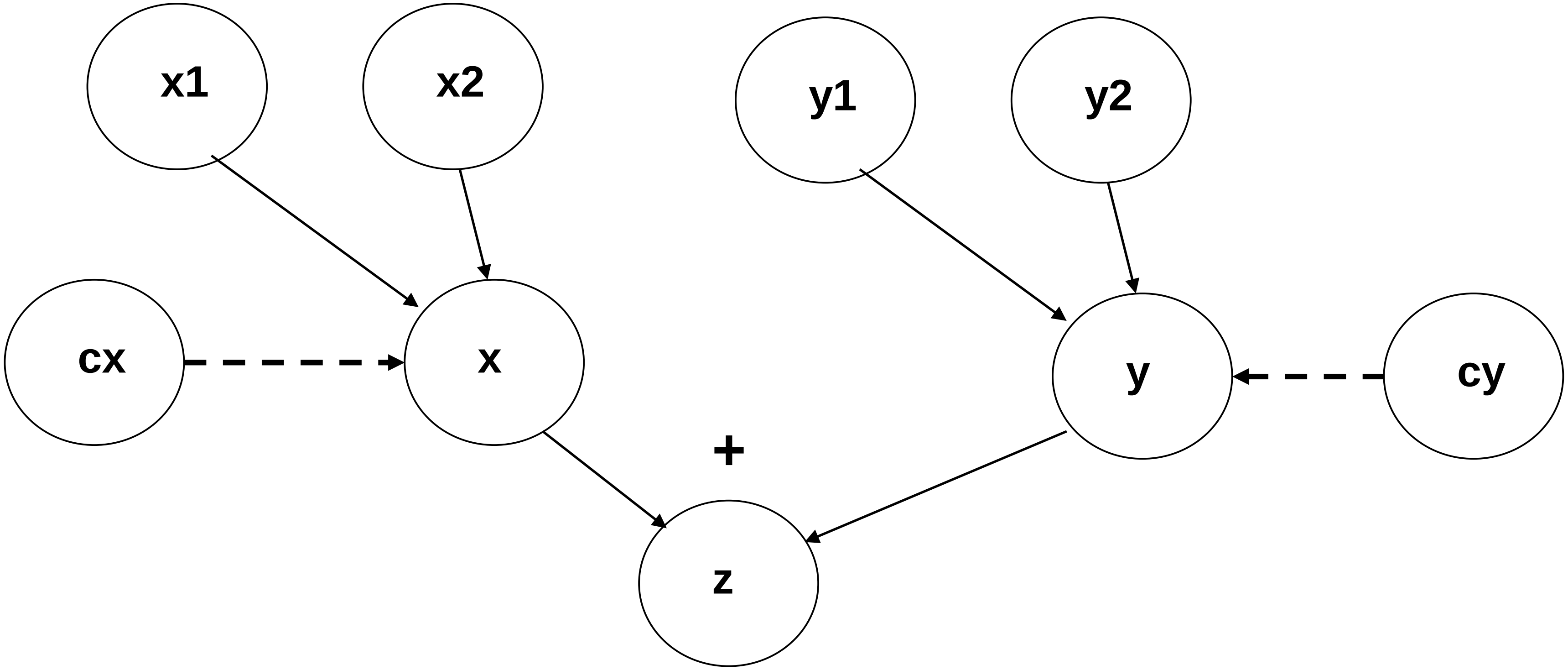
Data slices – sensitization



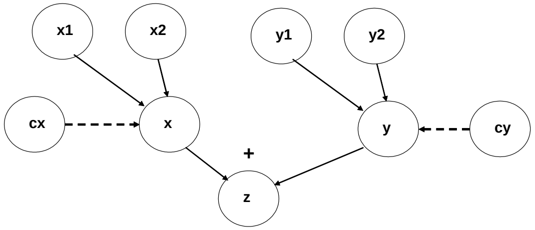
Data slices – sensitization

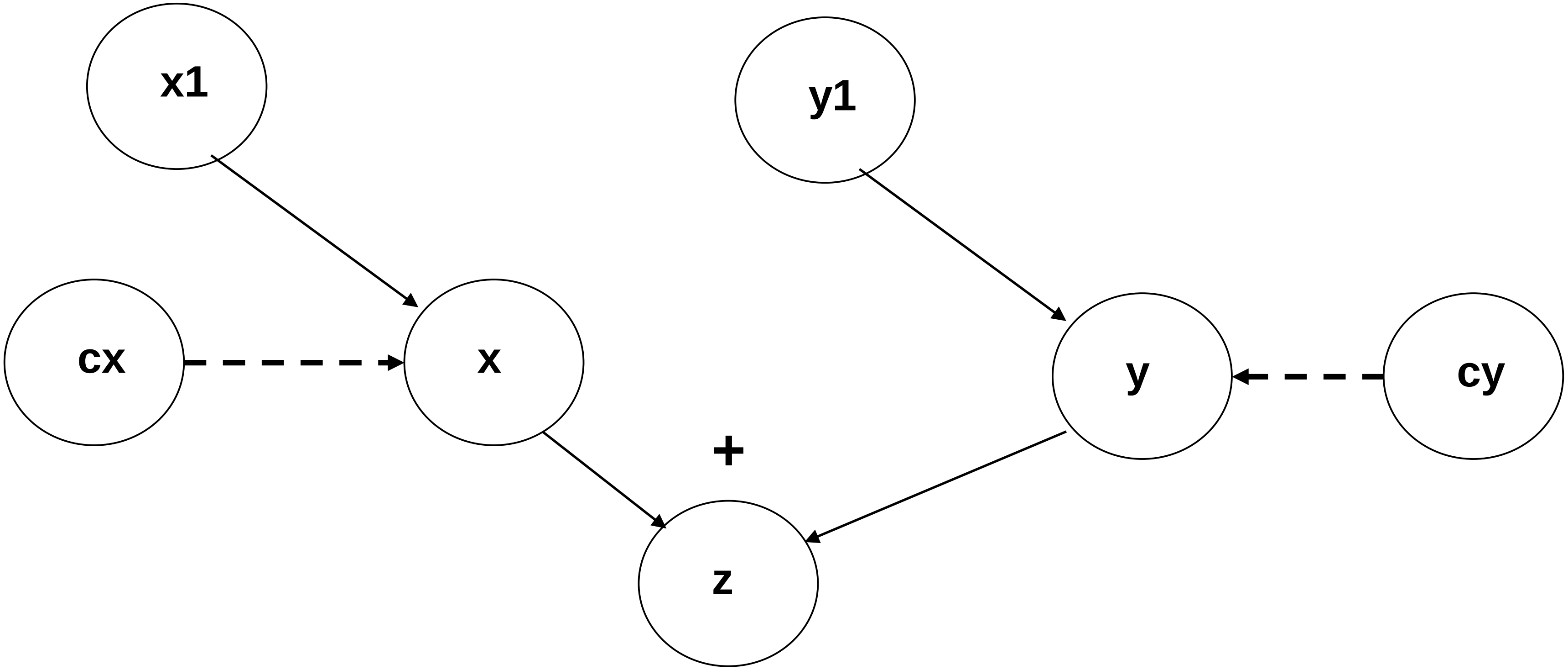


Data slices – example(2)

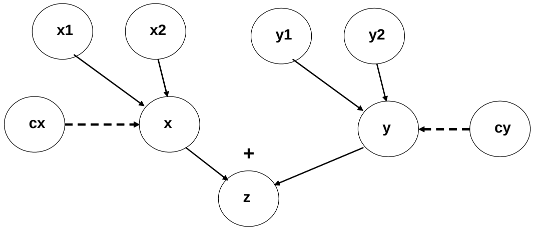


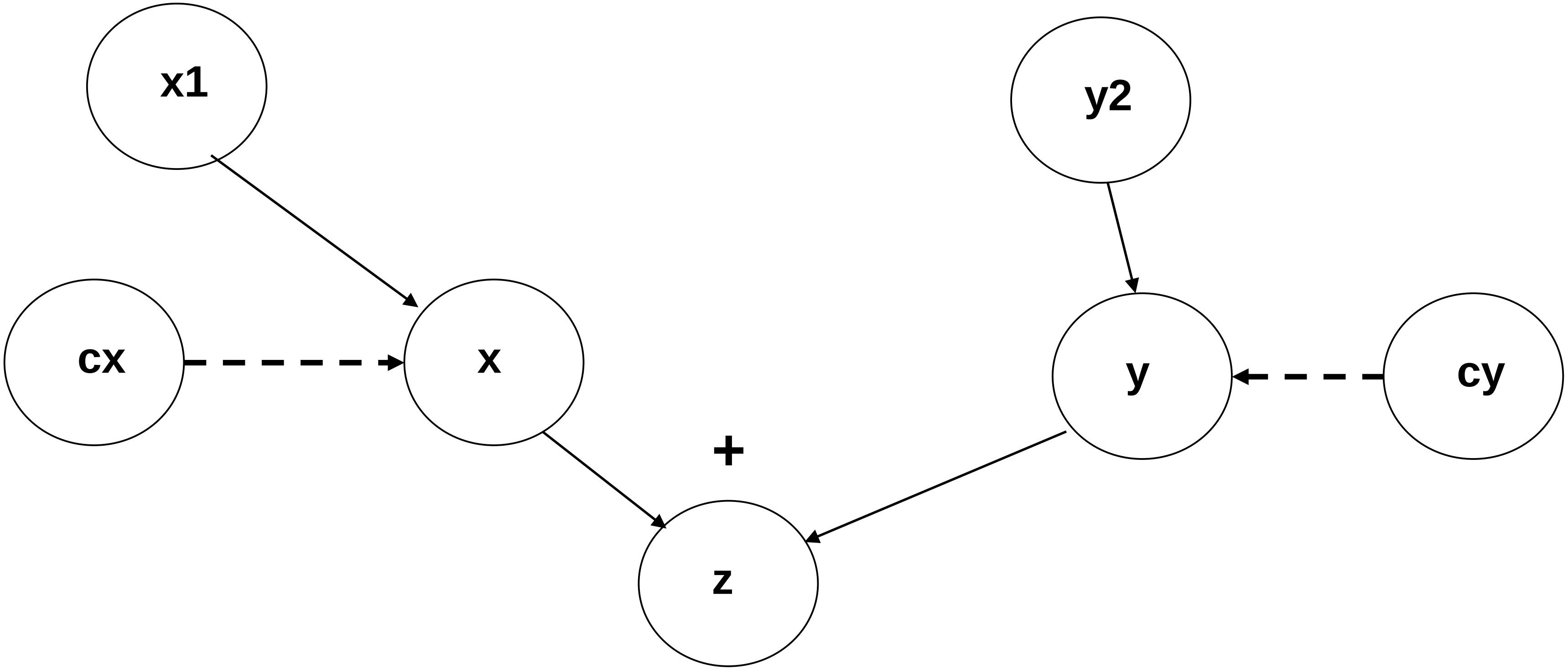
Data slices – example(2)



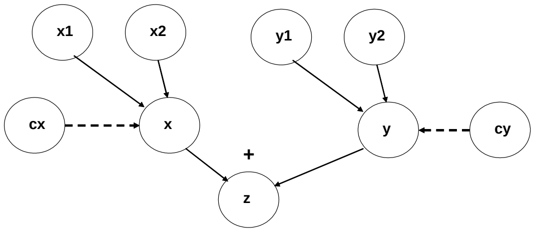


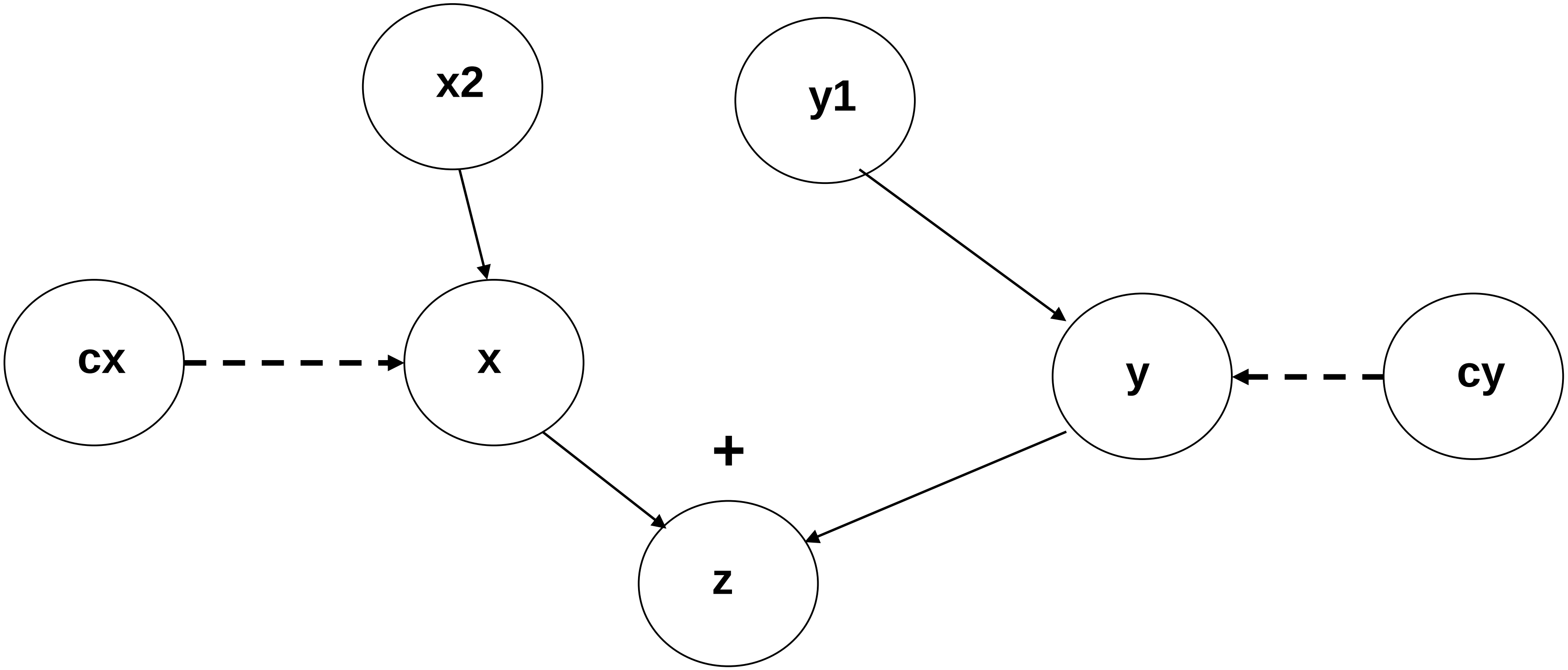
Data slices – example(2)



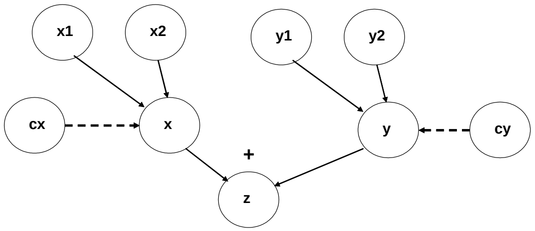


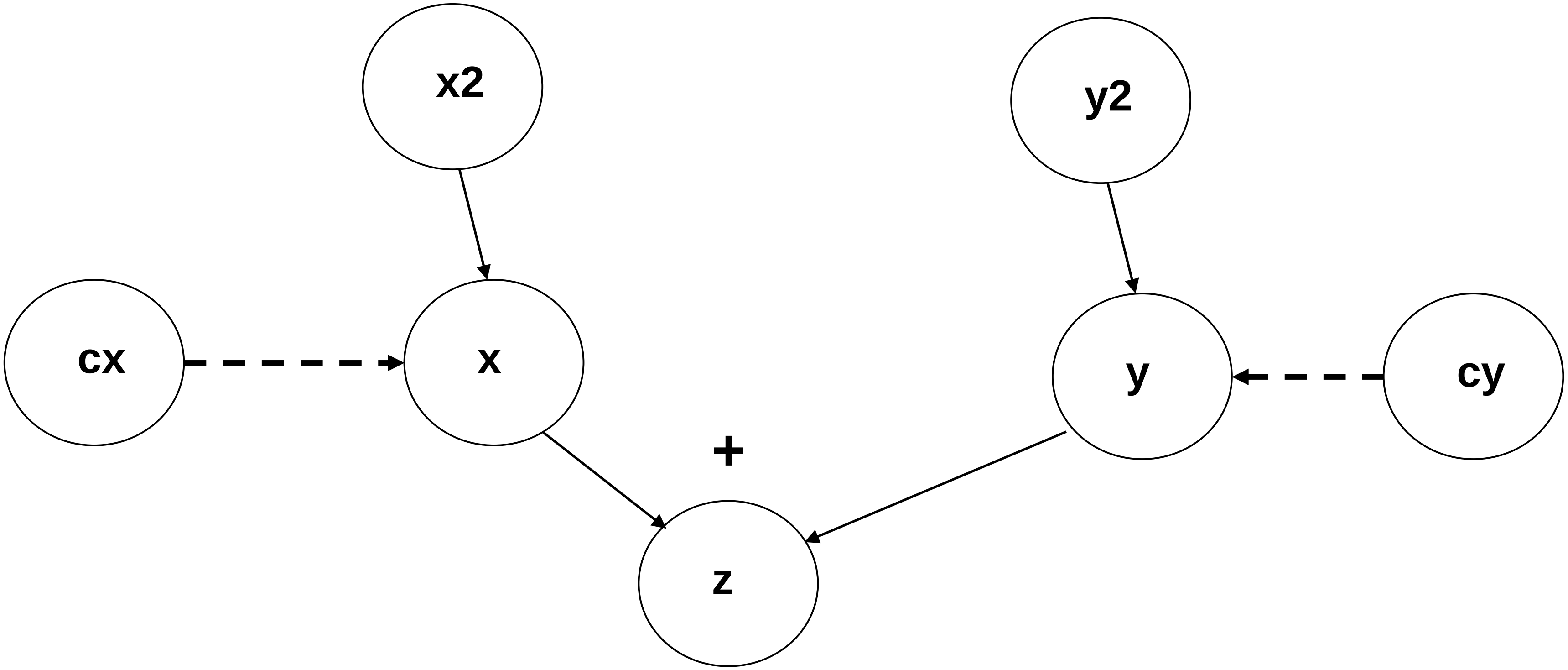
Data slices – example(2)



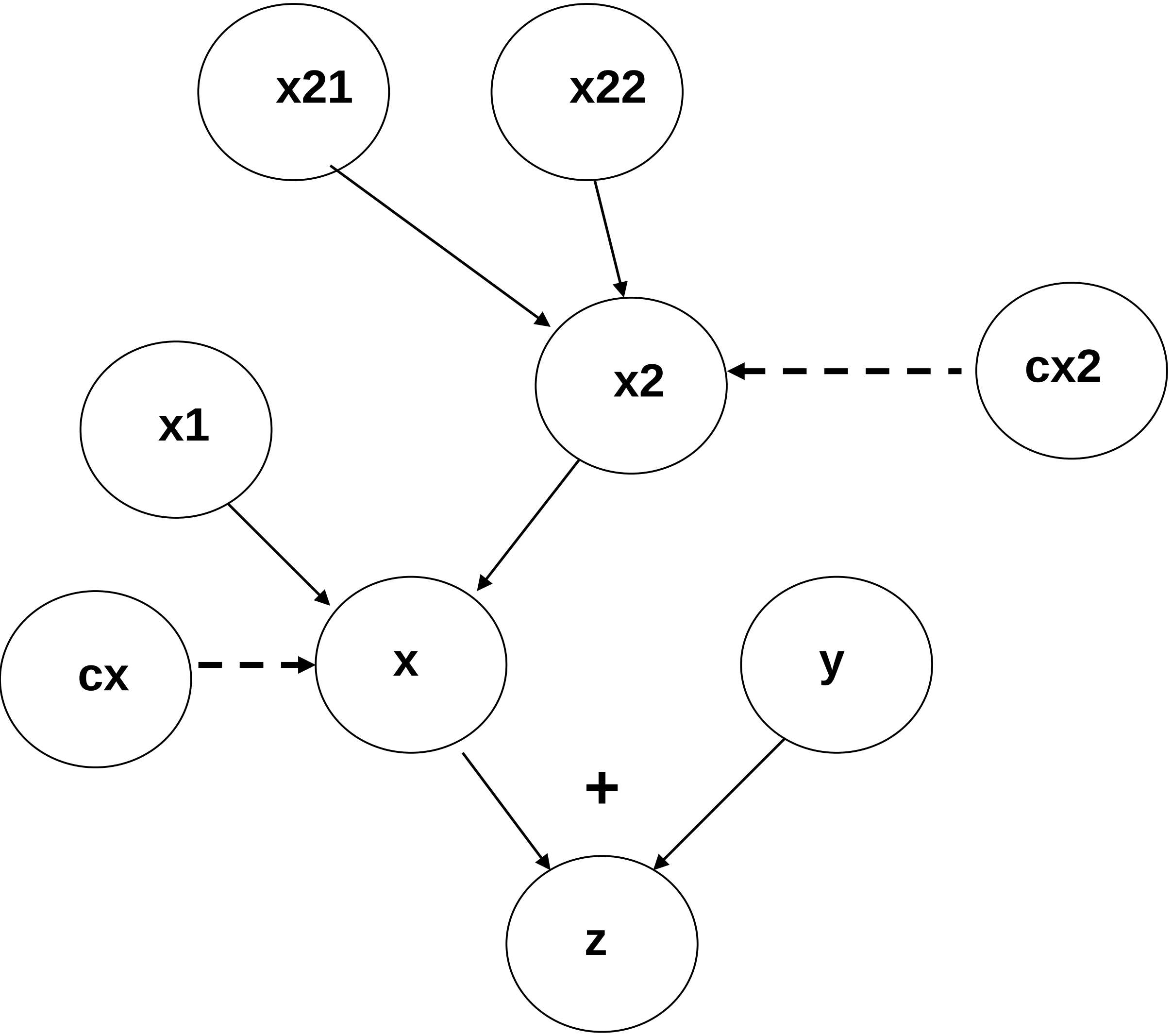


Data slices – example(2)



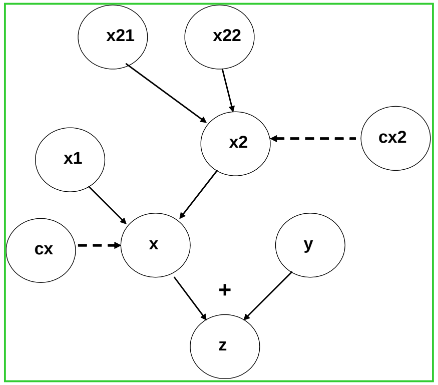


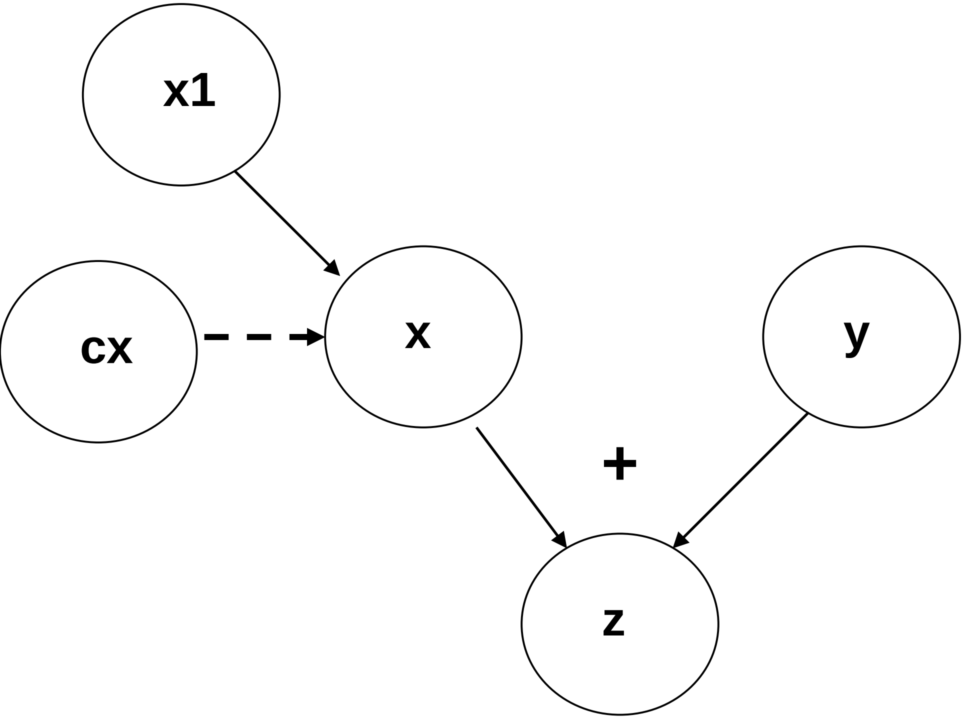
Data slices – example(3)



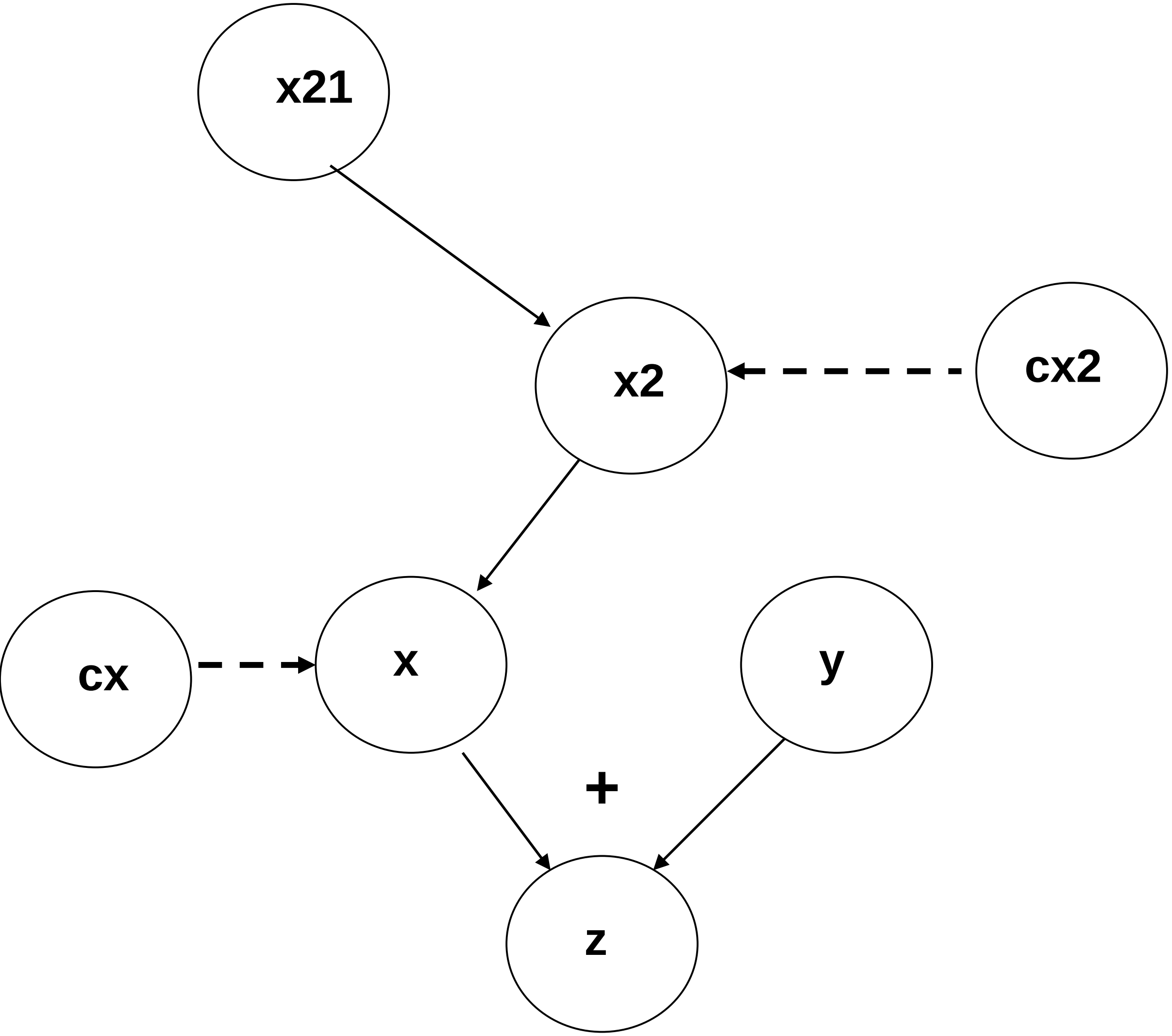
Original data flow graph DDG

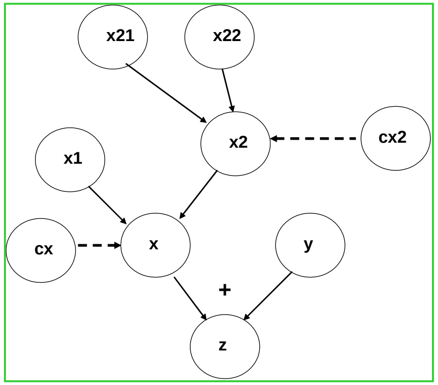
Data slices – example(3)



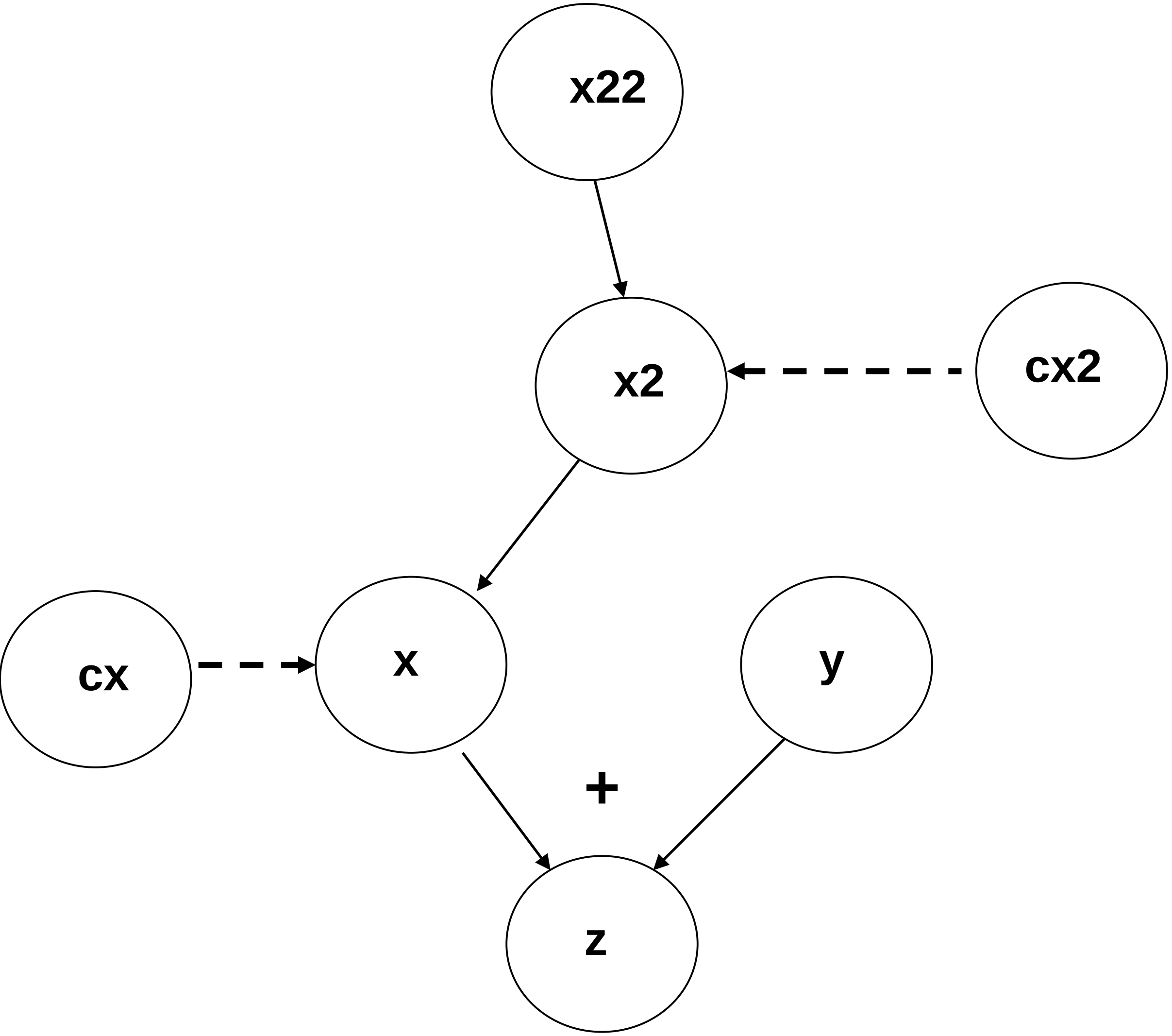


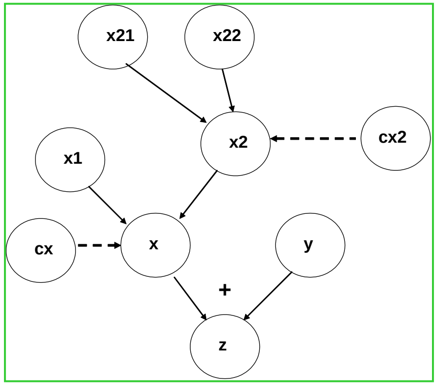
Data slices – example(3)





Data slices – example(3)





Sensitization of data slice

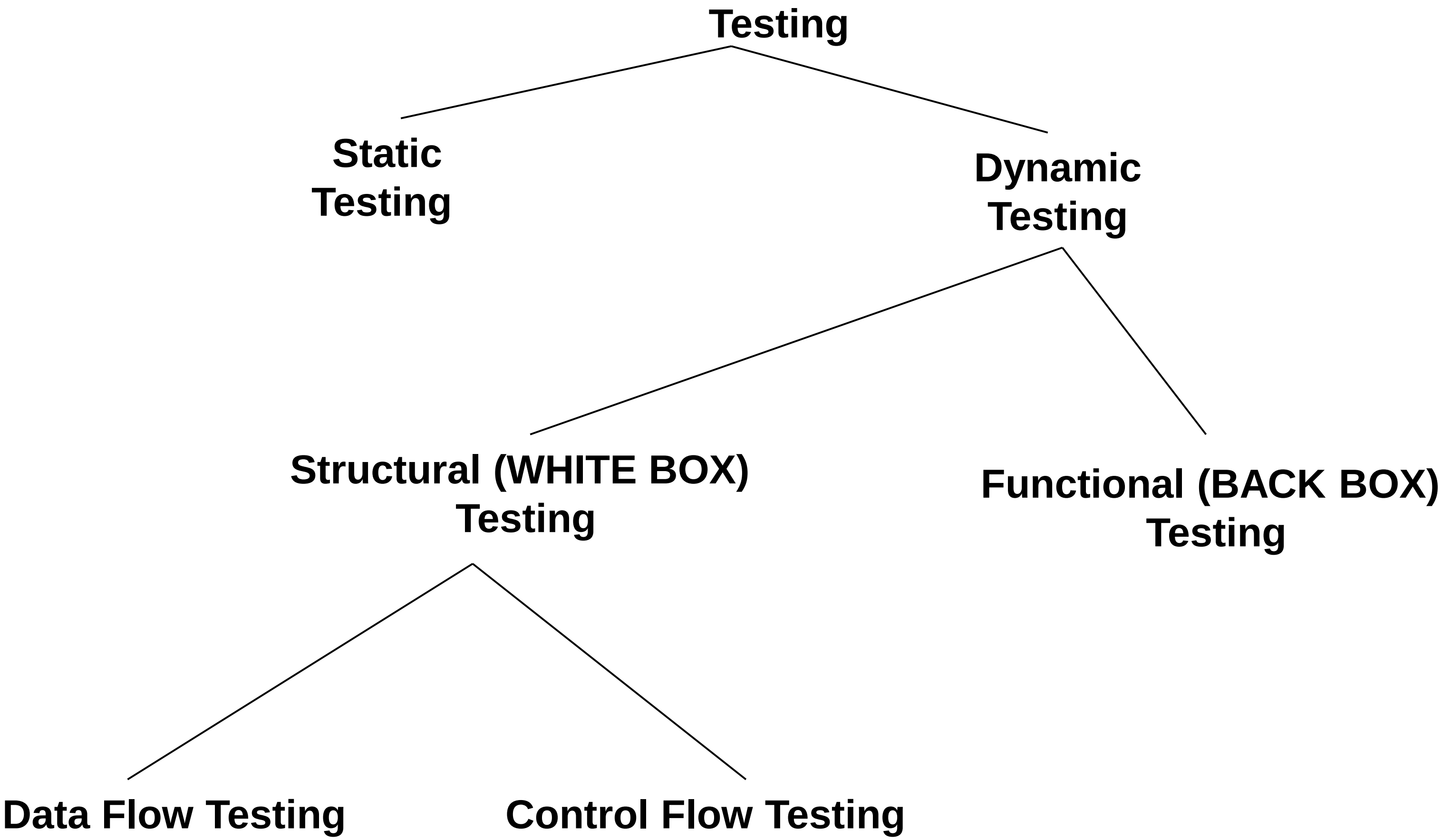
every input variable and constant involved in the slice needs to initialized with specific values

(a) involved in the predicate – ensure the realization of the desired predicate

(b) involved as a data input, potentially any value is allowed

Black Box and White Box: a comparative view

Software testing: an overview



Testing

Static Testing

Dynamic Testing

Structural (WHITE BOX) Testing

Functional (BACK BOX) Testing

Data Flow Testing

Control Flow Testing

Functional vs. structural testing

Functional testing*: focus on the external behaviour*

Structural testing: *internal implementation*

Level of abstraction

High **→ functional testing more likely (large software or its substantial parts)**

Low **→ structural testing (small objects)**

Functional vs. structural testing

Timeline

White Box → early sub-phases of component testing; need code

Black Box → early (before any code completed) and in late sub-phases (system and acceptance testing)

Functional vs. structural testing

Defect focus

Black Box **→ external functions, reduction of functional problems by customers**

White Box **→ internal implementations; faults detected and removed**

Functional vs. structural testing

Defect detection and fixing

White Box → easier to fix; WBT could miss types of defects (omission + design problems)

Black Box → more difficult; problems with interfaces and interactions

Functional vs. structural testing

Tester

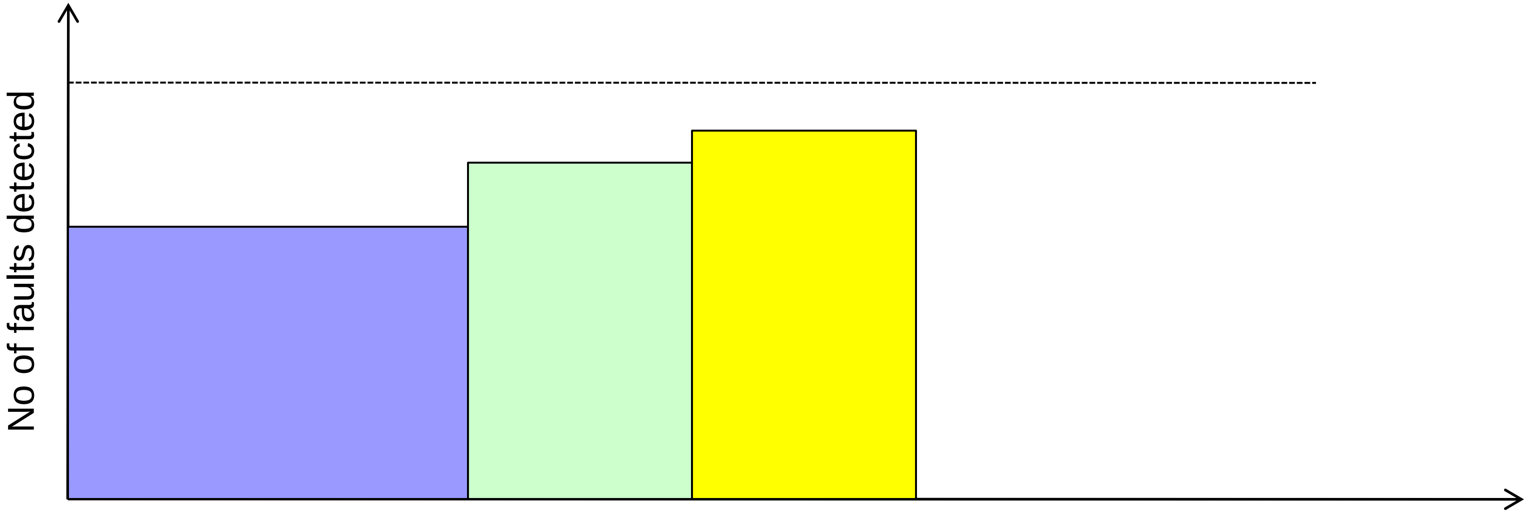
White Box → **include software developers**

Black Box → **professional tester; independent verification and validation**

Effectiveness of testing: a comparative view

The set of programs, effectiveness expressed in terms:

- number of test cases produced
- % of known faults detected



	Random testing	control-flow	data flow
	79.5%	85.5%	90%
# of test cases	100	34 (branch cov)	84 (all use coverage)

reference: Ntafos, 7 math programs

IEEE Trans on Software Eng.