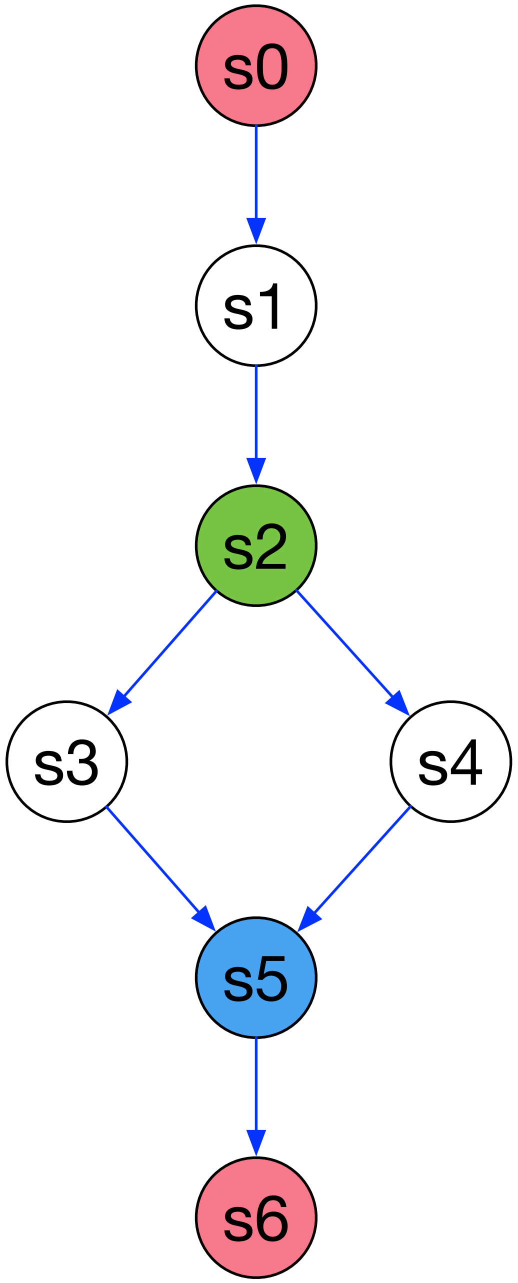
Code Coverage

Venkatesh-Prasad Ranganath
Kansas State University

A Program & its CFG

```
s0: z = input()
s1: x = input()
s2: if x > 5:
s3:     y = x * 5
    else:
s4:     y = z / 5
s5: print(y)
s6: return
```



Each node is a statement

Each solid edge is control flow edge between two statements

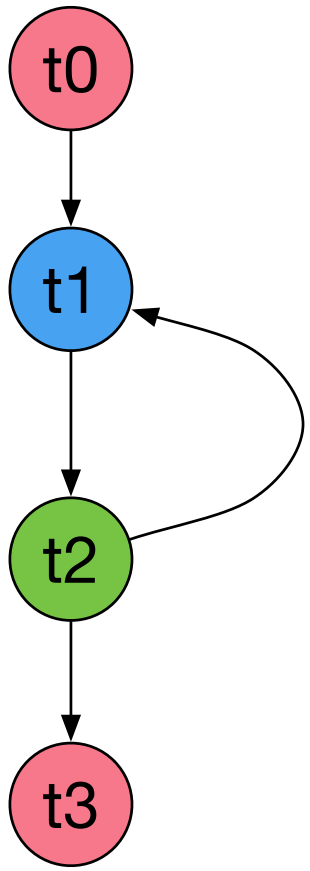
CFG is short for Control Flow Graph

Node and Edge Coverage*

- **Node (Statement) Coverage**
  - Fraction of graph nodes covered by tests
  - *Testing Goal:* Every node should be executed at least once
- **(Control Flow) Edge Coverage**
  - Fraction of graph edges covered by tests
  - *Testing Goal:* Every edge should be executed at least once

(Control Flow) Path Coverage*

- **Path** is a sequence of nodes in a graph such that consecutive nodes in the path are connected by a edge in the graph
- **(Control Flow) Path Coverage**
  - Fraction of graph paths covered by tests
  - How can we deal with programs with loops, i.e. graphs with infinite number of paths?

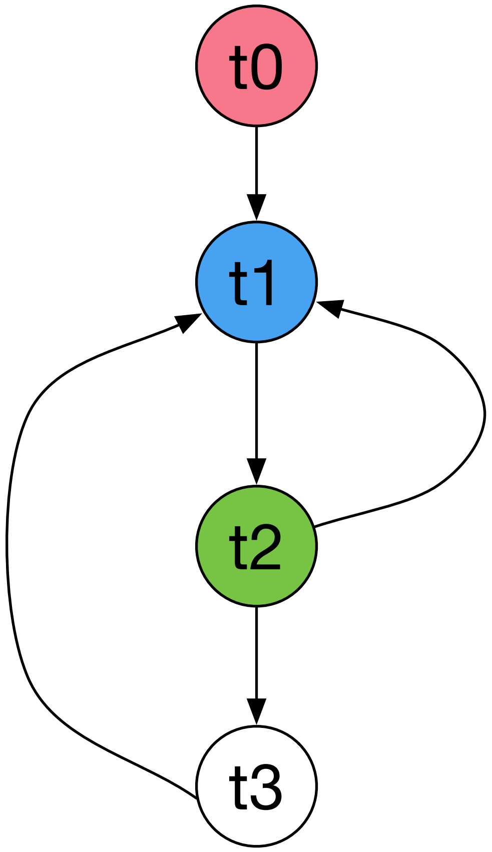


(Control Flow) Path Coverage*

- **(Control Flow) Path Coverage — Testing Goal**
  - Every path between every pair of nodes should be executed (may lead to redundancy)
  - Every path between source and sinks should be executed.
    - Ideal for programs without loops
  - Sufficient number of paths between source and sink nodes are executed such that all edges are executed
  - Every finite path between source and sinks should executed such that each loop is executed at least once (when all paths are considered)
    - Good enough for programs with loops

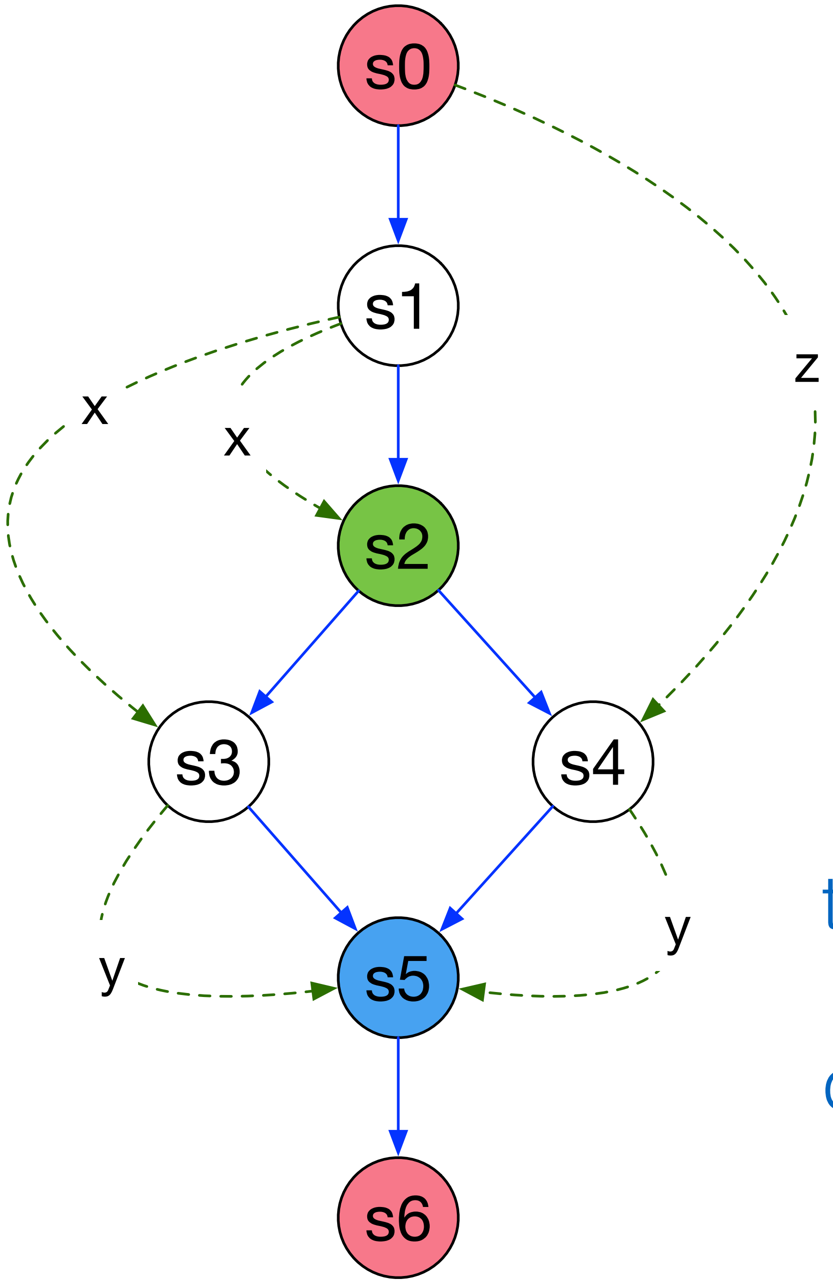
(Control Flow) Path Coverage*

- **(Control Flow) Path Coverage**
  - What about programs without sinks?
  - Every finite path between source and every node is executed with each loop executed at least once (when all paths are considered)
  - What about infeasible paths?



A Program & its DFG

```
s0: z = input()
s1: x = input()
s2: if x > 5:
s3:   y = x * 5
    else:
s4:   y = z / 5
s5: print(y)
s6: return
```



Each node is a statement

Each solid edge is the control flow between two statements

Each dashed edge is the data flow between two statements (from definition of a variable to its use)

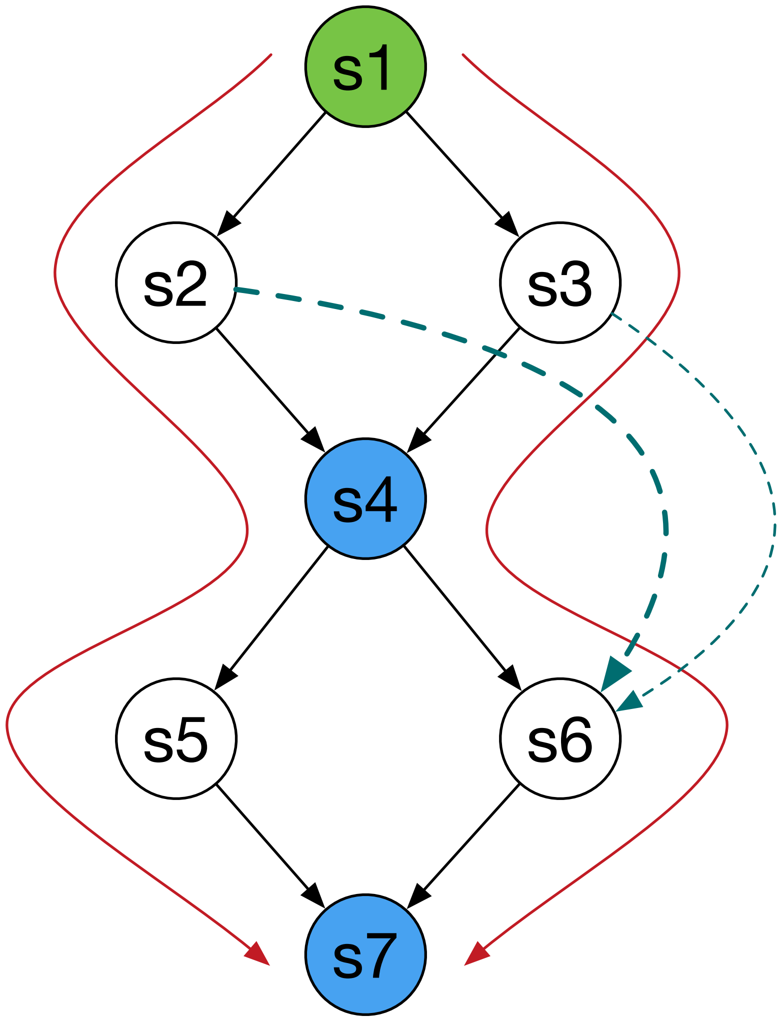
DFG is short for Data Flow Graph

Data Flow Coverage*

- Fraction of def-use edges covered by tests
- *Testing Goal:* Every def-use edge should be executed at least once

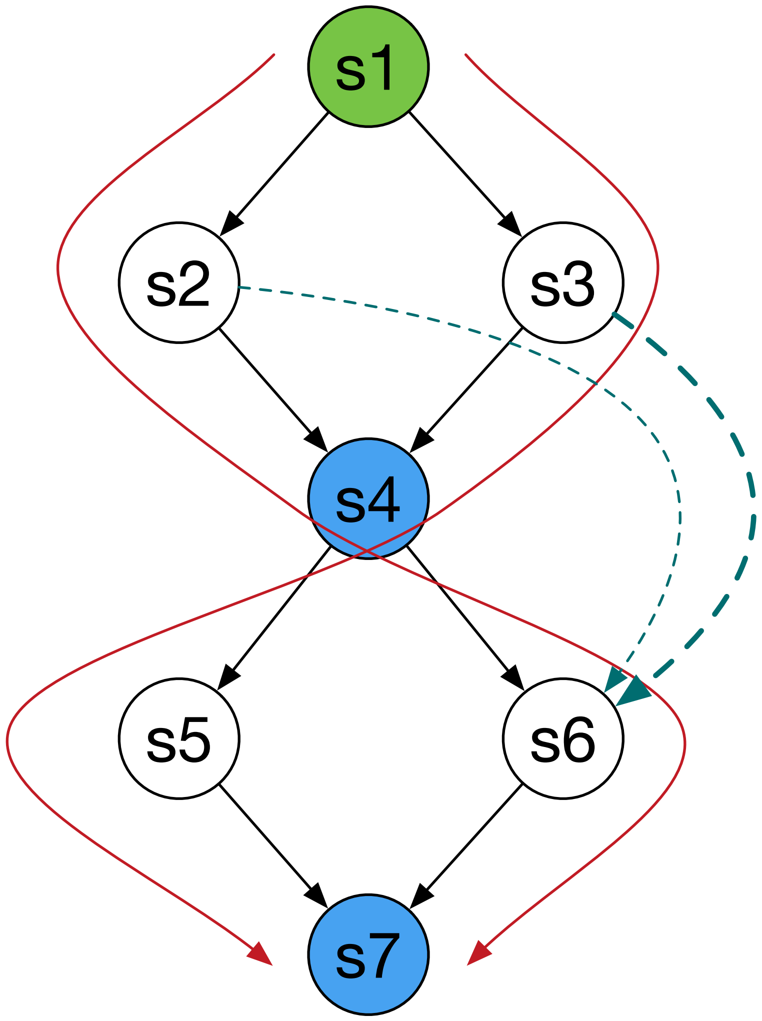
- How well will this work in case of programs with pointers and references?
- Can we detect all def-use edges?
- What about infeasible def-use edges?
- What about in case of programs written in OO languages?

Which is better?



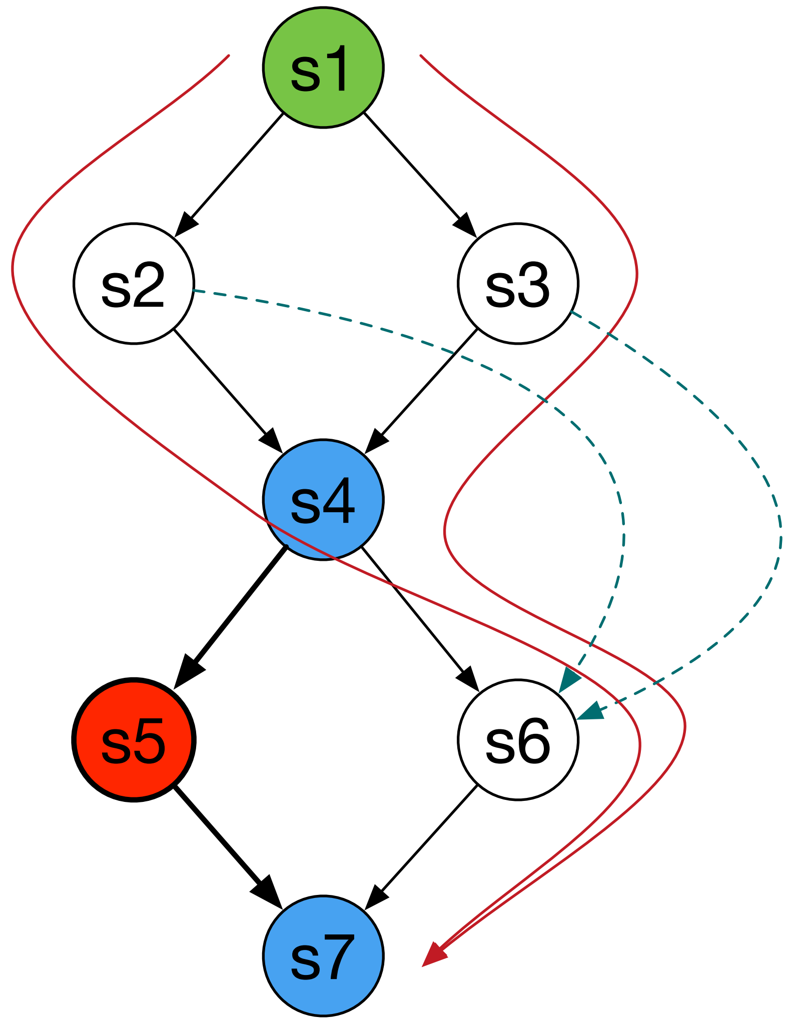
With 2 (red) paths, we get 100% node and edge coverage but miss exercising s2-s6 data flow edge

Which is better?



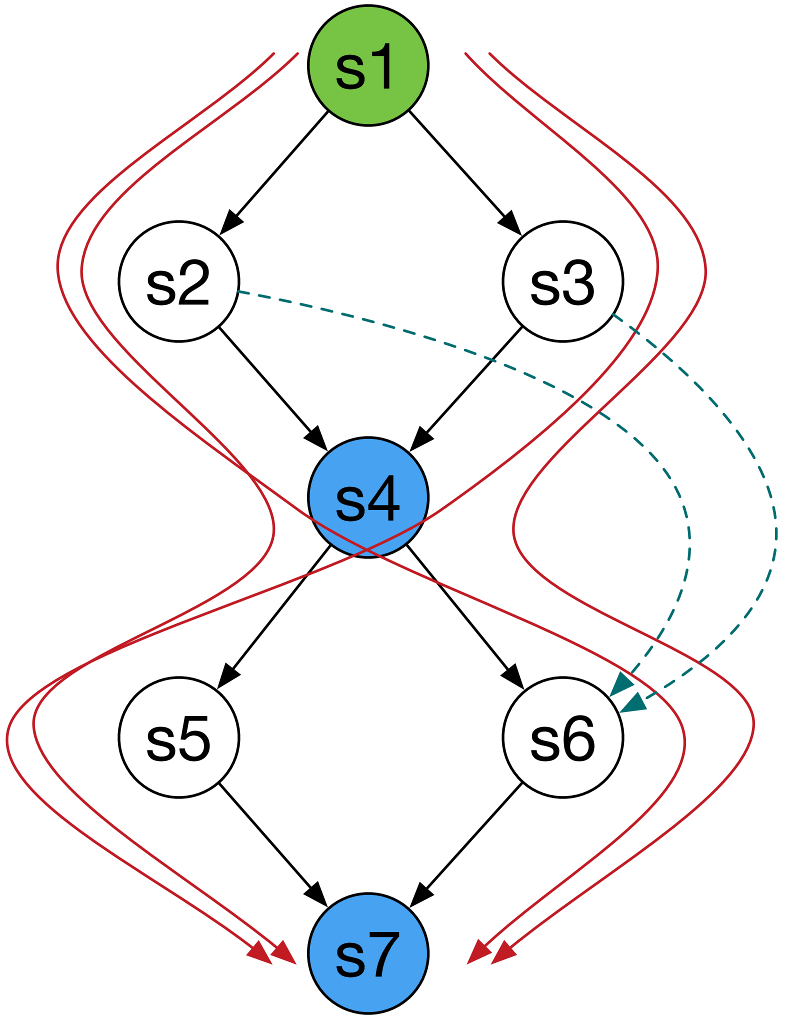
With 2 (red) paths, we get 100% node and edge coverage but miss exercising s3-s6 data flow edge

Which is better?



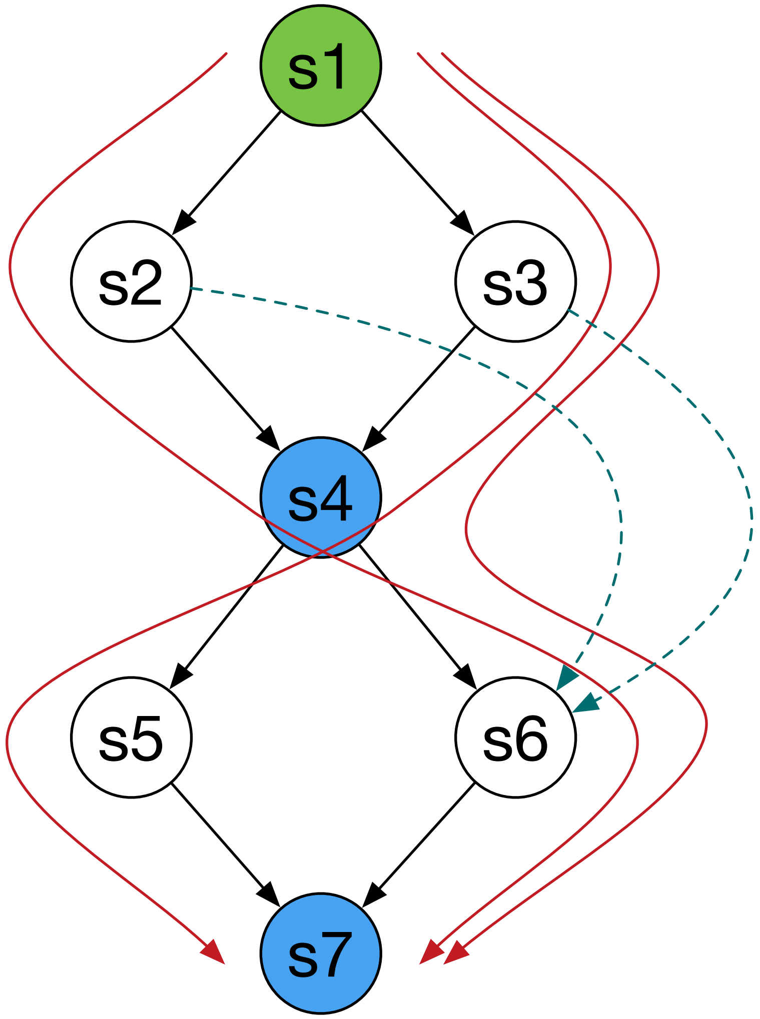
With 2 (red) paths, we get 100% data flow coverage (s2-s6 and s3-s6) but miss exercising s5 node

Which is better?



With 4 (red) paths, we get 100% node, edge, and data flow coverage but with some redundancy

Which is better?



With 3 (red) paths, we get 100% node, edge, and data flow coverage but with lesser redundancy

Branch Coverage*

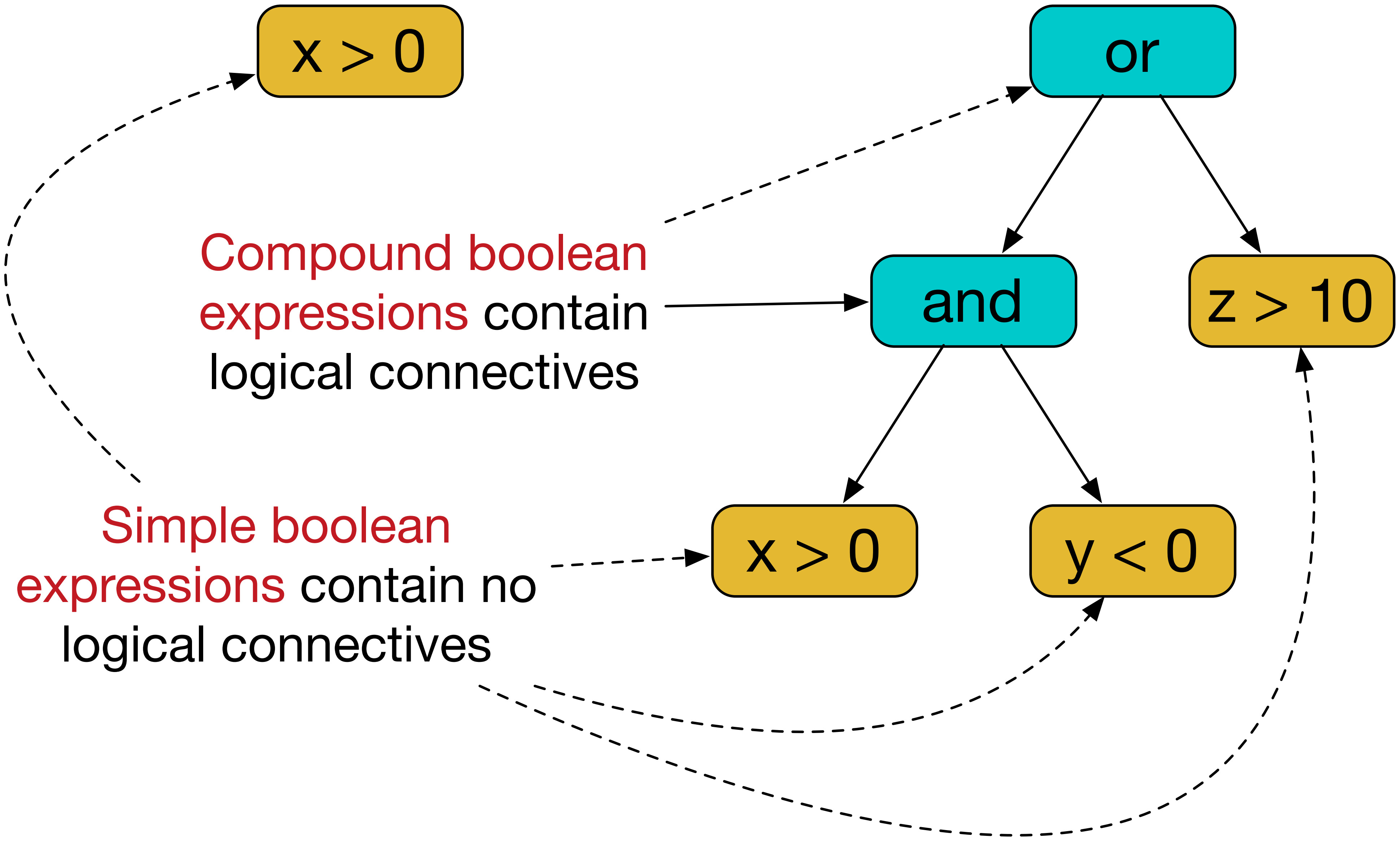
- Fraction of branches (edges) covered by tests
- *Testing Goal:* Every branch should be executed at least once

- How does this relate to node and edge coverage?

Boolean Expressions

$x > 0$

$x > 0$ and $y < 0$ or $z > 10$



Compound boolean expressions contain logical connectives

Simple boolean expressions contain no logical connectives

Condition Coverage*

- Fraction of boolean expression valuations covered by tests
- *Testing Goal:*
  - Every simple boolean expression should be evaluated to both true and false
  - Every compound boolean expression should be evaluated to both true and false

Condition Coverage

- What about coupling between sub-expressions of a compound expression?
  - `(x>0 and y) or (x<=0 and z)`
- What about masking between sub-expressions of a compound expression?
  - `(x>100 and y)` where `x` ranges from 0 to 100