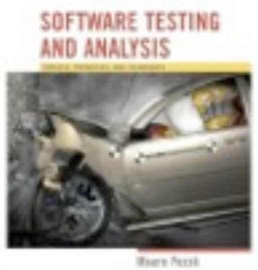
Introduction to structural testing and dataflow testing

Automated testing and verification

J.P. Galeotti - Alessandra Gorla

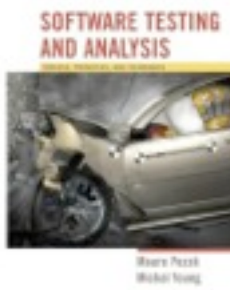
Structural testing

- Judging test suite thoroughness based on the *structure* of the program itself
- Also known as “white-box”, “glass-box”, or “code-based” testing
- To distinguish from functional (requirements-based, “black-box” testing)
- “Structural” testing is still testing product functionality against its specification. Only the measure of thoroughness has changed.



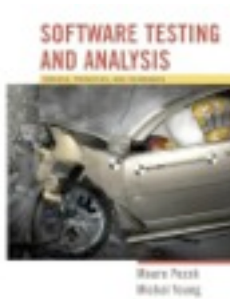
Why structural (code-based) testing?

- One way of answering the question "What is *missing* in our test suite?"
  - If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed
  - But what's a "part"?
    - Typically, a control flow element or combination:
    - Statements (or CFG nodes), Branches (or CFG edges)
    - Fragments and combinations: Conditions, paths
- Complements functional testing: Another way to recognize cases that are treated differently
  - Recall fundamental rationale: Prefer test cases that are treated *differently* over cases treated the same



No guarantees

- Executing all control flow elements does not guarantee finding all faults
  - Execution of a faulty statement may not always result in a failure
    - The state may not be corrupted when the statement is executed with some data values
    - Corrupt state may not propagate through execution to eventually lead to failure
- What is the value of structural coverage?
  - Increases confidence in thoroughness of testing
    - Removes some obvious *inadequacies*



Structural testing **complements** functional testing

- Control flow testing includes cases that may not be identified from specifications alone
  - Typical case: implementation of a single item of the specification by multiple parts of the program
  - Example: hash table collision (invisible in interface spec)
- Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria
  - Typical case: missing path faults

Structural testing in practice

- Create functional test suite first, then measure structural coverage to see what is missing
- Interpret unexecuted elements
  - may be due to natural differences between specification and implementation
  - or may reveal flaws of the software or its development process
    - inadequacy of specifications that do not include cases present in the implementation
    - coding practice that radically diverges from the specification
    - inadequate functional test suites
- Attractive because automated
  - coverage measurements are convenient progress indicators
  - sometimes used as a criterion of completion
    - use with caution: does not ensure *effective* test suites

Statement testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

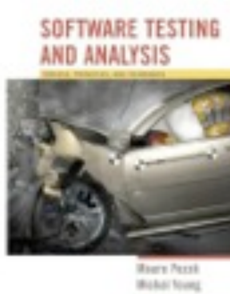
- Coverage:

$$\frac{\#\text{ executed statements}}{\#\text{ statements}}$$

- **Rationale**: a fault in a statement can only be revealed by executing the faulty statement

Statements or blocks?

- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to *basic block* coverage or *node coverage*
  - Difference in granularity, not in concept
- No essential difference
  - 100% node coverage <-> 100% statement coverage
    - but levels will differ below 100%
  - A test case that improves one will improve the other
    - though not by the same amount, in general

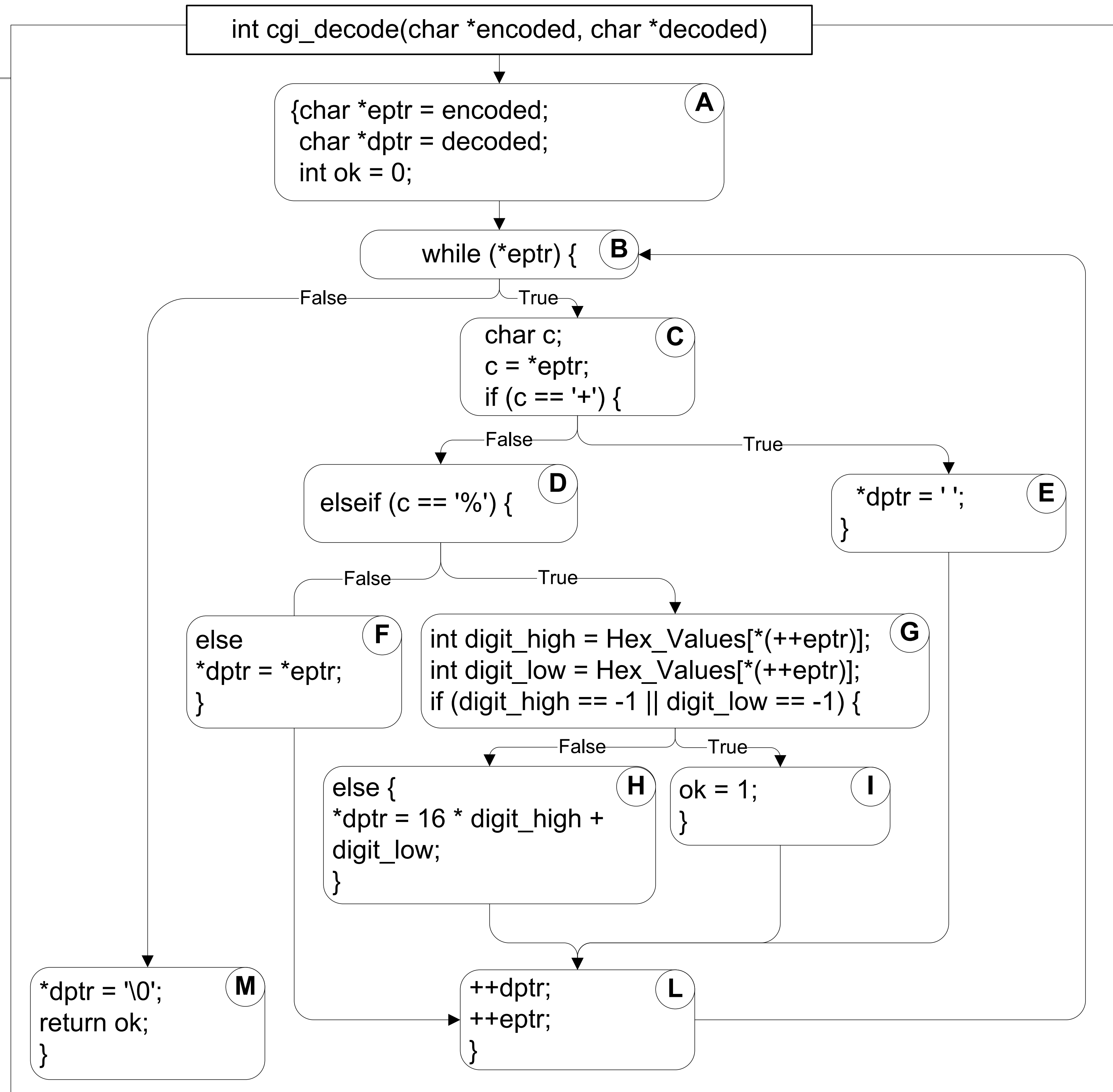


Example

T_0 =
{“”, “test”,
“test+case%1Dadequacy”}
17/18 = 94% Stmt Cov.

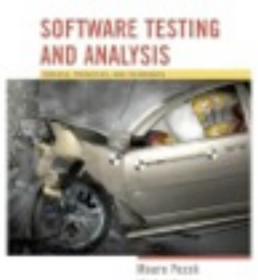
T_1 =
{“adequate+test
%0Dexecution%7U”}
18/18 = 100% Stmt Cov.

T_2 =
{“%3D”, “%A”, “a+b”,
“test”}
18/18 = 100% Stmt Cov.



Coverage is not size

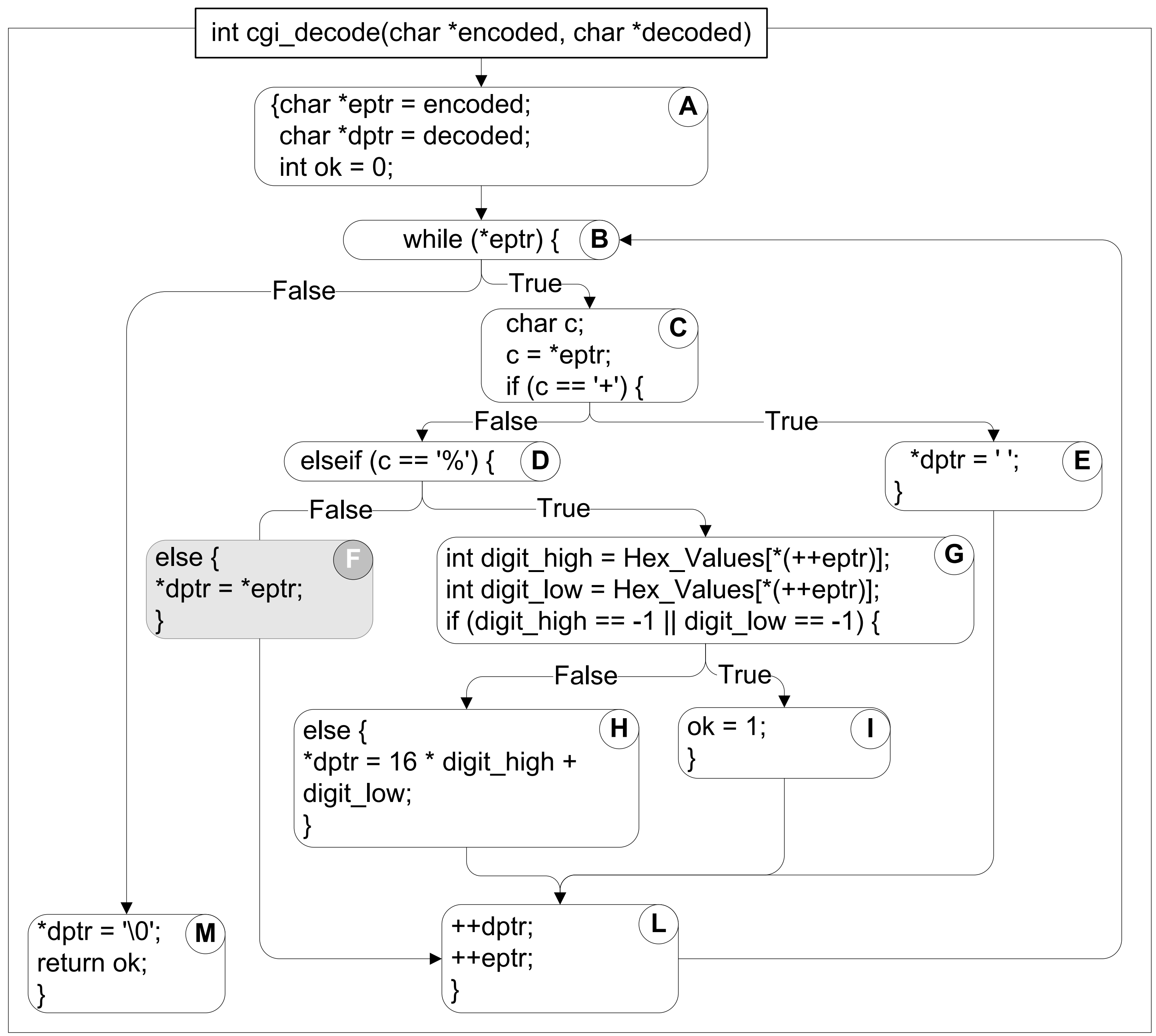
- Coverage does not depend on the number of test cases
  - $T_0, T_1 : T_1 >_{coverage} T_0 \qquad T_1 <_{cardinality} T_0$
  - $T_1, T_2 : T_2 =_{coverage} T_1 \qquad T_2 >_{cardinality} T_1$
- Minimizing test suite size is seldom the goal
  - small test cases make failure diagnosis easier
  - a failing test case in T_2 gives more information for fault localization than a failing test case in T_1



“All statements” can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose block F were missing
  - Statement adequacy would not require *false* branch from D to L

T_3 =
{“”, “+%0D+%4J”}
100% Stmt Cov.
No *false* branch from D



Branch testing

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

T_3 = {“”, “+%0D+%4J”}

100% Stmt Cov.   88% Branch Cov. (7/8 branches)

T_2 = {“%3D”, “%A”, “a+b”, “test”}

100% Stmt Cov.   100% Branch Cov. (8/8 branches)

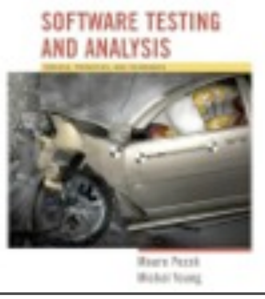
Statements vs branches

- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true (see T_3)
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

Subsume relation

- Branch criterion subsumes statement criterion

Does this mean that if it is possible to find a fault with a test suite that satisfies statement criterion then the same fault will be discovered by any other test suite satisfying branch criterion?



Subsume relation

- Branch criterion subsumes statement criterion

Does this mean that if it is possible to find a fault with a test suite that satisfies statement criterion then the same fault will be discovered by any other test suite satisfying branch criterion?

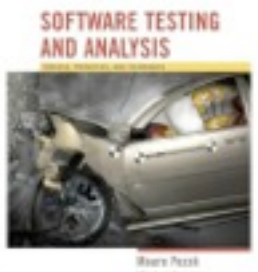
NO!

“All branches” can still miss conditions

- Sample fault: missing operator (negation)

  digit_high == 1 || digit_low == -1

- Branch adequacy criterion can be satisfied by varying only digit_high
  - The faulty sub-expression might never determine the result
  - We might never really test the faulty condition, even though we tested both outcomes of the branch



Other structural testing criteria

- Basic condition testing
- Compound conditions testing
- MC/DC
- Path testing
- Boundary interior testing
- .....
    - **(to be continued...)**

Dataflow testing

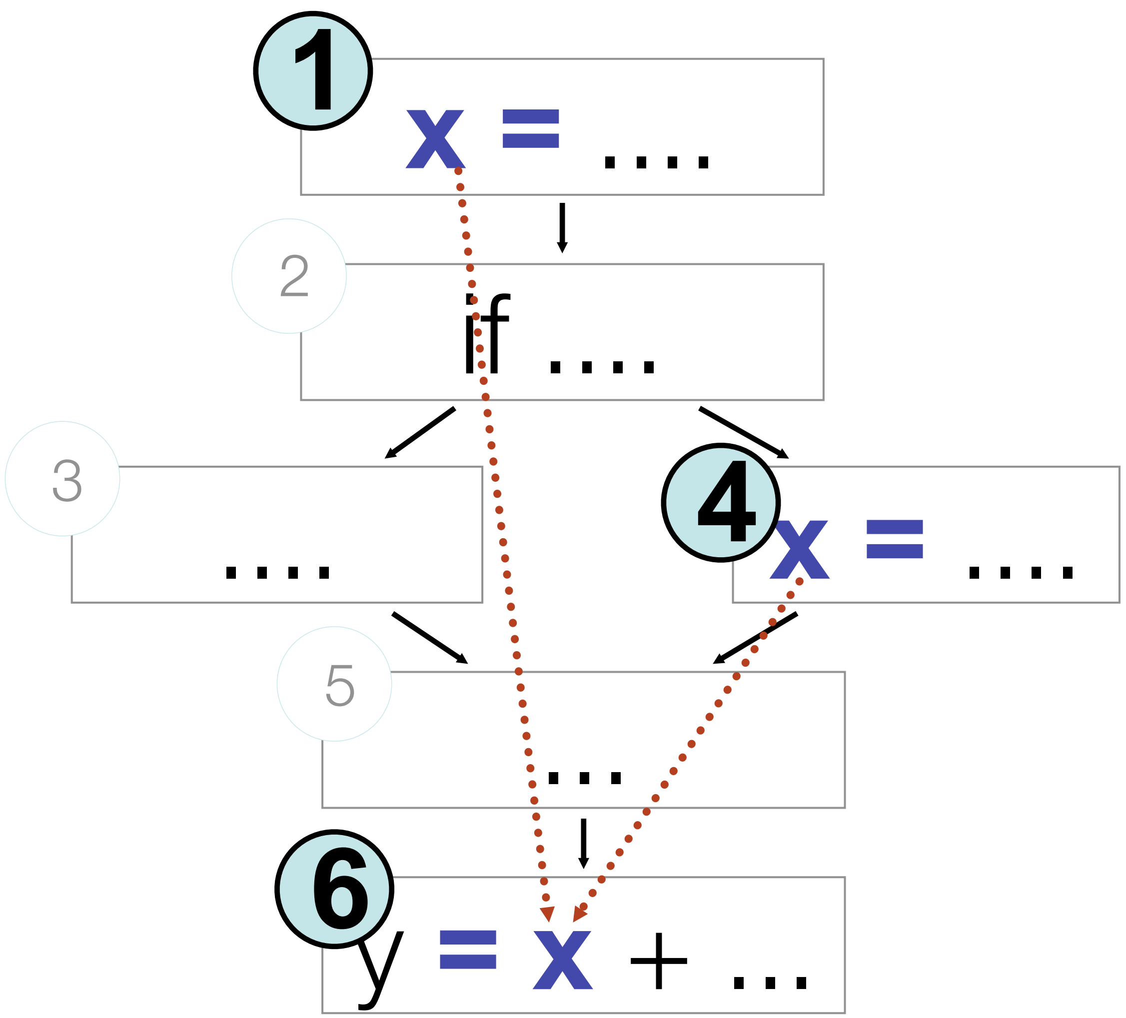
Automated testing and verification

J.P. Galeotti - Alessandra Gorla

Motivation

- Middle ground in structural testing
  - Node and edge coverage don't test interactions
  - Path-based criteria require impractical number of test cases
    - And only a few paths uncover additional faults, anyway
  - Need to distinguish "important" paths
- Intuition: Statements interact through *data flow*
  - Value computed in one statement, used in another
  - Bad value computation revealed only when it is used

Dataflow concept



- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are *def-use (DU) pairs*
  - defs at 1,4
  - use at 6

Terms

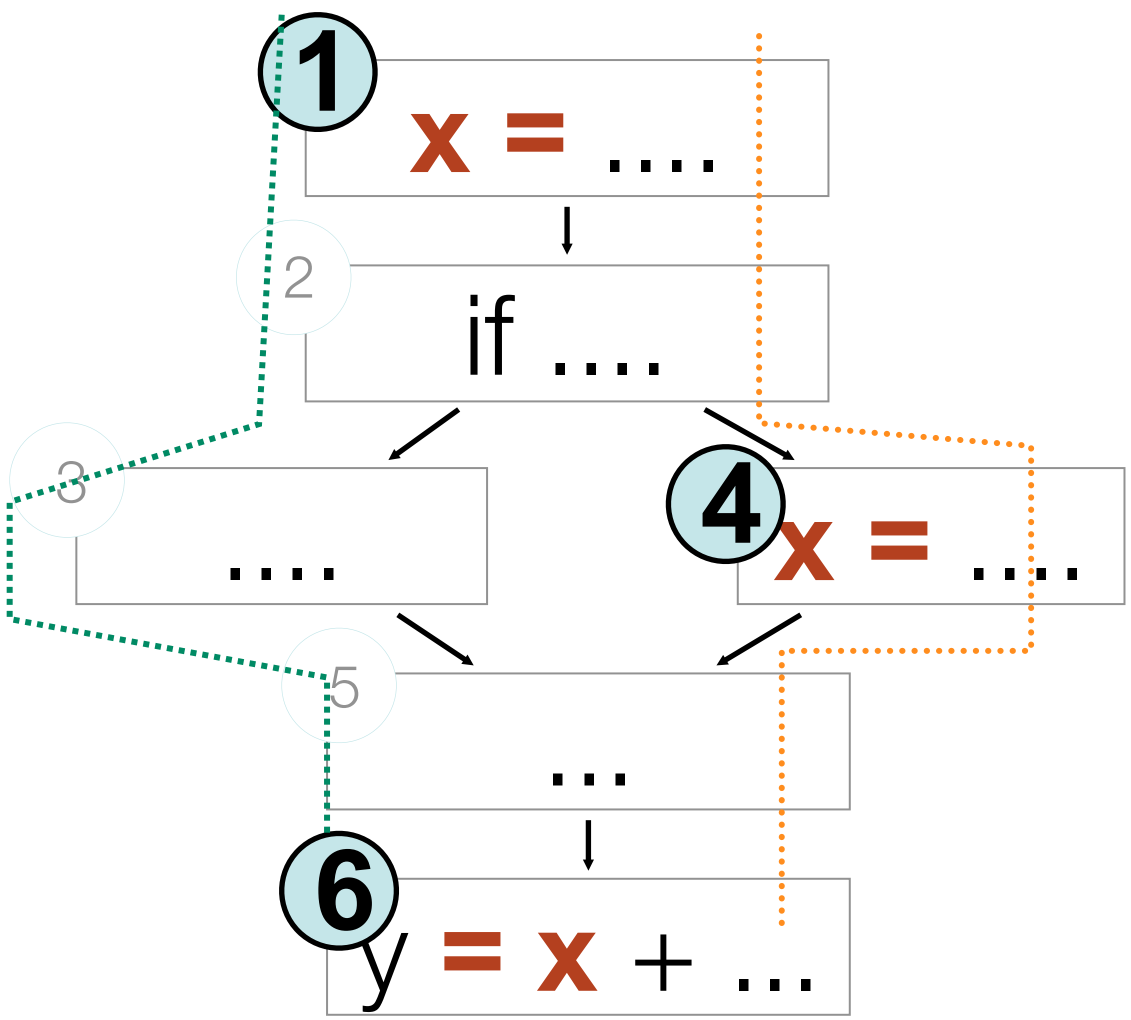
- DU pair: a pair of *definition* and *use* for some variable, such that at least one DU path exists from the definition to the use

  x = ... is a *definition* of x

  = ... x ... is a *use* of x

- DU path: a definition-clear path on the CFG starting from a definition to a use of a same variable
  - Definition clear: Value is not replaced on path
  - Note – loops could create infinite DU paths between a def and a use

Definition-clear path

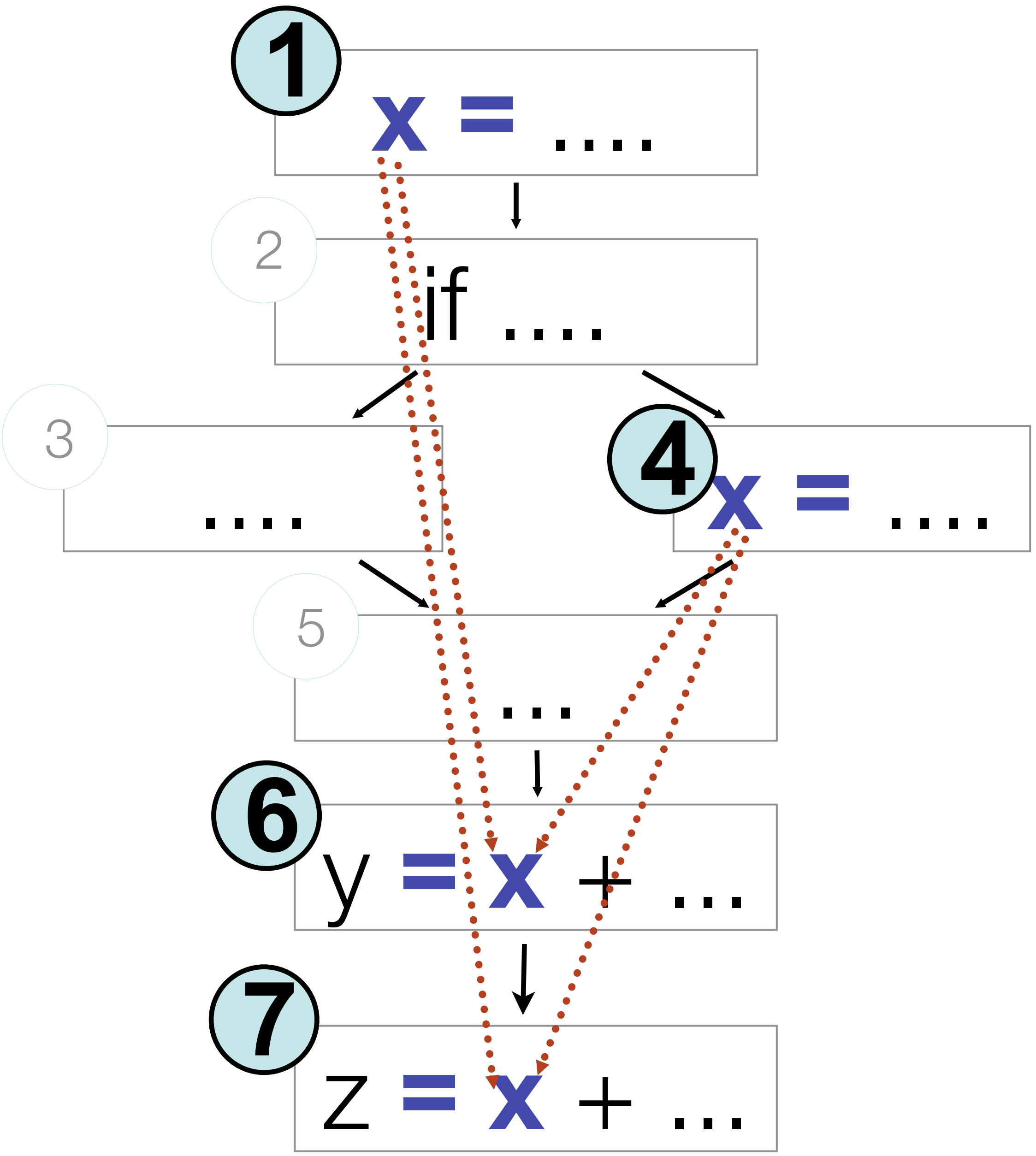


- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is "killed" (reassigned) at node 4
- (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

Adequacy criteria

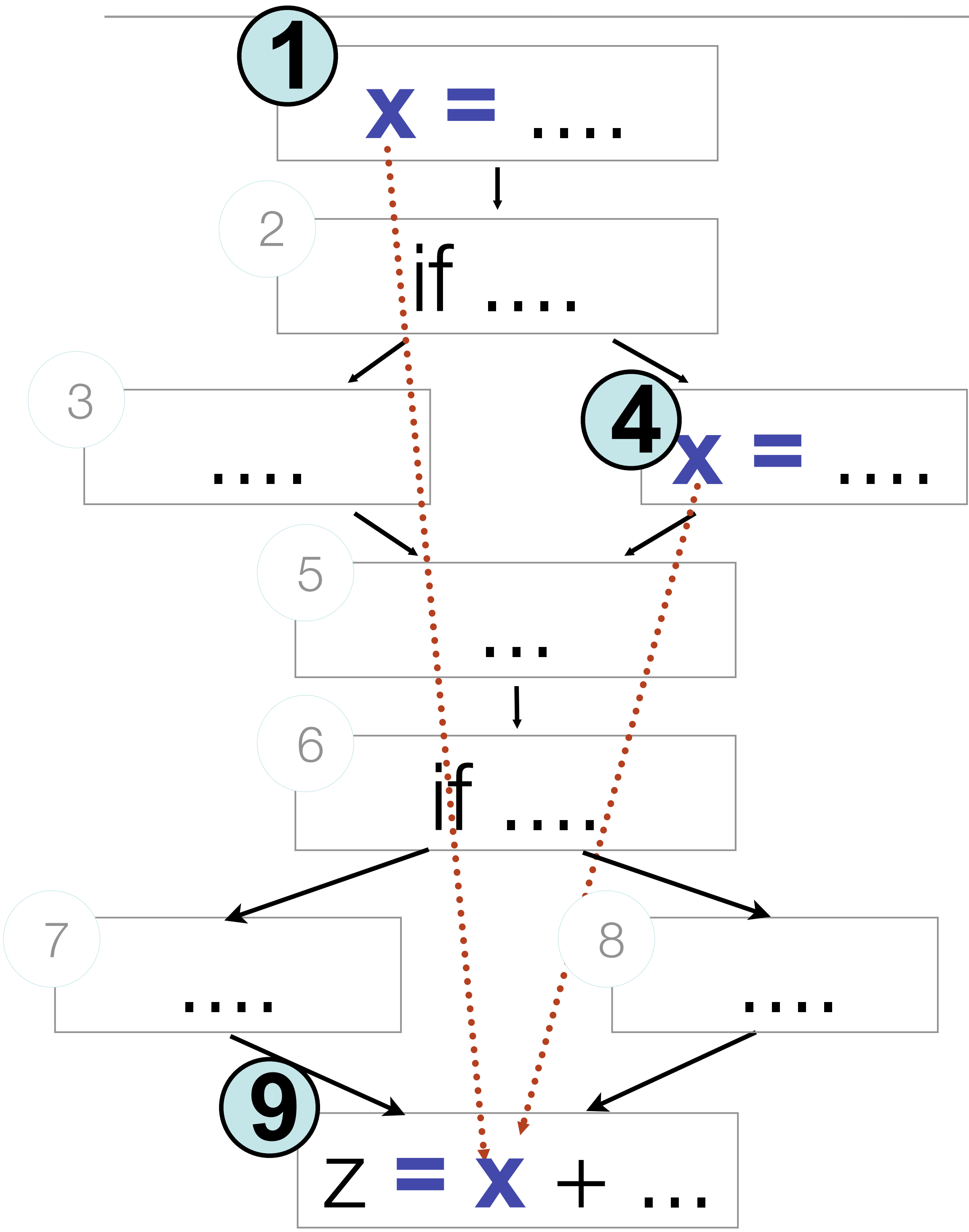
- **All DU pairs**: Each DU pair is exercised by at least one test case
- **All DU paths**: Each *simple* (non looping) DU path is exercised by at least one test case
- **All definitions**: For each definition, there is at least one test case which exercises a DU pair containing it
    - (Every computed value is used somewhere)

All du pairs (all-uses)



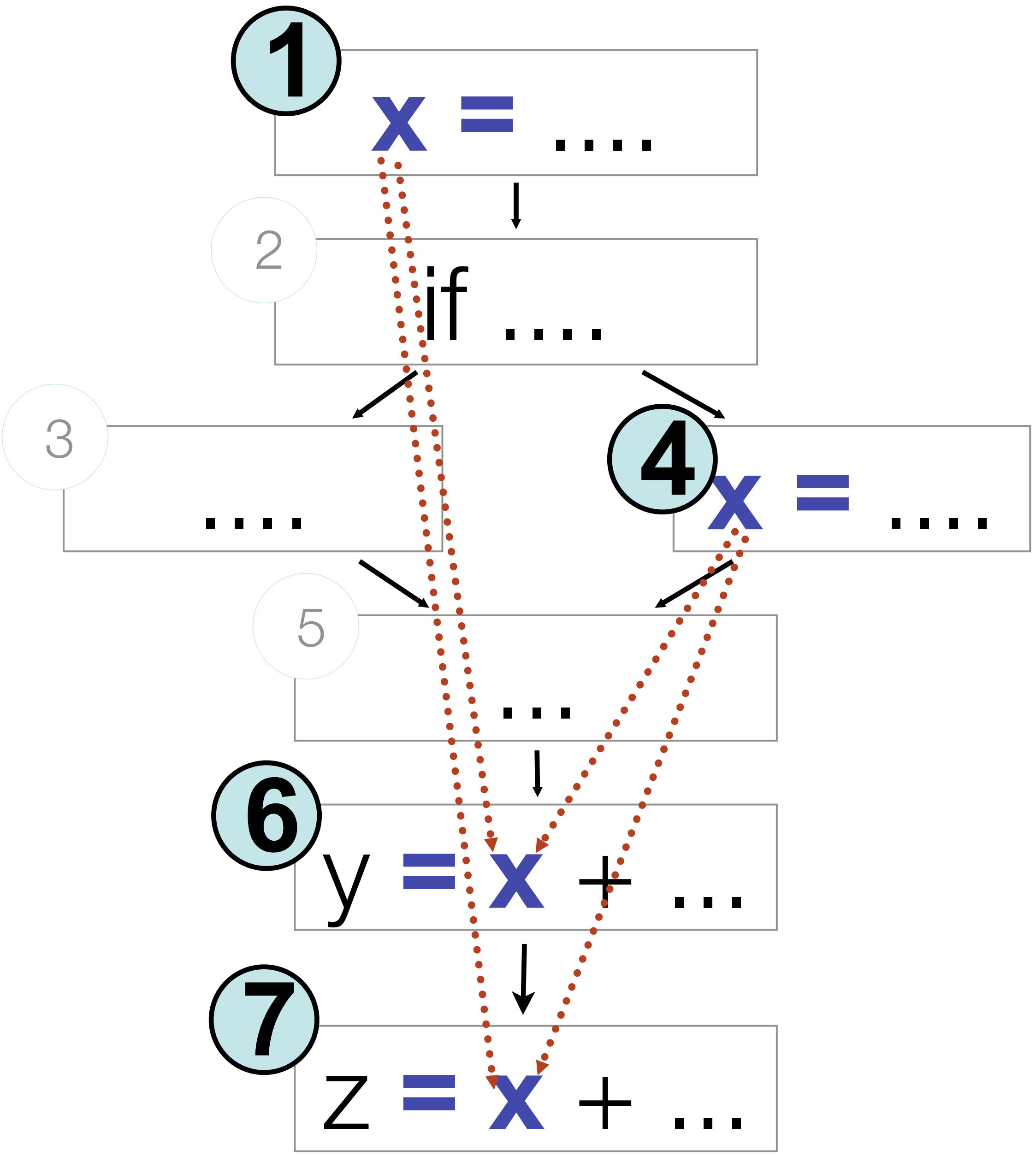
- Requires to cover all the following pairs:
  - def at 1 - use at 6
  - def at 1 - use at 7
  - def at 4 - use at 6
  - def at 4 - use at 7

All du paths



- Requires to cover all the following pairs:
  - def at 1 - use at 9 (through 7)
  - def at 1 - use at 9 (through 8)
  - def at 4 - use at 9 (through 7)
  - def at 4 - use at 9 (through 8)

All definitions

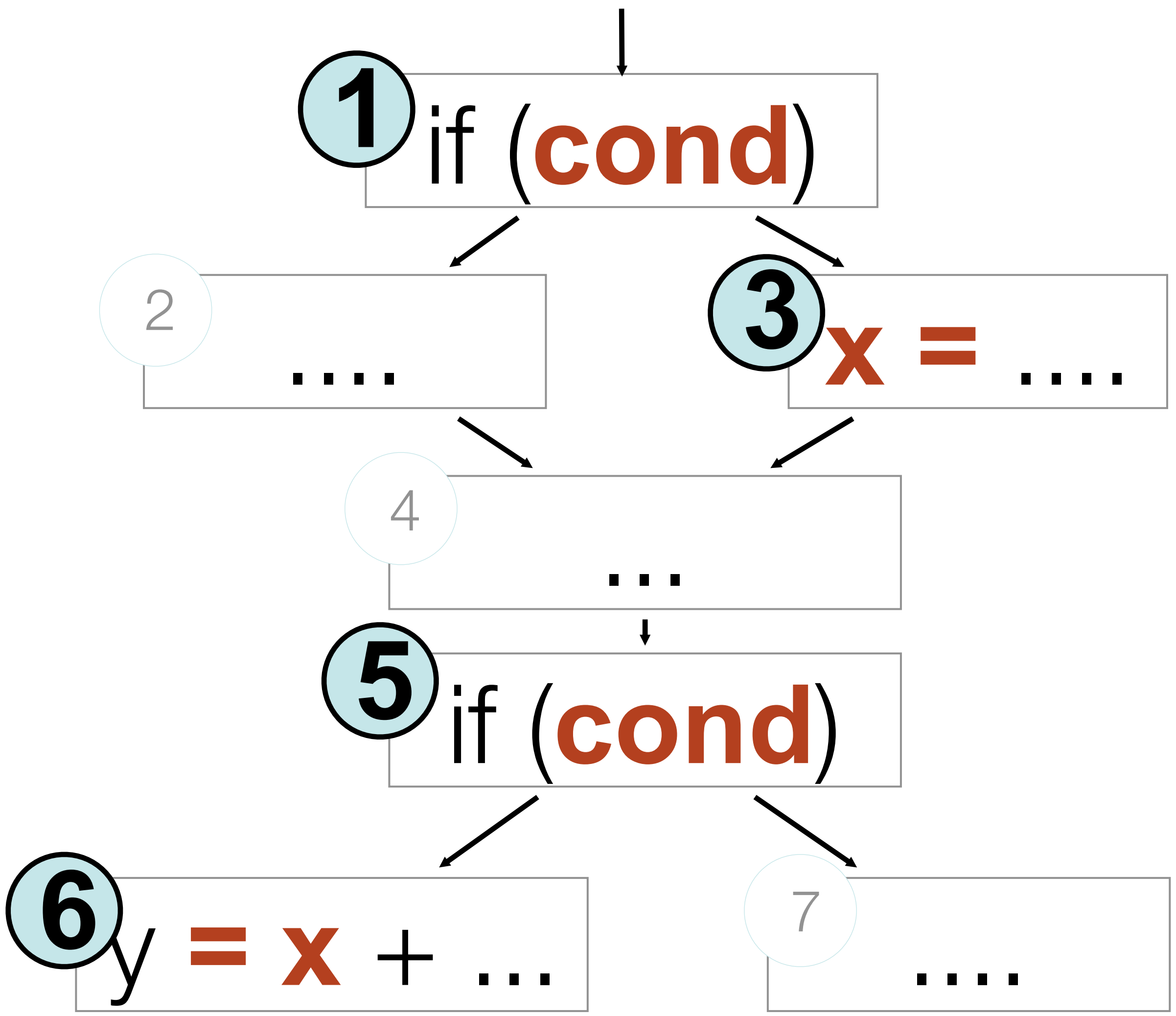


- Requires to cover 2 pairs:
  - def at 1 - use at 6

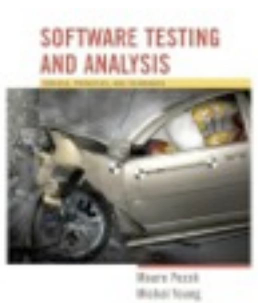
    OR
  - def at 1 - use at 7
  - def at 4 - use at 6

    OR
  - def at 4 - use at 7

Infeasibility

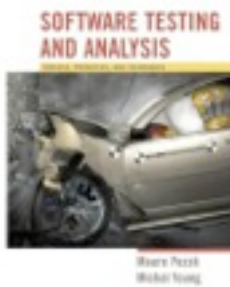


- Suppose *cond* has not changed between 1 and 5
    - Or the conditions could be different, but the first implies the second
- Then (3,5) is not a (feasible) DU pair
    - But it is difficult or impossible to determine which pairs are infeasible
- Infeasible test obligations are a problem
    - No test case can cover them



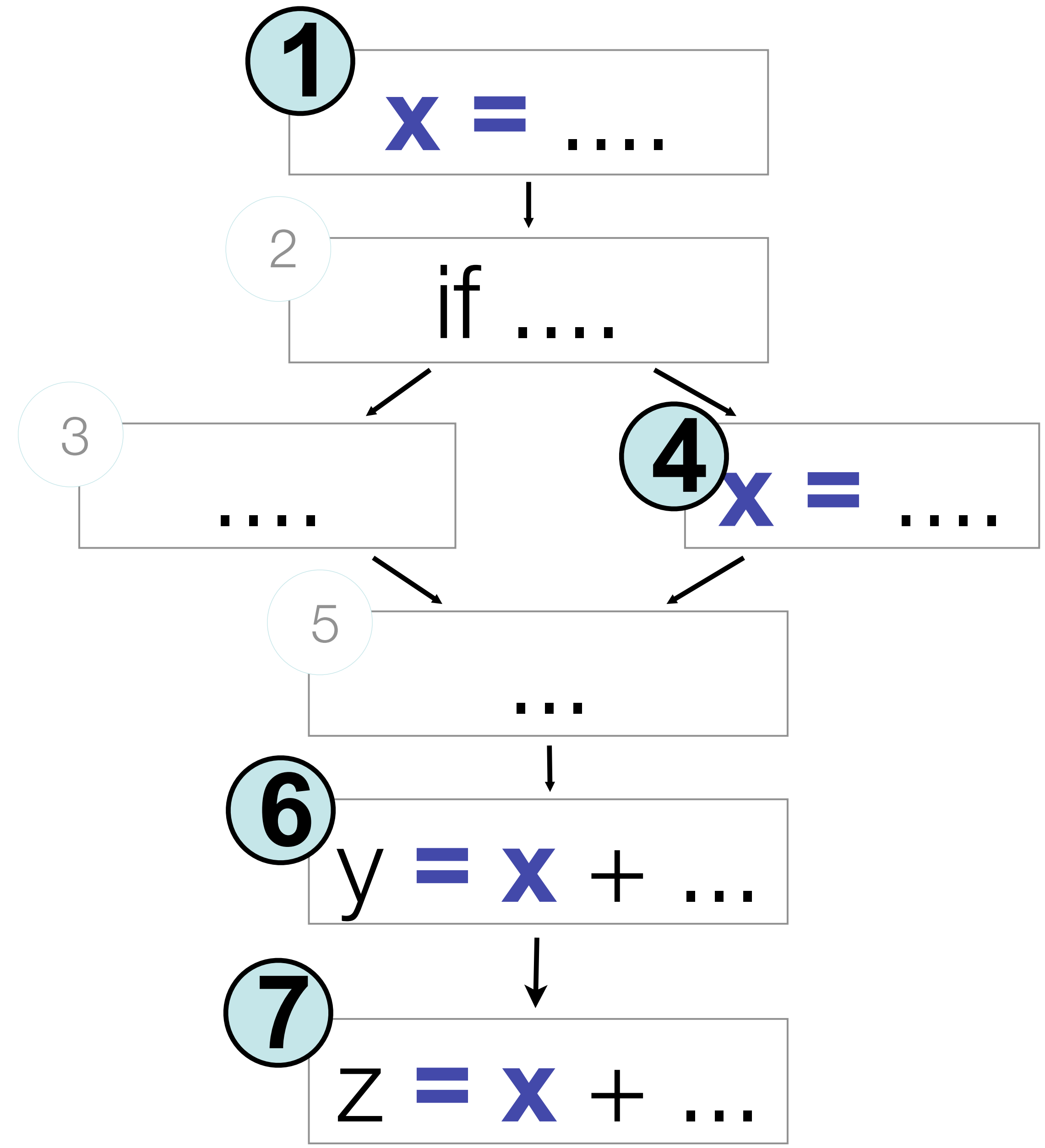
Infeasibility

- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant
  - Combinations of elements matter!
  - Impossible to (infallibly) distinguish feasible from infeasible paths. More paths = more work to check manually.
- In practice, reasonable coverage is (often, not always) achievable
  - Number of paths is exponential in worst case, but often linear
  - All DU *paths* is more often impractical



Reaching definitions analysis

- It is a **forward may** analysis



in[n], out[n] = set of definitions of variables

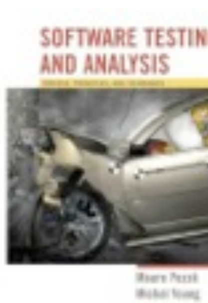
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

$\oplus = \cup$ (of sets)

in$[n] := \cup \{$ ***out***$[m] \mid m$ ***pred***$(n) \}$

out$[n] :=$ **gen**$(n) \cup ($***in***$[n]$ - ***kill***$[n])$



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

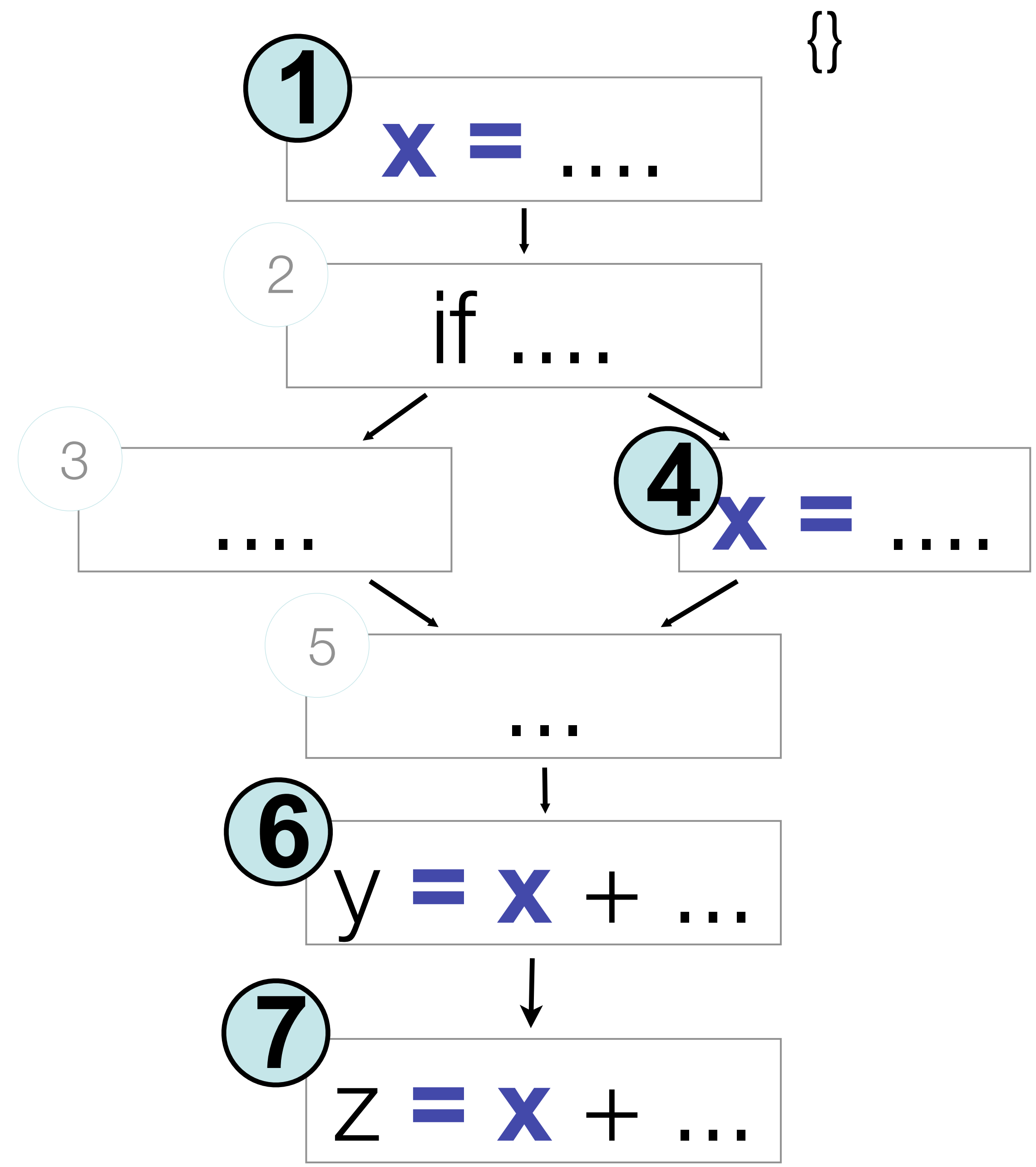
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

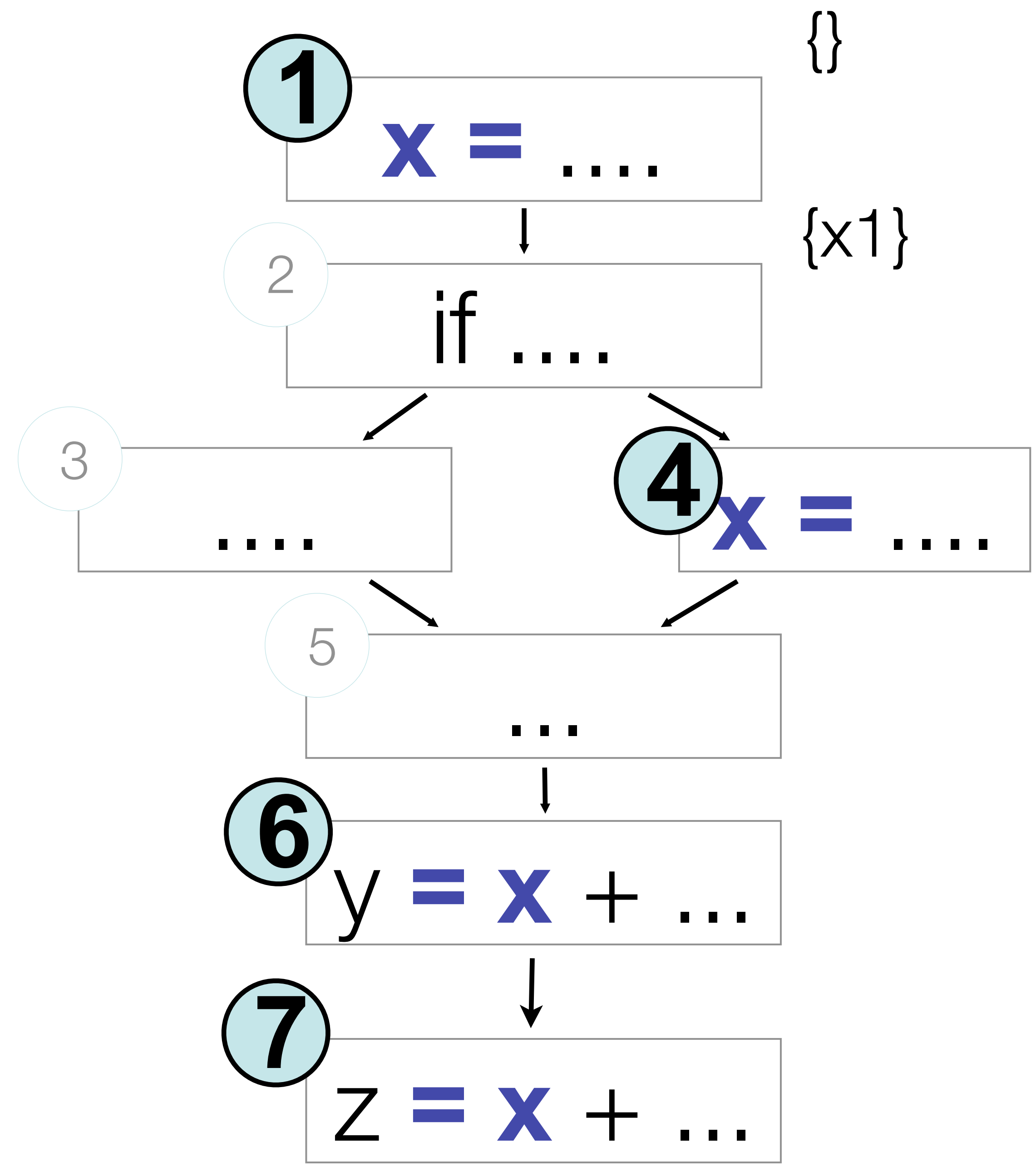
gen(n) = vn where var v is defined at node n

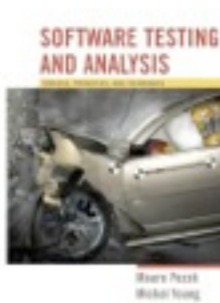
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])





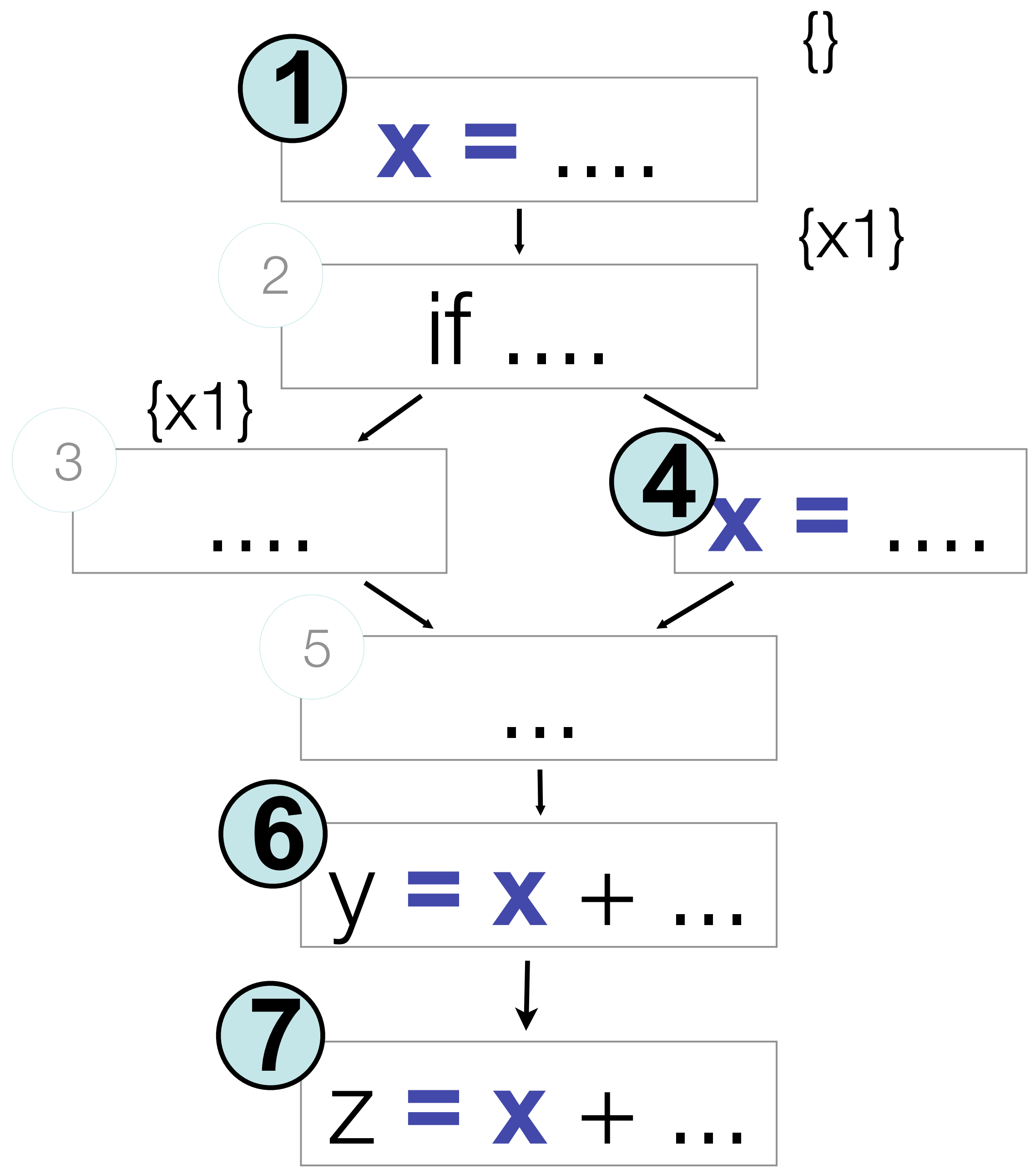
Reaching definitions analysis

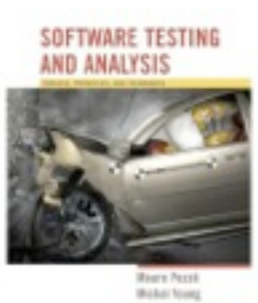
- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables
    gen(n) = vn where var v is defined at node n
    kill(n) = vx where var v is defined at node n and x
⊕ = ∪ (of sets)

in[n] := ∪ { ***out***[m] | m ***pred***(n) }

out[n] := **gen**(n) ∪ (***in***[n] - ***kill***[n])





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

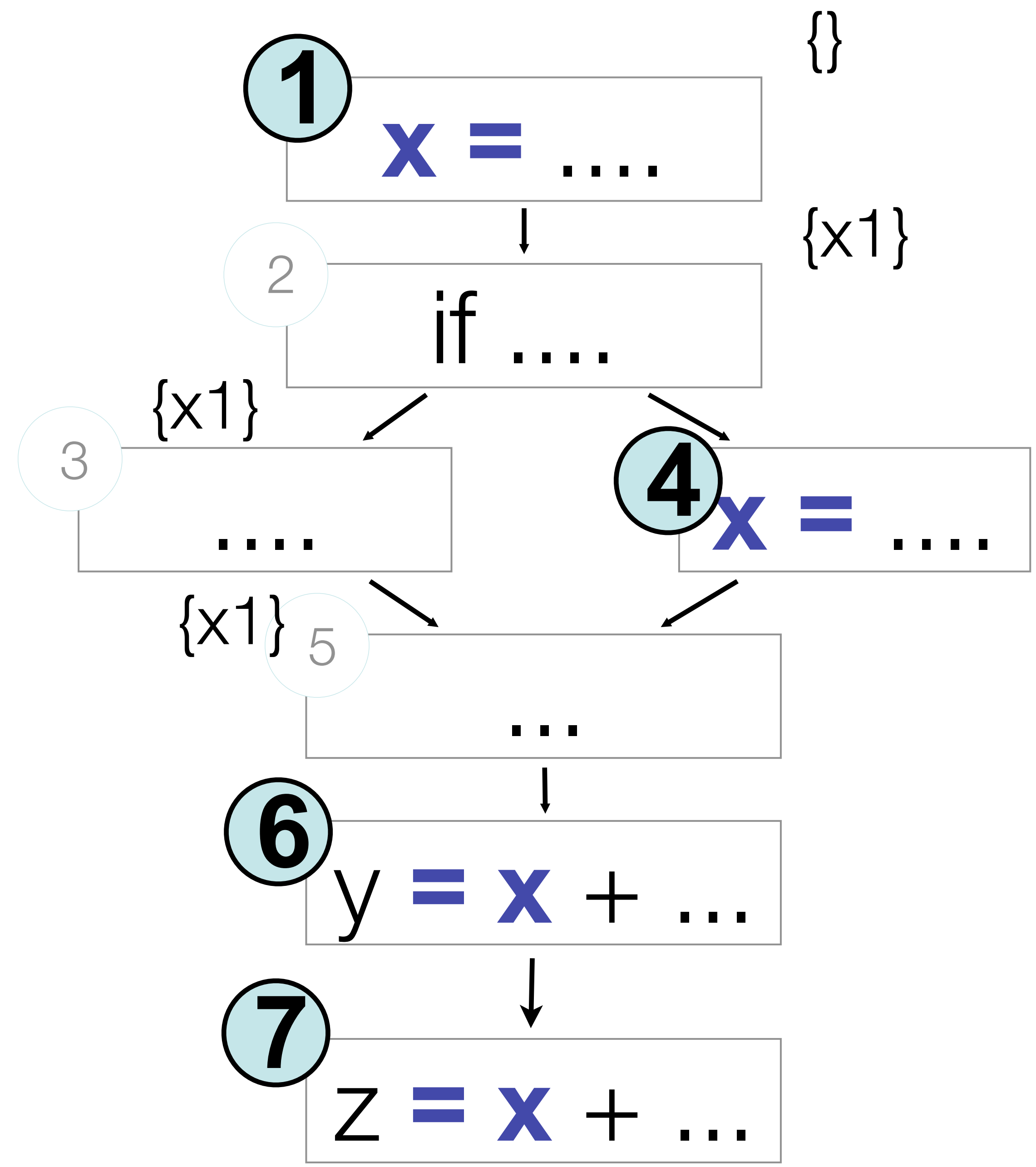
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[n] := ∪ { ***out***[m] | m ***pred***(n) }

out[n] := **gen**(n) ∪ (***in***[n] - ***kill***[n])



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

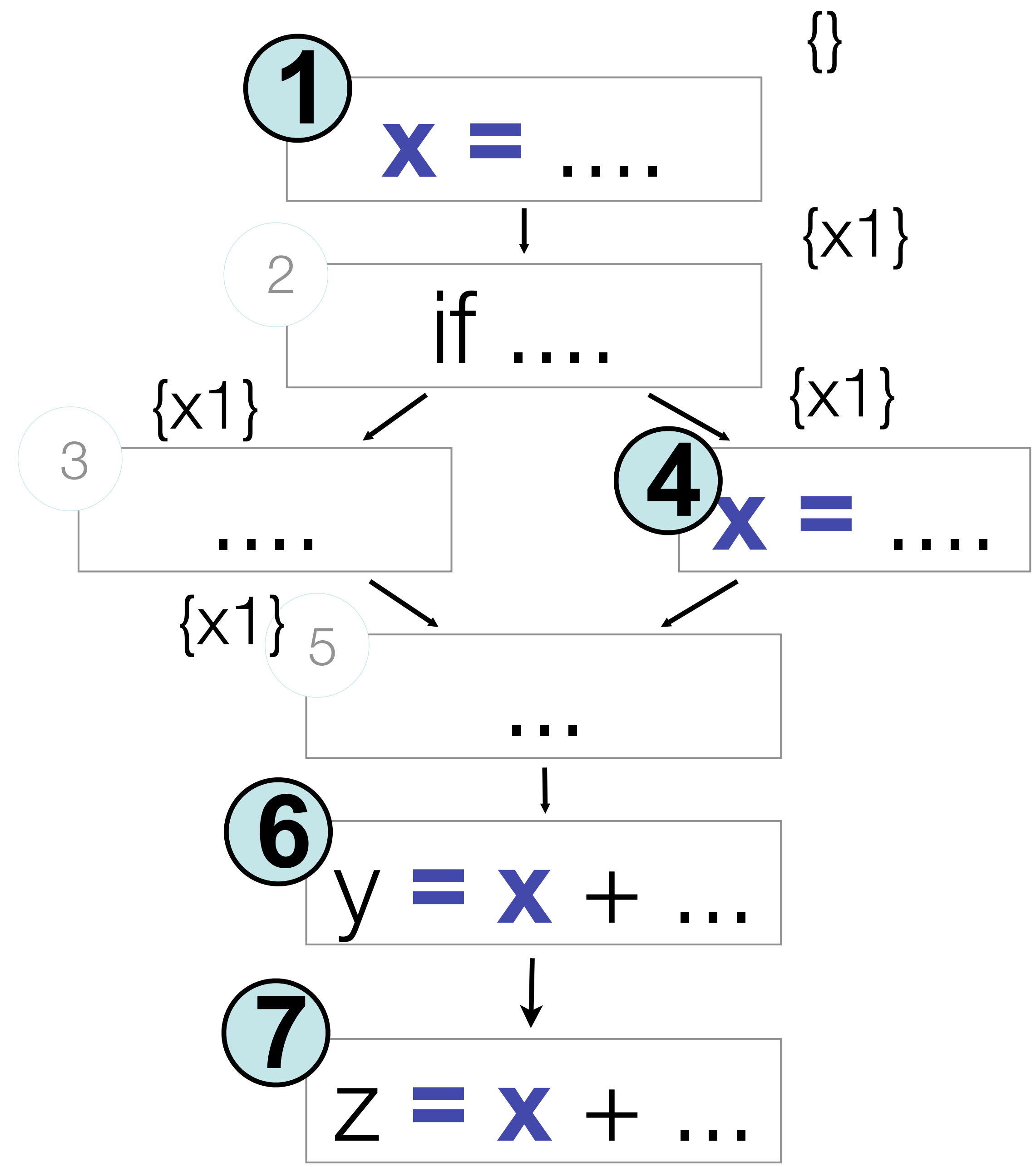
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

$\oplus$ = ∪ (of sets)

***in**[n] :=* ∪ *{ **out**[m] | m **pred**(n) }*

out[n] := **gen**(n) ∪ (***in**[n]* - ***kill**[n])*



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

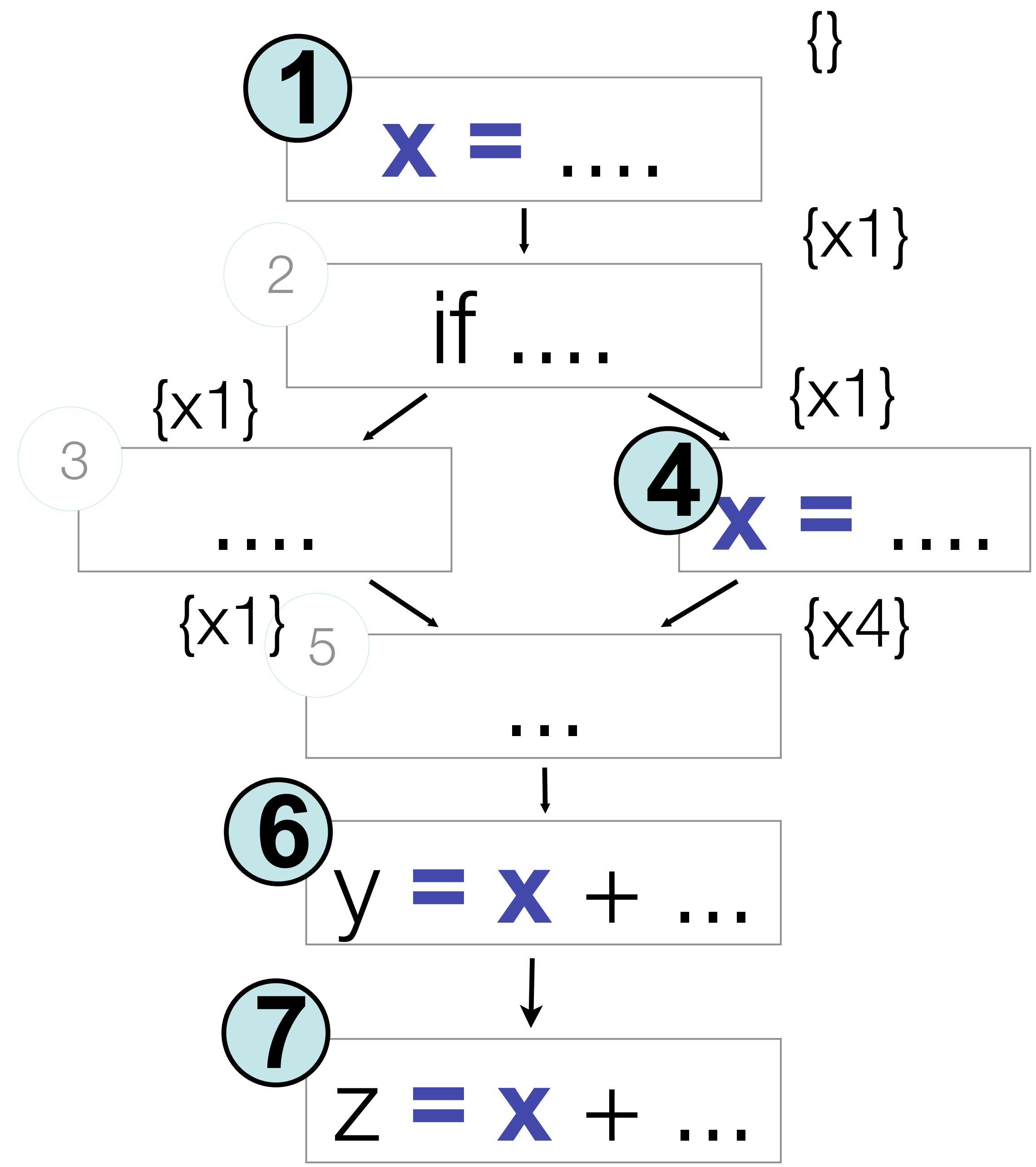
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

$\oplus$ = ∪ (of sets)

***in**[n]* := ∪ *{ **out**[m] | m **pred**(n) }*

out[n] := **gen**(n) ∪ (***in**[n] - **kill**[n])*



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

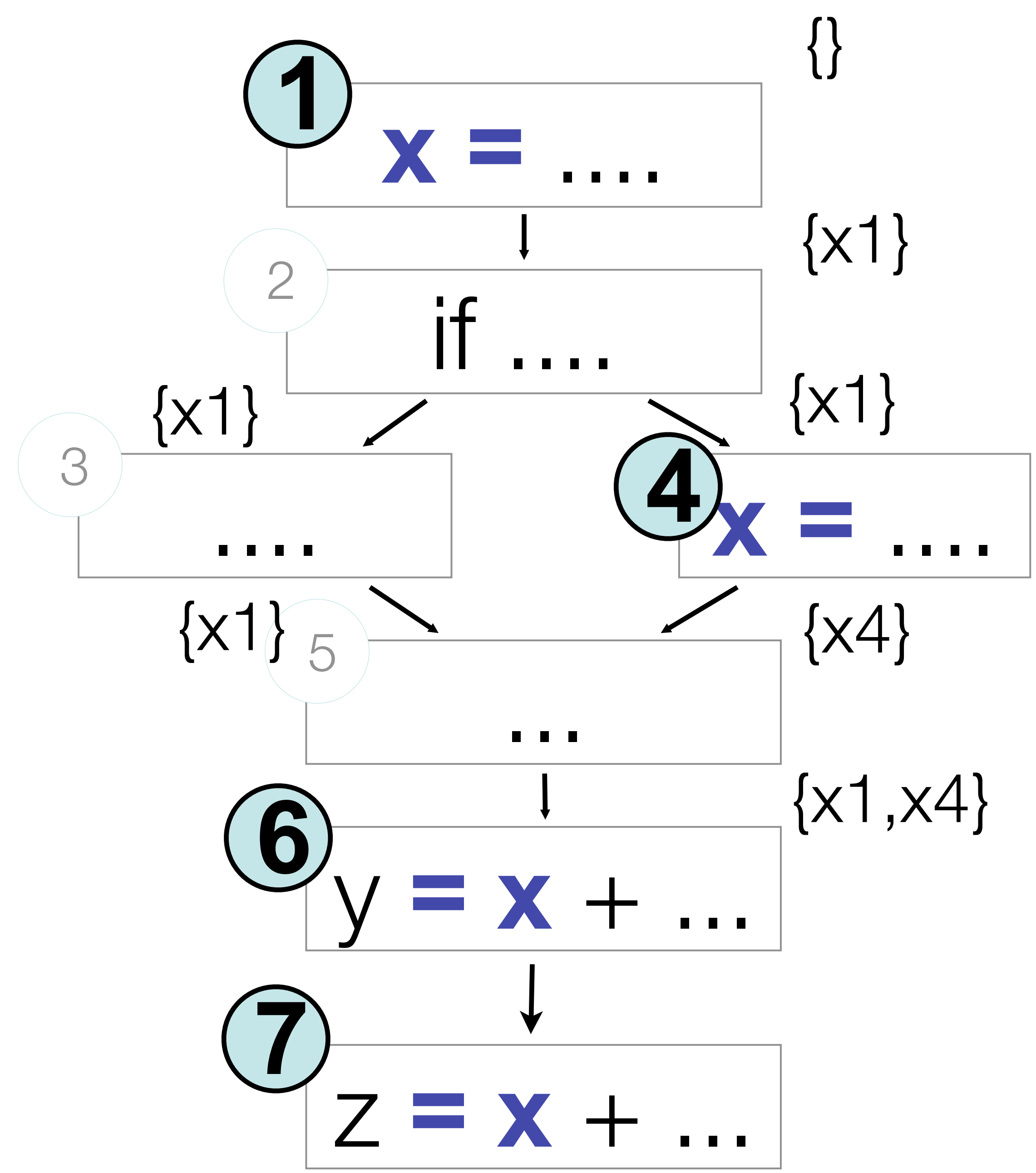
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

***in**[n] :=* ∪ *{ **out**[m] | m **pred**(n) }*

out[n] := **gen**(n) ∪ (***in**[n] - **kill**[n])*



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

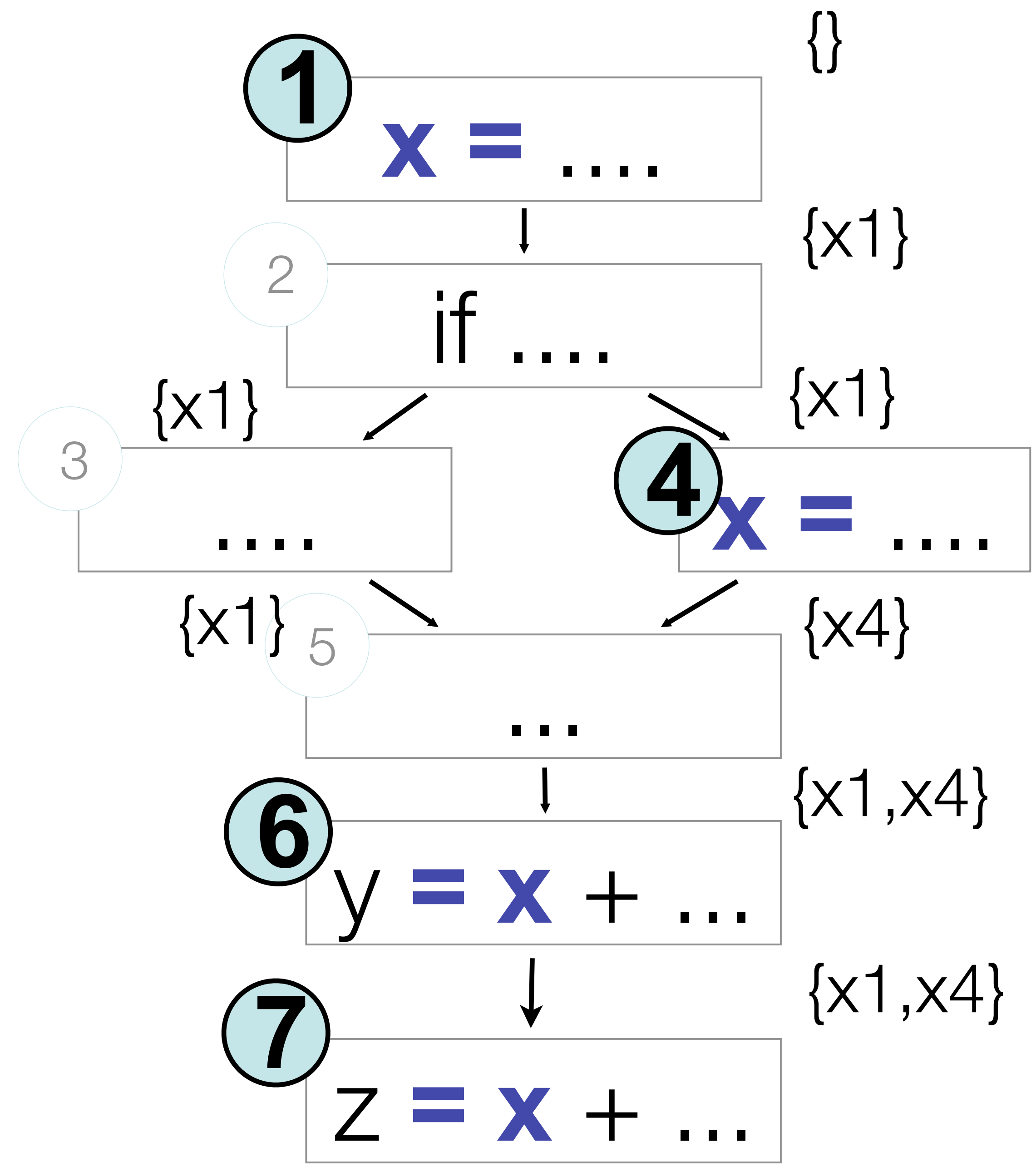
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

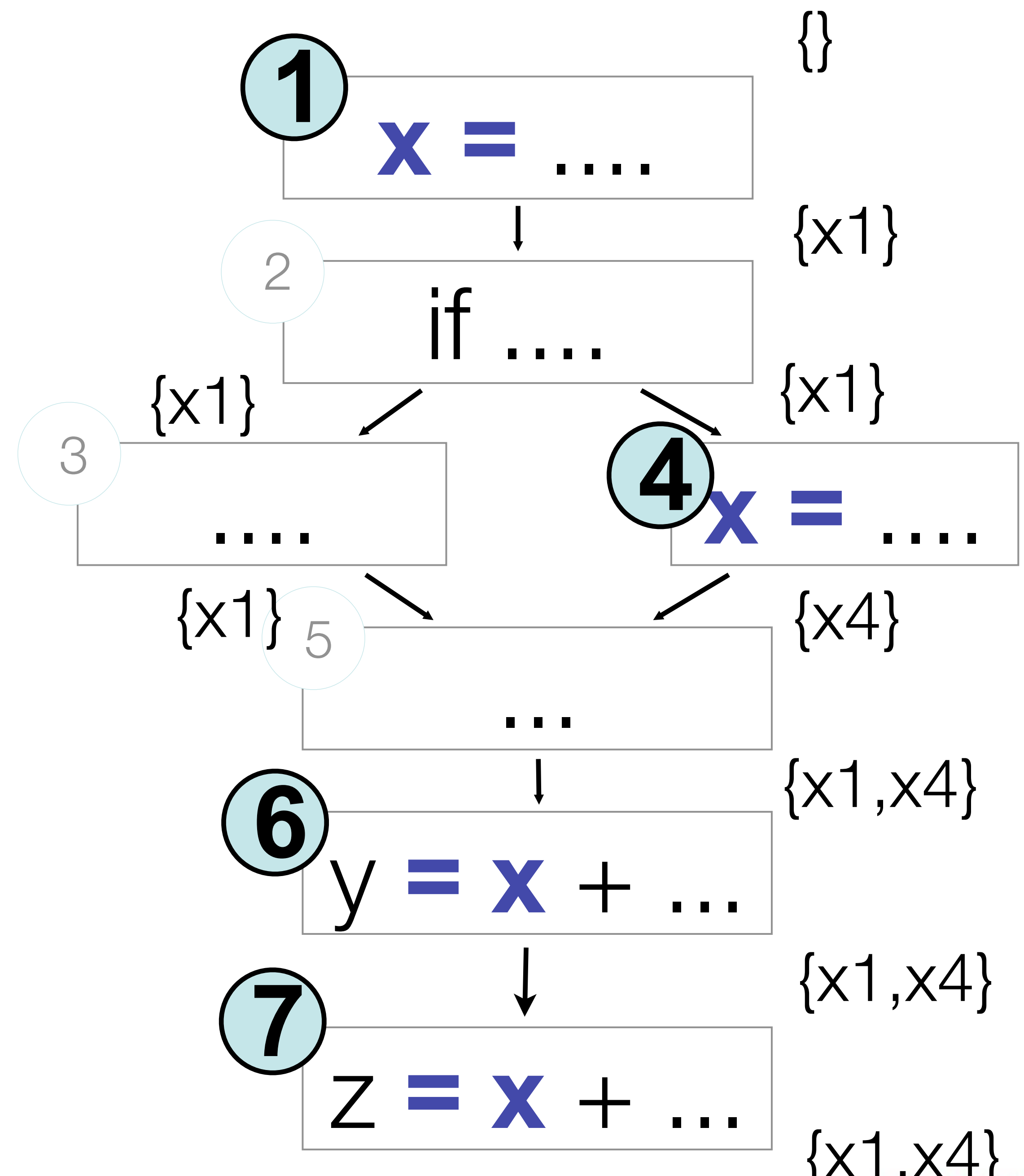
gen(n) = vn where var v is defined at node n

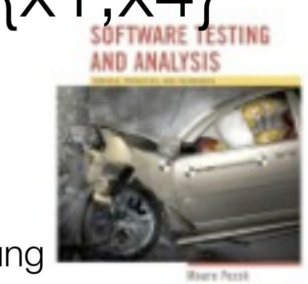
kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])





Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

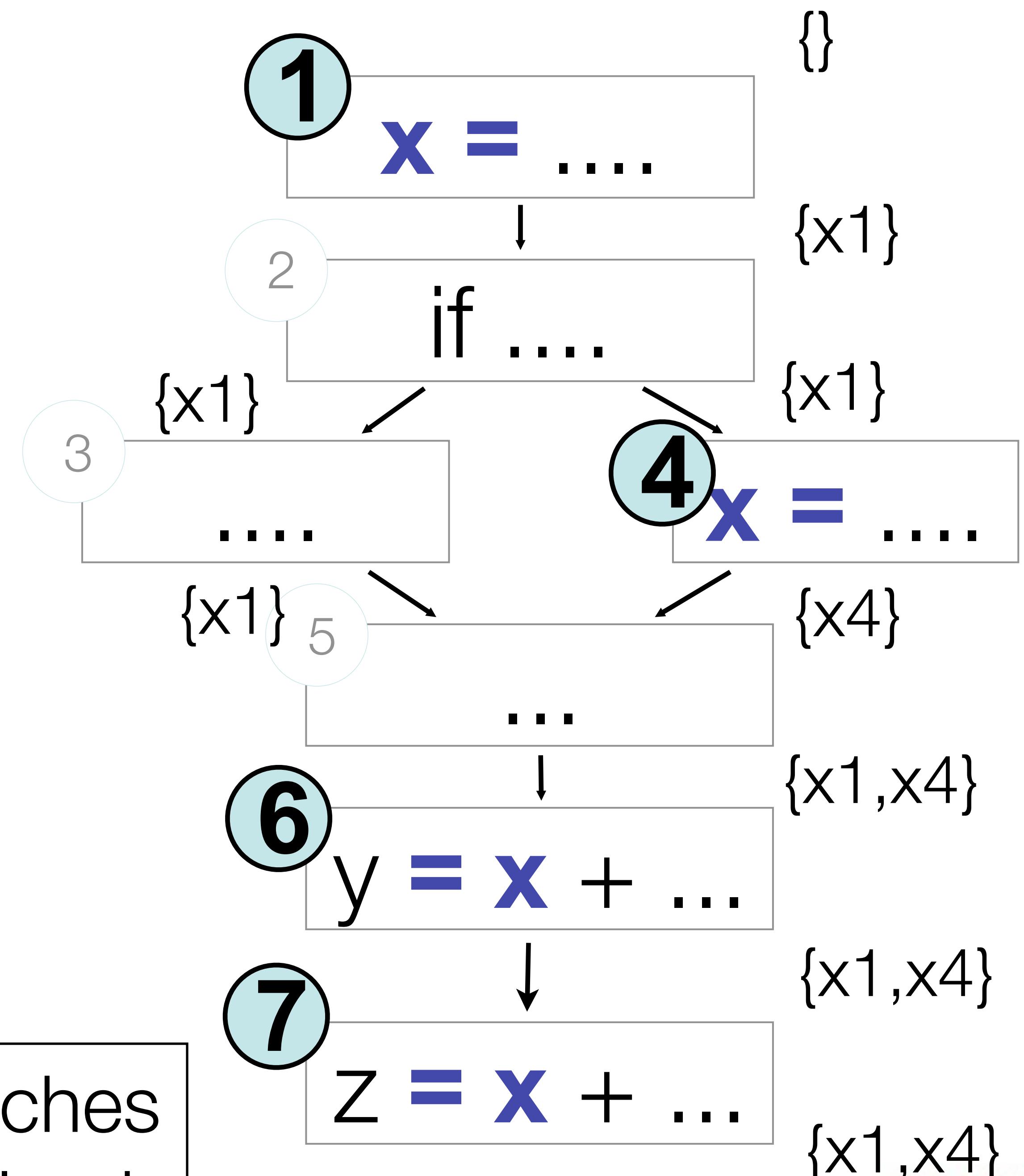
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

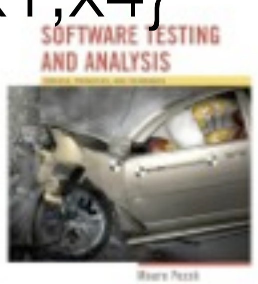
$\oplus = \cup$ (of sets)

in*[n] :=* ∪ *{* ***out****[m]* | *m* ***pred****(n) }*

out[n] := **gen**(n) ∪ (***in****[n]* - ***kill****[n])*



Every time a definition of variable x reaches a use of variable x we found a new DU pair



Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

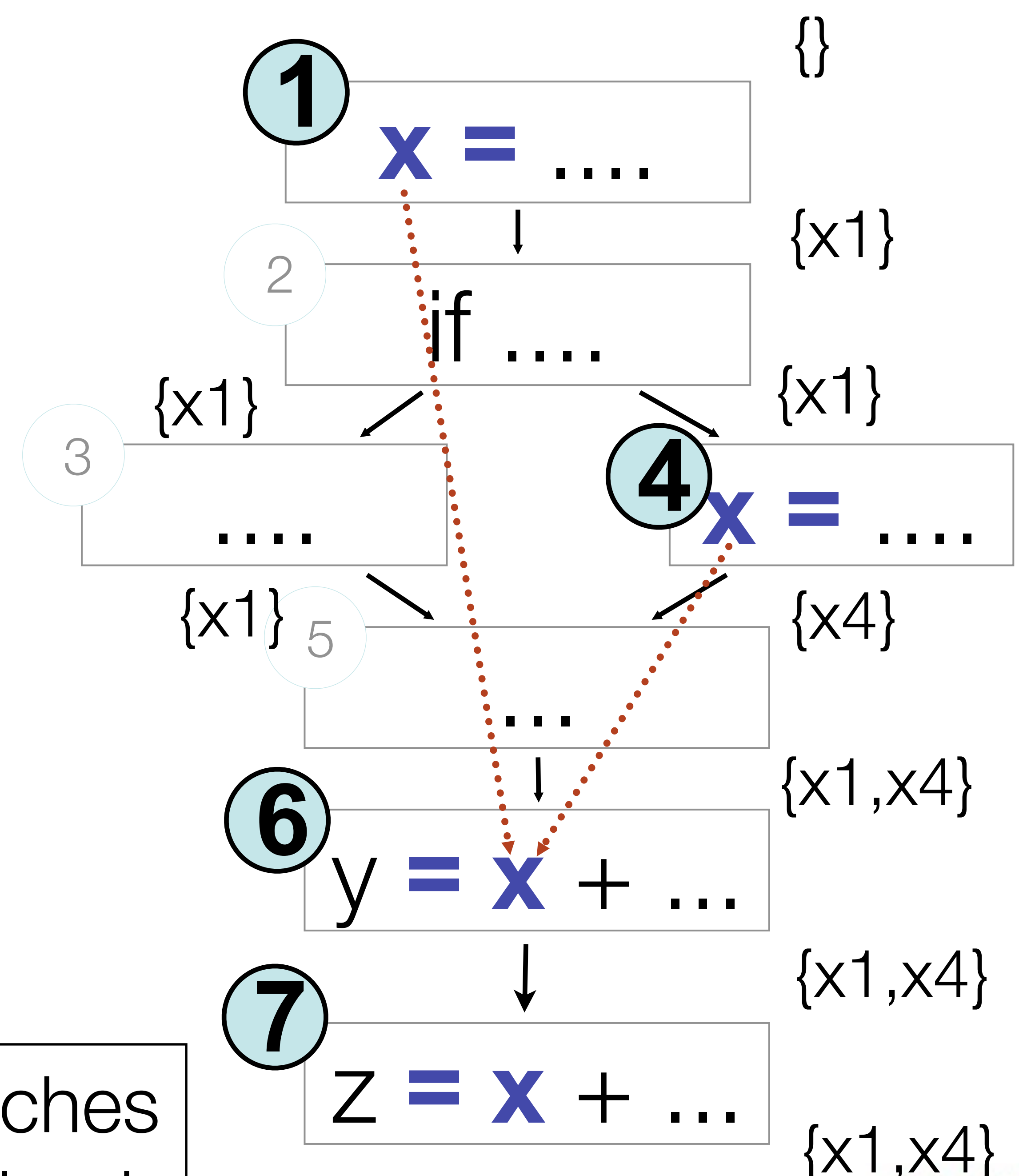
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])



Every time a definition of variable x reaches a use of variable x we found a new DU pair

Reaching definitions analysis

- It is a **forward may** analysis

in[n], out[n] = set of definitions of variables

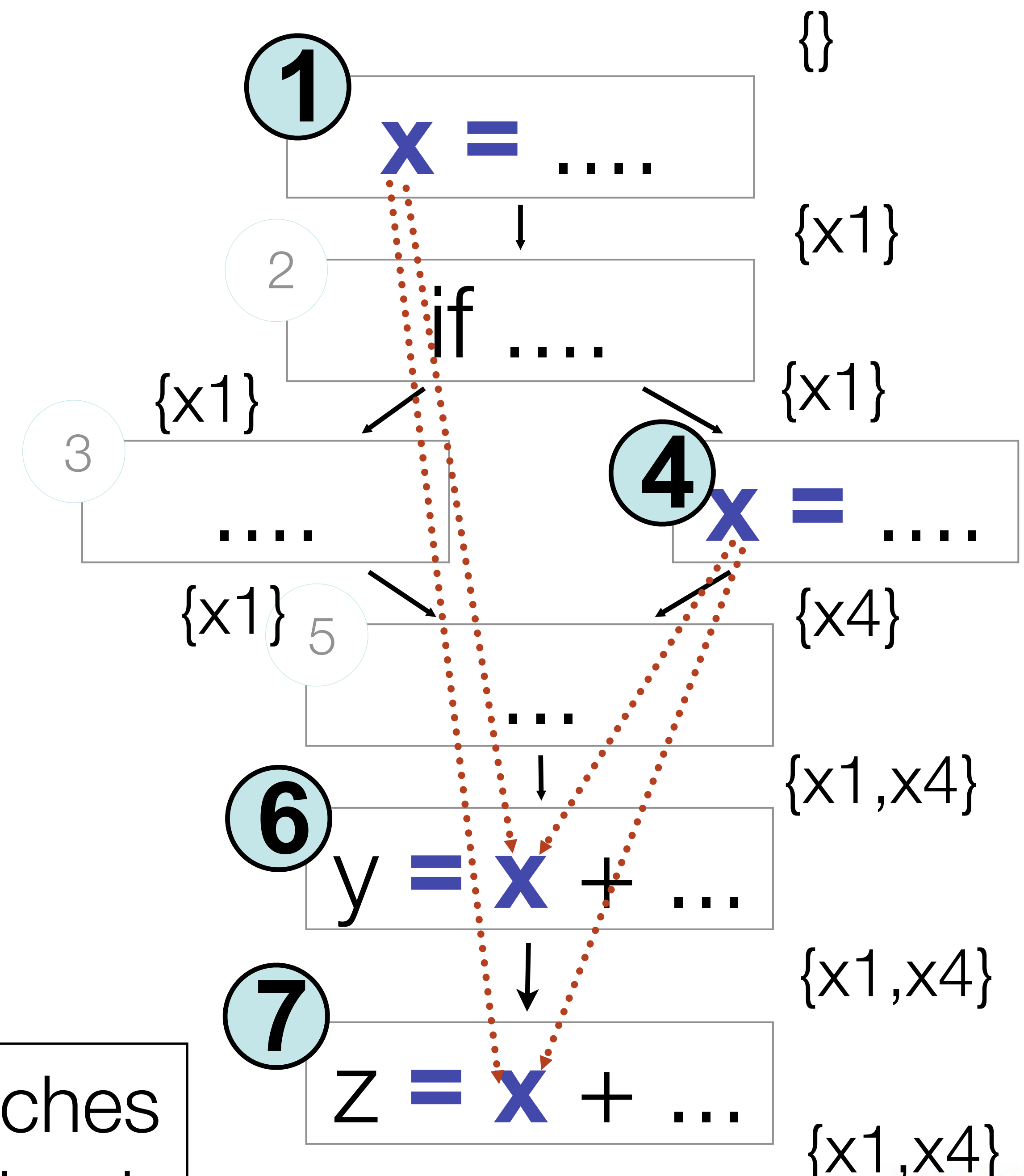
gen(n) = vn where var v is defined at node n

kill(n) = vx where var v is defined at node n and x

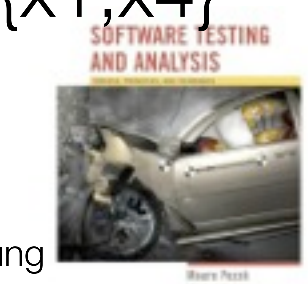
⊕ = ∪ (of sets)

in[*n*] := ∪ { ***out***[*m*] | *m* ***pred***(*n*) }

out[n] := **gen**(n) ∪ (***in***[*n*] - ***kill***[*n*])



Every time a definition of variable x reaches a use of variable x we found a new DU pair



Subsumes relation between data flow criteria