Code Based Testing

Path Testing

Structural Testing

- Complement of/to Functional Testing
- Based on Implementation
- Powerful mathematical formulation
  - program graph
  - define-use path
  - slice
- Basis for Coverage Metrics
  (identify gaps and redundancies)
- Usually done at the unit level
- Not very helpful to identify test cases
- Commercial tool support

Code Based Testing

- Statement Testing
- Multiple Testing
- Loop
- Path
- Modified Path (MaCabe path)
- Data flow
- Transaction Flow

Statement
 Cover at least once

- Assignment
- Input
- Output
- If-Then-Else (Predicate conditions)
- While Do
- Switch
- Function/Procedure Call Statement

A Variable declaration is not a statement
Since statement testing does not cover all statements, branch testing is used

Path Testing

- Paths derived from some graph construct
- When a test case executes, it traverses a path
- Huge number of paths implies some simplification needed
- Big Problem: infeasible paths
- Big Question: what kinds of faults are "associated" with what kinds of paths?
- By itself, path testing can lead to a false sense of security

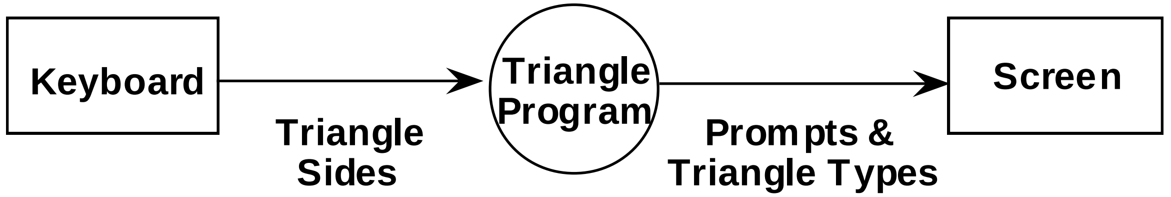
Program Graphs

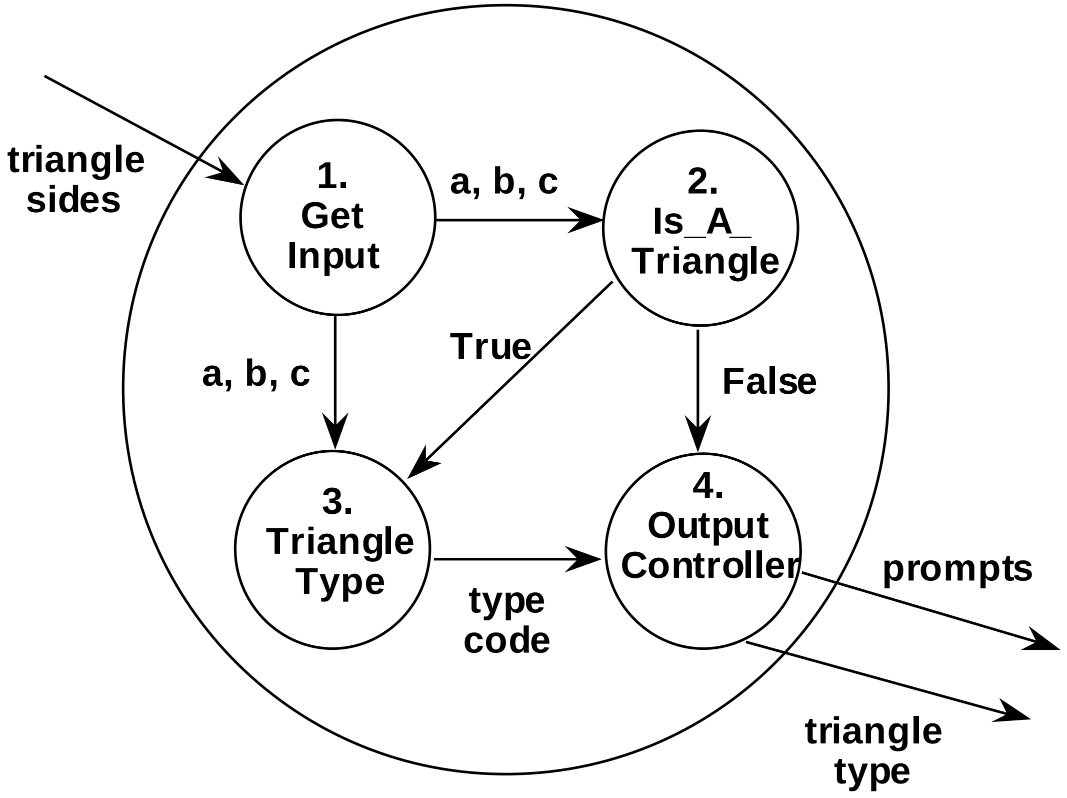
Given a program written in an imperative programming language, its program graph is a directed graph in which:

(Traditional Definition) nodes are program statements, and edges represent flow of control (there is an edge from node i to node j iff the statement corresponding to node j can be executed immediately after the statement corresponding to node i).

(Improved Definition) nodes are either entire statements or fragments of a statement, and edges represent flow of control (there is an edge from node i to node j iff the statement (fragment) corresponding to node j can be executed immediately after the statement or statement fragment corresponding to node i).

Triangle Program Specification





Data Dictionary

Triangle Sides : a + b + c

a, b, c are non-negative integers

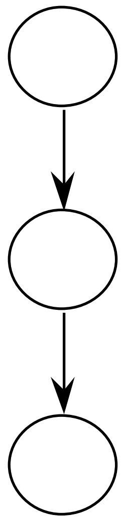
type code : equilateral | isosceles |
scalene | not a triangle

prompts : 'Enter three integers which
are sides of a triangle'

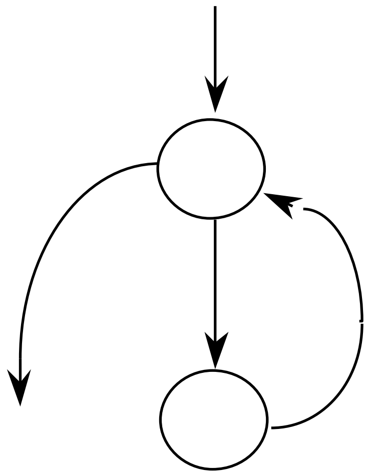
triangle type : 'Equilateral' | 'Isosceles' |
'Scalene' | 'Not a Triangl

Structured Programing Constructs

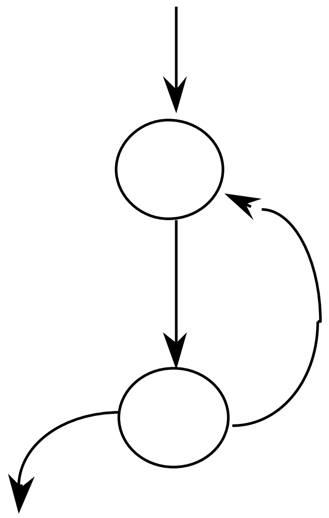
Sequence



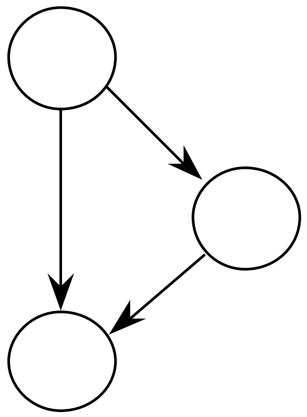
Pre-test Loop



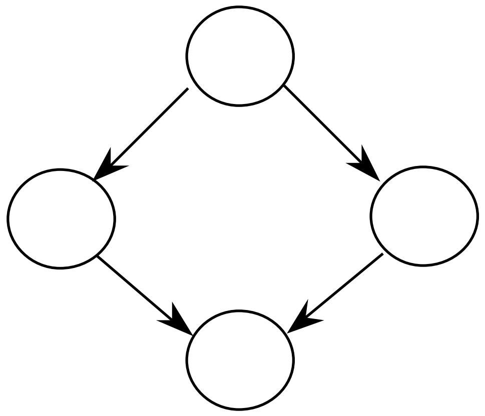
Post-test Loop



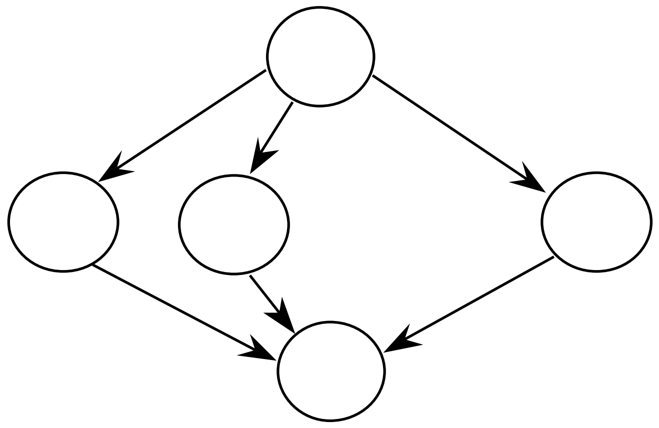
If-Then



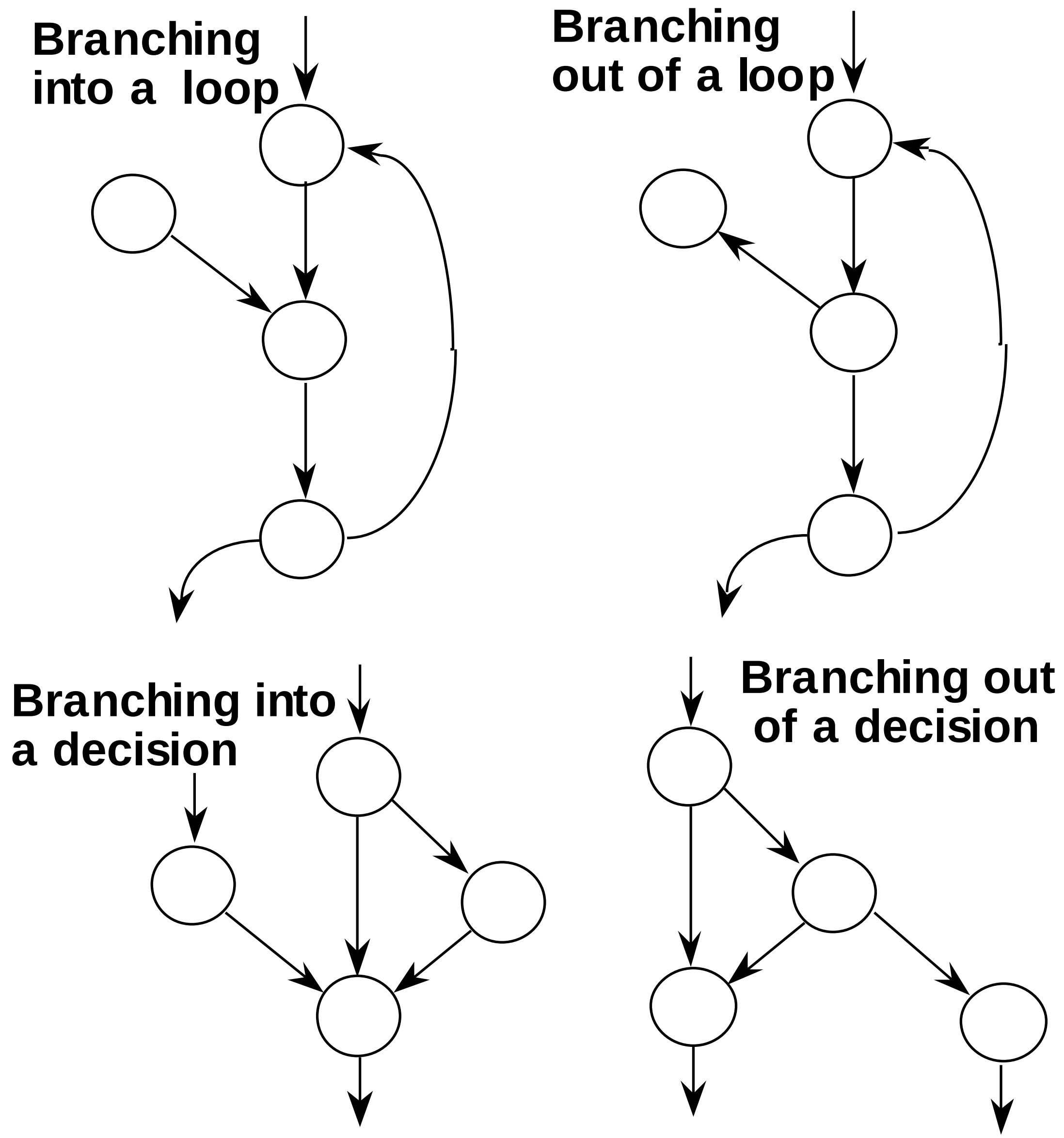
If-Then-Else



Case

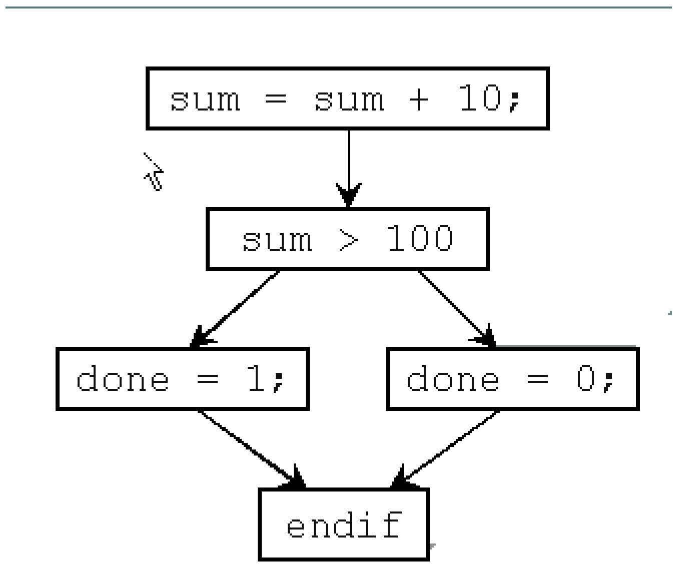


Violations of Structured Programming Precepts



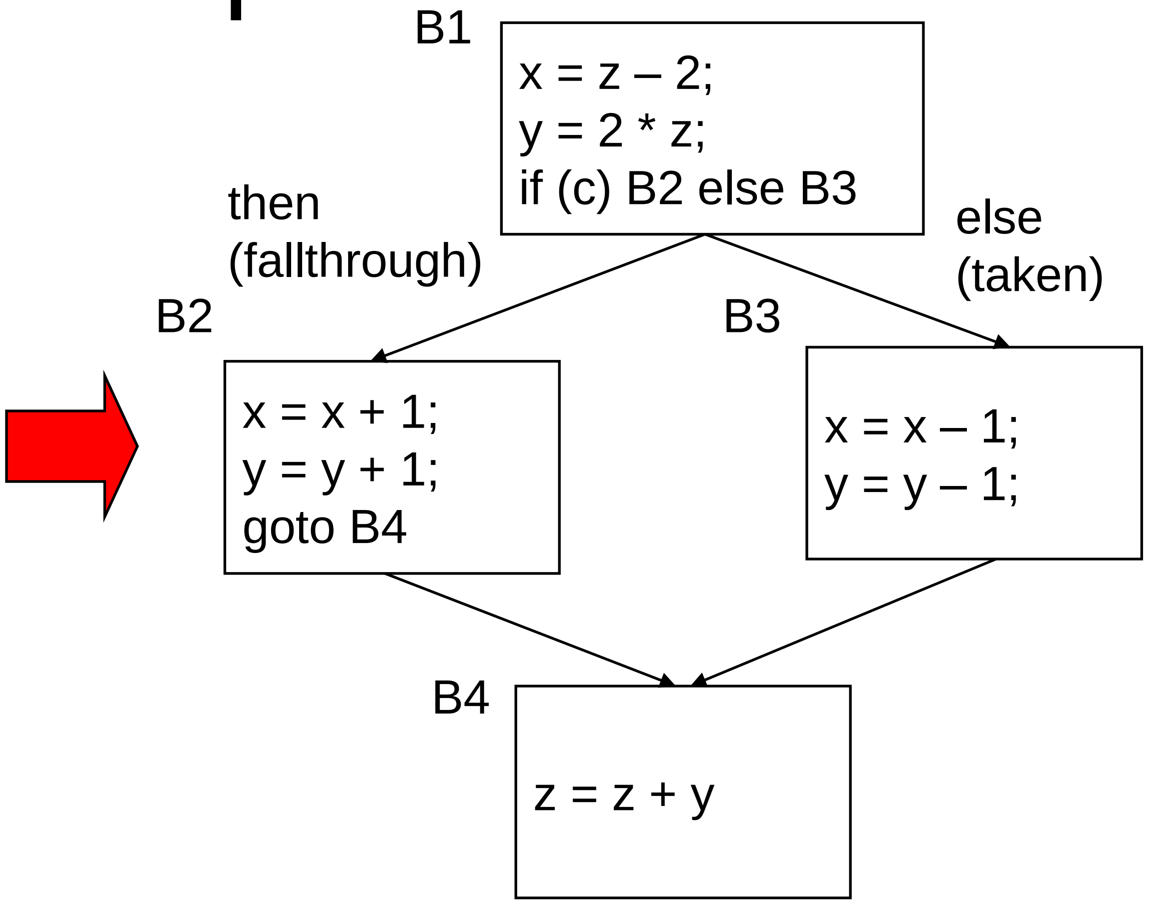
Simple example for CFG

- Sum=Sum+10; ①
- If (sum >100) ②
  Done=1; ③
- Else
  - Done=0; ④
- Endif ⑤



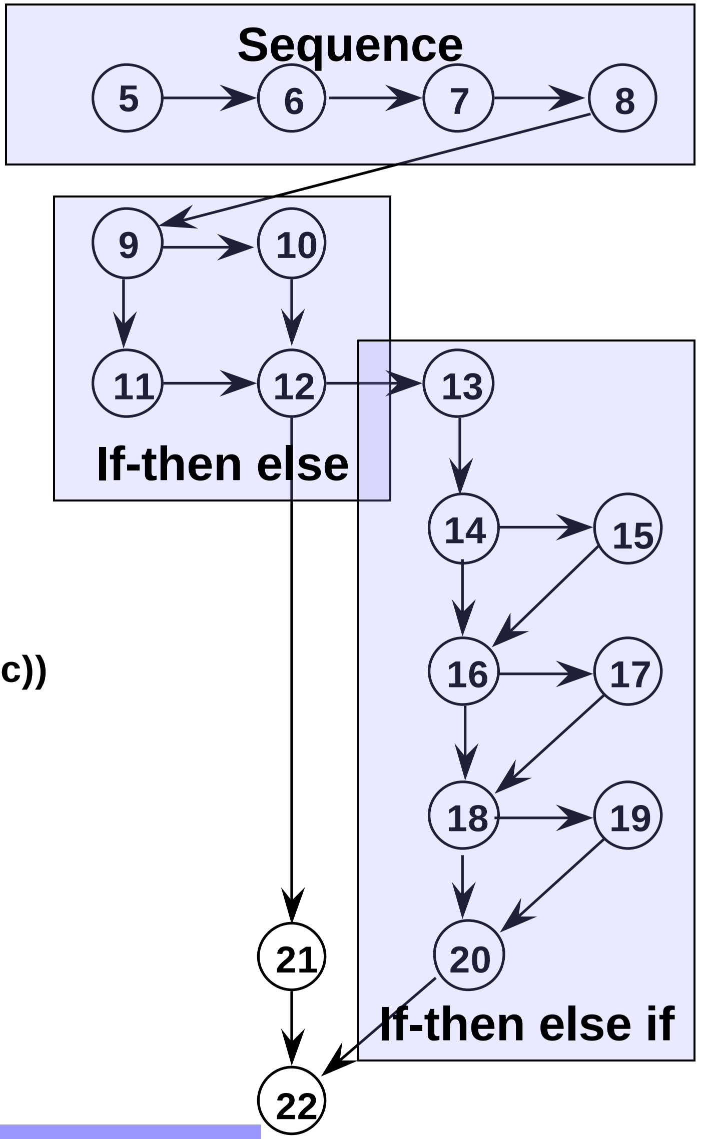
CFG Example

```
x = z – 2;
y = 2 * z;
if (c) {
    x = x + 1;
    y = y + 1;
}
else {
    x = x – 1;
    y = y – 1;
}
z = x + y
```



Triangle Program Graph

```
 1.  program triangle (input,output);
 2.  VAR    a, b, c         :   integer;
 3.          IsATriangle    : boolean;
 4.  begin
 5.   writeln('Enter three integers which');
 6.   writeln('are sides of a triangle ');
 7.   readln(a,b,c);
 8.   writeln('Side A is ',a,'Side B is ',b,'Side C is ',c);
 9.   IF (a < b + c) AND (b < a + c) AND (c < a + b)
10.       THEN IsATriangle := TRUE
11.       ELSE IsATriangle := FALSE ;
12.   IF IsATriangle THEN
13.     BEGIN
14.       IF (a = b) XOR (a = c) XOR (b = c) AND NOT((a=b) AND (a=c))
15.           THEN Writeln ('Triangle is Isosceles');
16.       IF (a = b) and (b = c)
17.           THEN Writeln ('Triangle is Equilateral');
18.       IF (a <> b) AND (a <> c) AND (b <> c)
19.          THEN Writeln ('Triangle is Scalene');
20.        END
21.     ELSE  WRITELN('Not a Triangle');
22.  end.
```



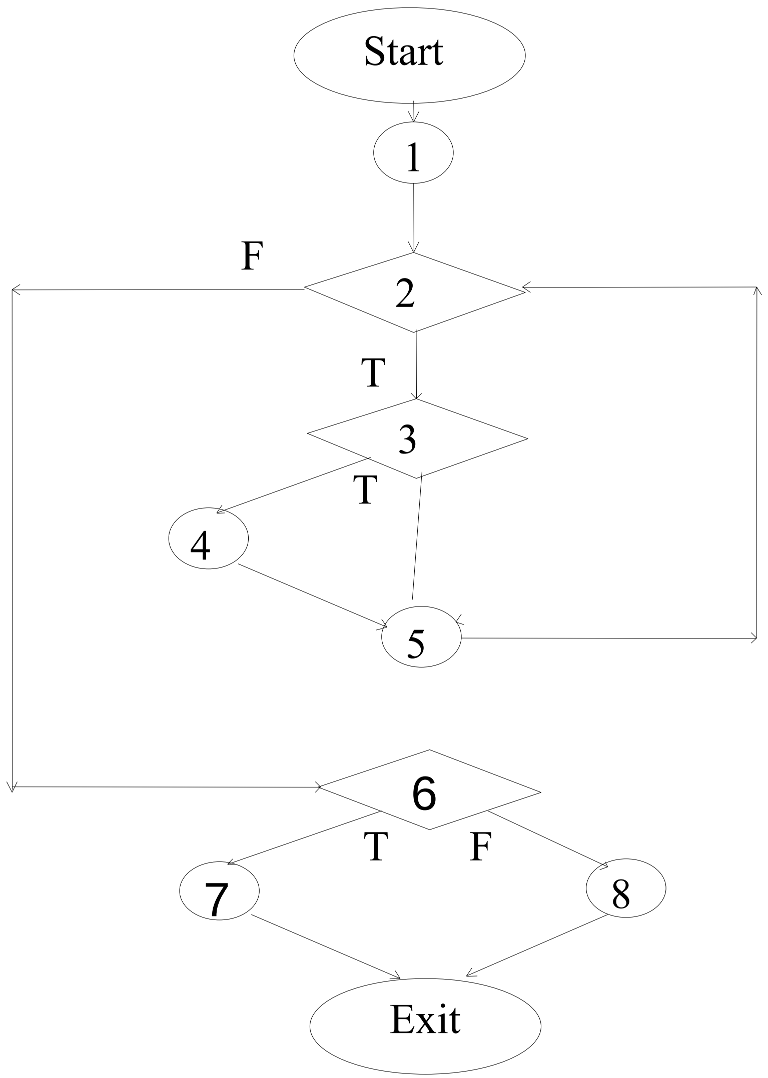
Acyclic graph 5 is source 22 is sink

Draw a Control flow graph

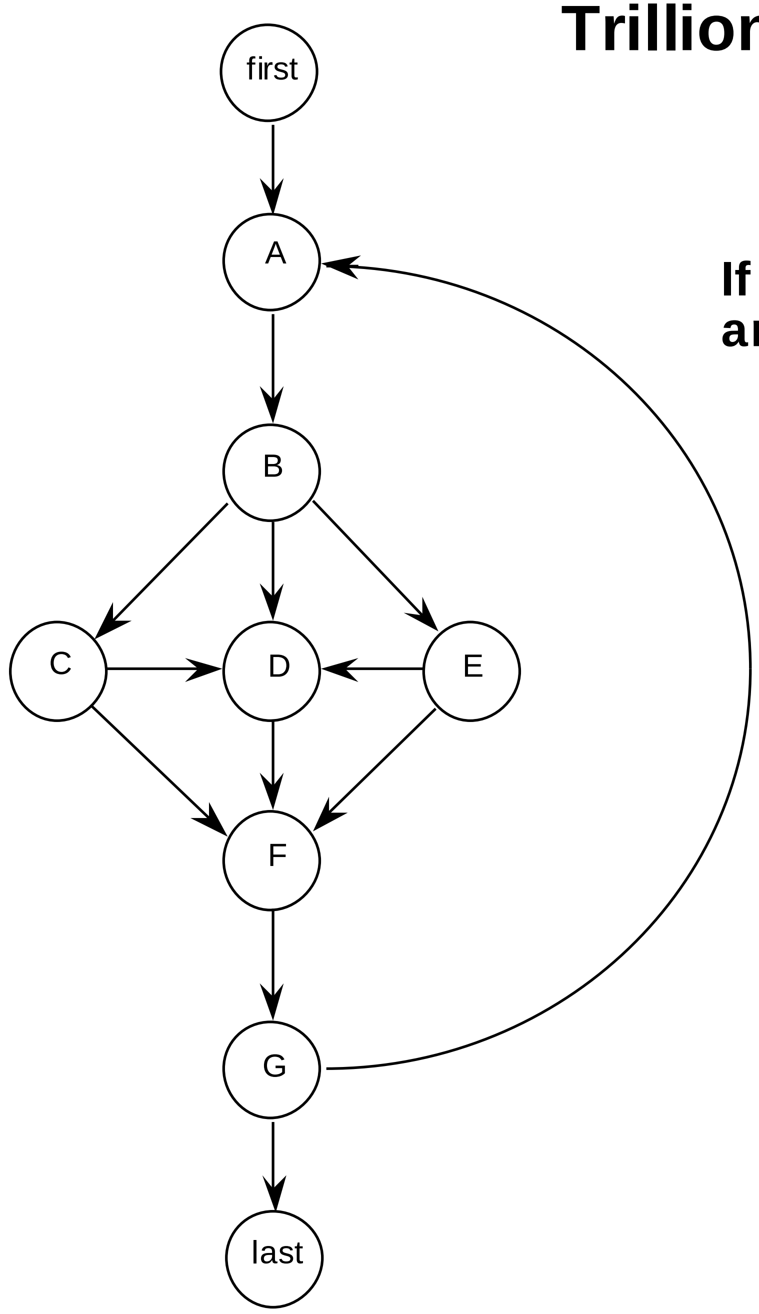
```
FindMean (FILE ScoreFile)
{   float SumOfScores = 0.0;
    int NumberOfScores = 0;
    float Mean=0.0; float Score;                          1
    Read(ScoreFile, Score);
2   while (! EOF(ScoreFile) {
  3   if (Score  > 0.0 ) {
            SumOfScores = SumOfScores + Score;            4
            NumberOfScores++;
            }

        Read(ScoreFile, Score);                           5
    }
    /* Compute the mean and print the result */
6   if (NumberOfScores > 0) {
          Mean = SumOfScores / NumberOfScores;            7
          printf(" The mean score is %f\n", Mean);
    } else
          printf ("No scores found in file\n");           8
}
```

Constructing the Logic Flow Diagram



Trillions of Paths



If the loop executes up to 18 times, there are 4.77 Trillion paths.

In this program, 5 paths lead from Node B to node F

Stephen R. Schach, *Software Engineering*, (2nd edition) Richard D. Irwin, Inc. and Aksen Associates, Inc. 1993

DD-Paths

A DD-Path (decision-to-decision) is a chain in a program graph such that

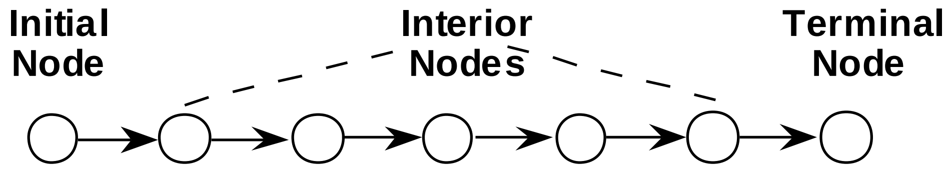
Case 1: it consists of a single node with indeg = 0,

Case 2: it consists of a single node with outdeg = 0,

Case 3: it consists of a single node with indeg •2 or outdeg •2,

Case 4: it consists of a single node with indeg = 1 and outdeg = 1,

Case 5: it is a maximal chain of length •1.



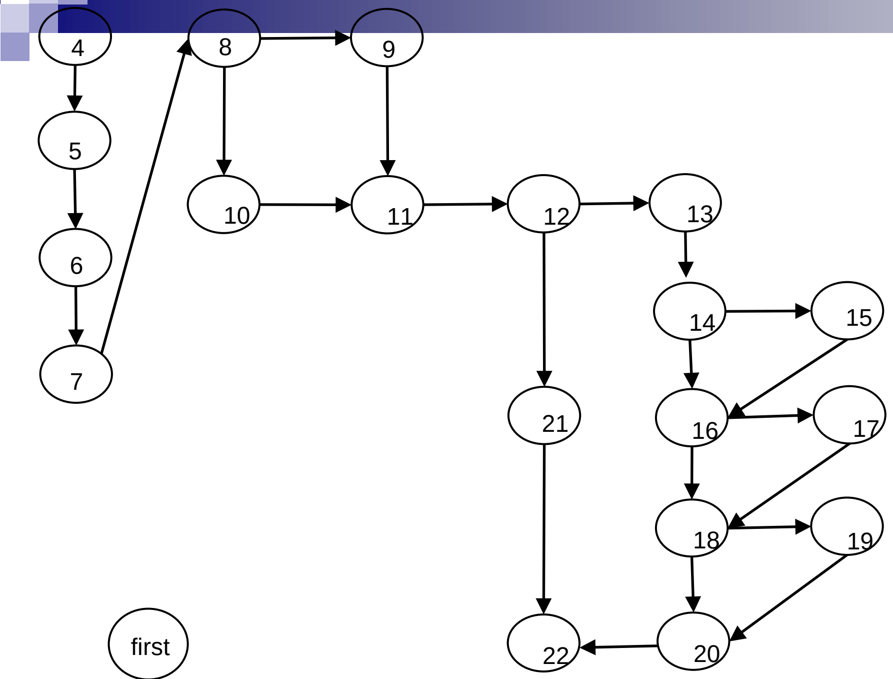
a 2-connected chain

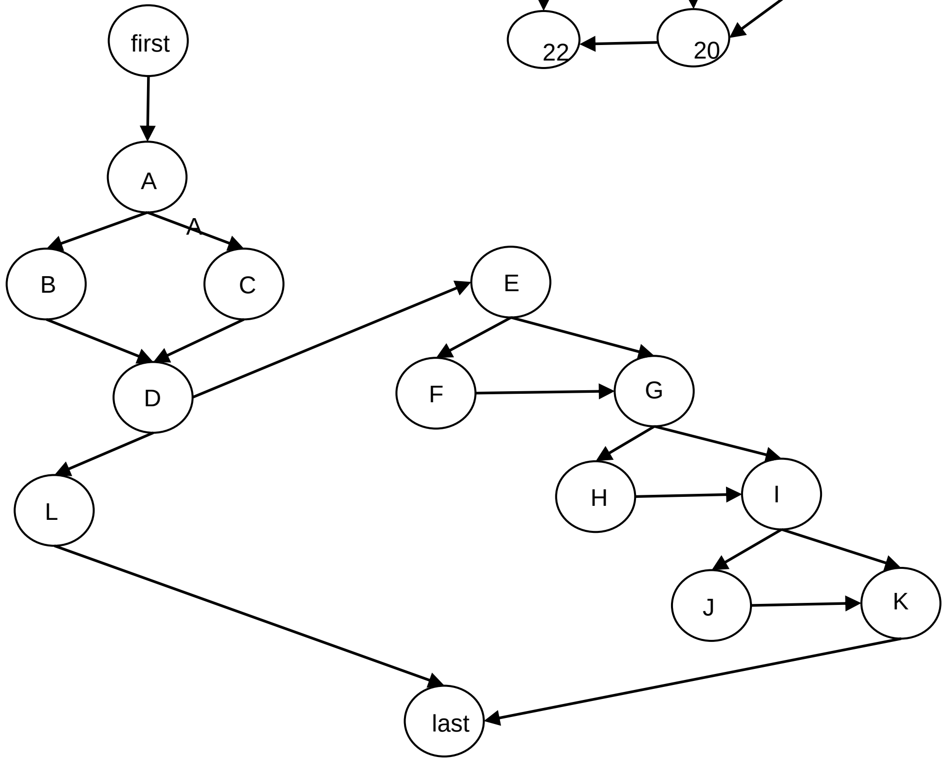
Works well for 2nd Generation language like FORTRAN

DD-Path Graph

- Given a program written in an imperative language, its *DD-Path graph* is a labeled directed graph, in which nodes are DD-Paths pf its program graph, and edges represent control flow between successor DD-Paths.

- In this respect, a DD-Path is a condensation graph. For example 2-connected program graph nodes are collapsed to a single DD-Path graph node.





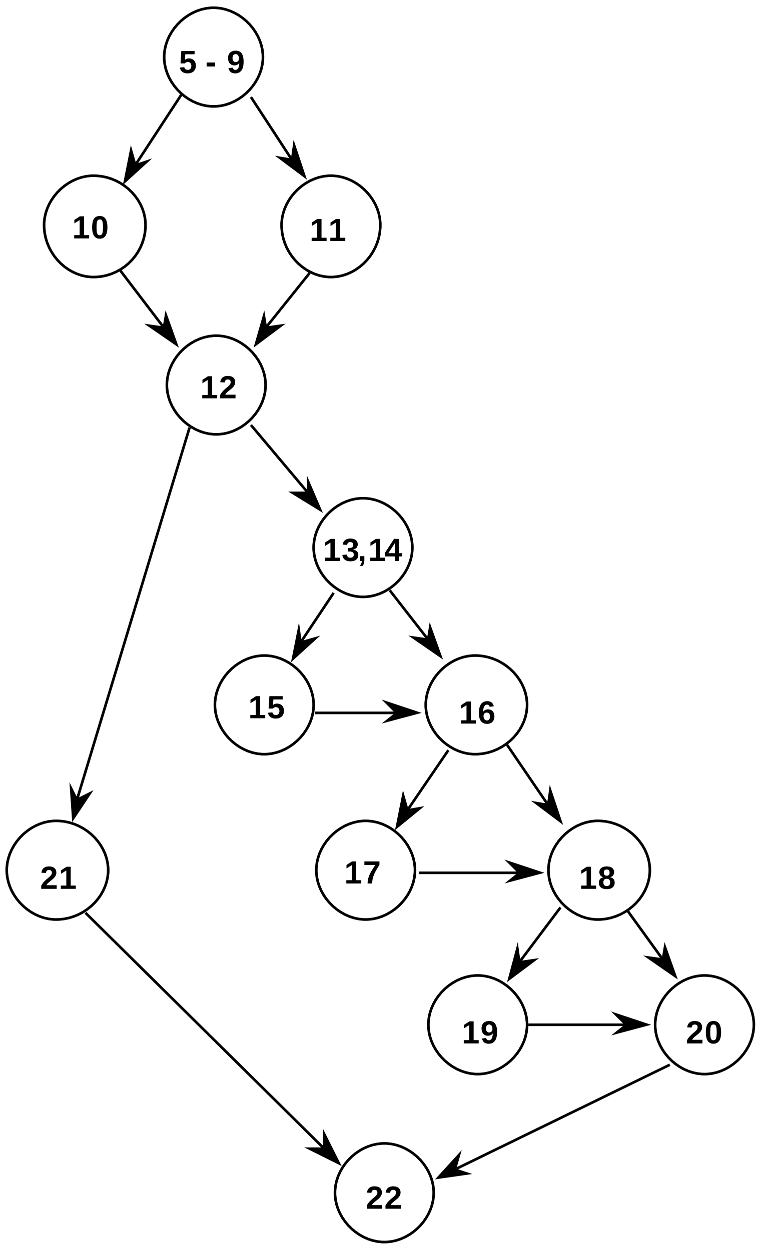
Program Graph Nodes	DD-Path Name	Case #
4	first	1
5-8	A	5
9	B	4
10	C	4
11	D	3
12-14	E	5
15	F	4
16	G	3
17	H	4
18	I	3
19	J	4
20	K	3
21	L	4
22	last	2

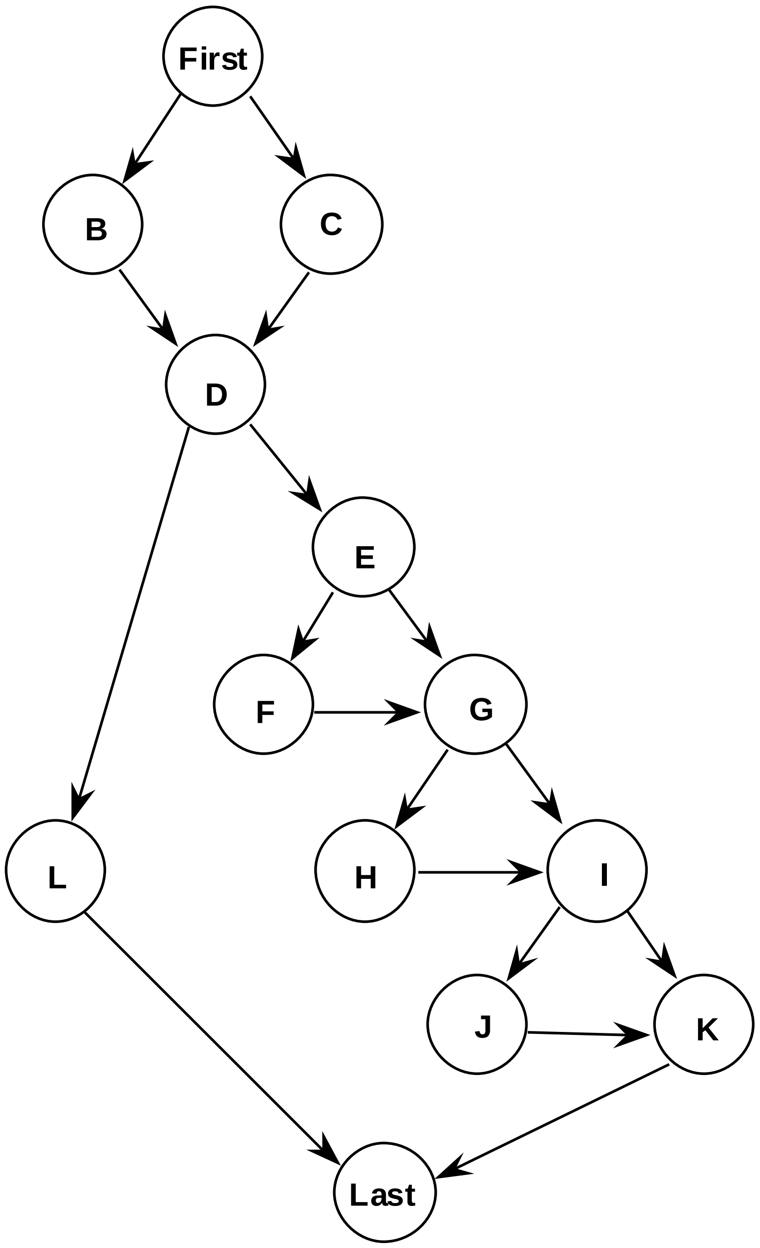
DD-Path Graph

Given a program written in an imperative language, its DD-Path graph is the directed graph in which nodes are DD-Paths of its program graph, and edges represent control flow between successor DD-Paths.

- **a form of condensation graph**
- **2-connected components are collapsed into individual node**
- **single node DD-Paths (corresponding to Cases 1 - 4) preserve the convention that a statement fragment is in exactly one DD-Path**

DD-Path Graph

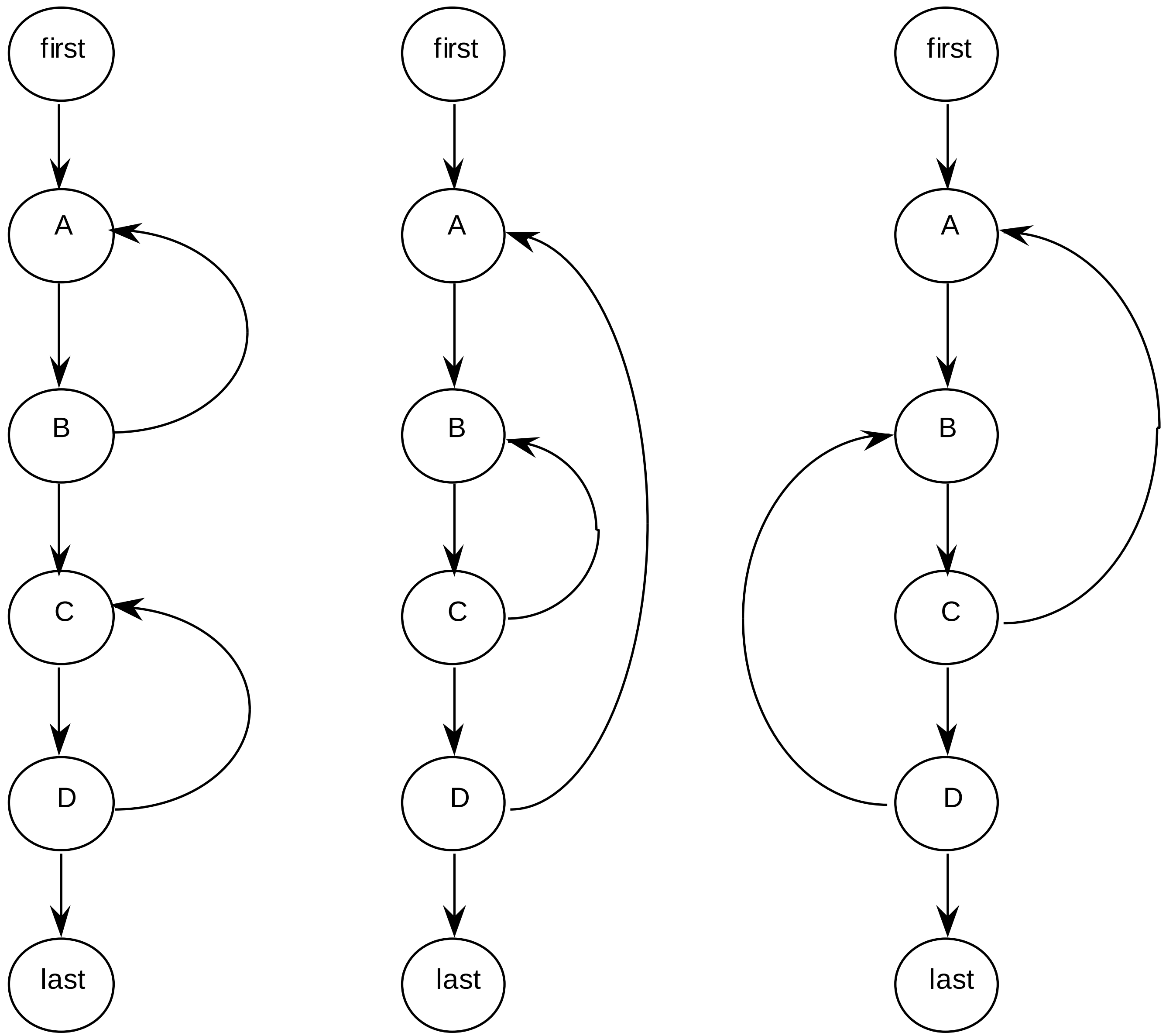




Structural Test Coverage Metrics

Metric	Description of Coverage
C_0	Every statement
C_1	Every DD-Path (predicate outcome)
C_{1p}	Every predicate to each outcome
C_2	C_1 coverage + loop coverage
Cd	C_1 coverage + Every dependent pair of DD-Paths
C_{MCC}	Multiple condition coverage
C_{ik}	Every program path that contains up to k repetitions of a loop (usually k = 2)
C_{stat}	"Statistically significant" fraction of paths
$C_{\bullet}$	All possible execution paths

Concatenated, Nested, and Knotted Loops



Test Coverage Metrics

- The motivation of using DD-paths is that they enable very precise descriptions of test coverage.
- In our quest to identify gaps and redundancy in our test cases as these are used to exercise (test) different aspects of a program we use formal models of the program structure to reason about testing effectiveness.
- Test coverage metrics are a device to measure the extend to which a set of test cases covers a program.

Test Coverage Metrics

Metric	Description of Coverage
C_0	Every Statement
C_1	Every DD-Path
C_1P	Every predicate to each outcome
C_2	C_1 Coverage + loop coverage
C_d	C_1 Coverage + every dependent pair of DD-Paths
C_{MCC}	Multiple condition coverage
C_ik	Every program path that contains up to k repetitions of a loop (usually k=2)
C_{stat}	"Statistically significant" fraction of paths
C_∞	All possible execution paths

Statement and Predicate Coverage Testing

- Statement coverage based testing aims to devise test cases that collectively exercise all statements in a program.

- Predicate coverage (or branch coverage, or decision coverage) based testing aims to devise test cases that evaluate each simple predicate of the program to True and False. Here the term simple predicate refers to either a single predicate or a compound Boolean expression that is considered as a single unit that evaluates to True or False. **This amounts to traversing every edge in the DD-Path graph.**

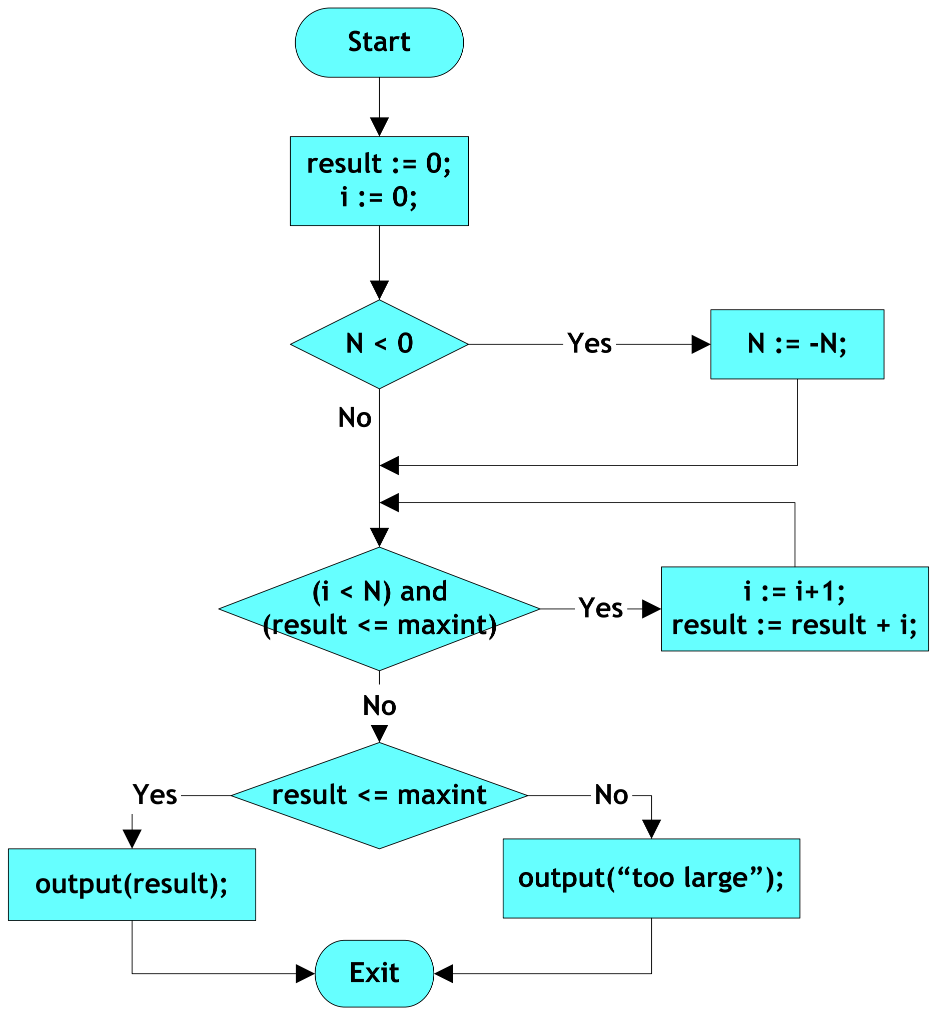
- For example in predicate coverage for the condition

  if(A or B) then C we could consider the test cases A=True, B= False (true case), and A=False, B=False (false case). Note if the program was encoded as *if(A) then C* we would not detect any problem.

Statement coverage

- Execute (exercise) every statement of a program
  - Generate a set of test cases such that each statement of the program is executed at least one
- Weakest white-box criterion
- Supported by many commercial and freeware tools (test coverage or code coverage tools)
  - Standard Unix tool: tcov
  - A listing indicates how often each statement was executed and the percentage of statements executed
- Note: in case of unreachable statements, statement coverage is not possible

Example : statement coverage

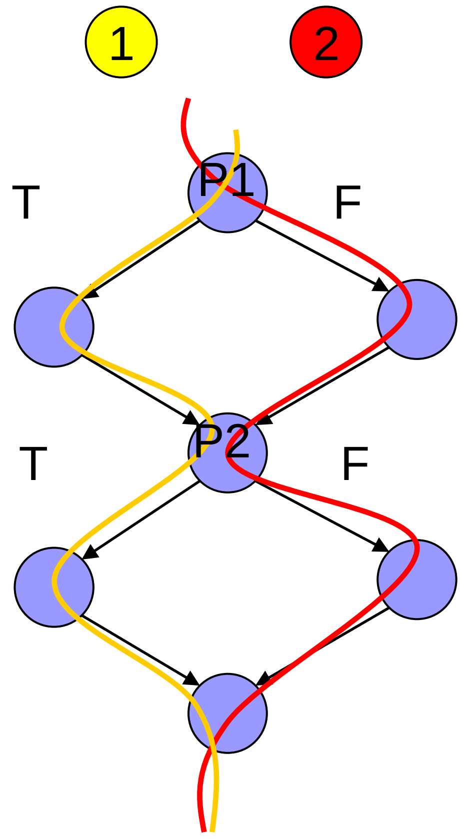


Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

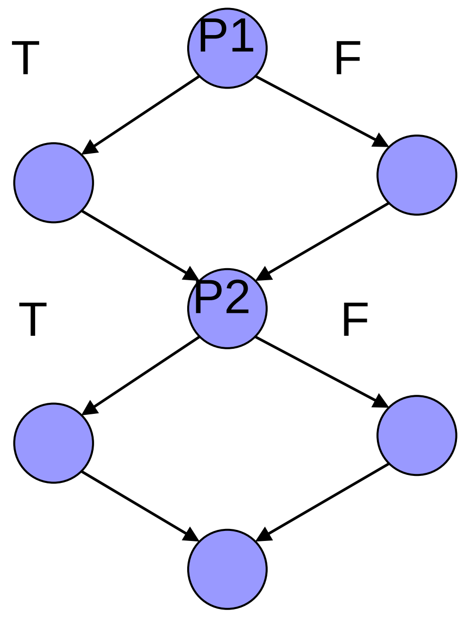
DD-Path Graph Edge Coverage C1

Here a T, T and F, F combination will suffice to have DD-Path Graph edge coverage or Predicate coverage C1



DD-Path Coverage Testing C_1P

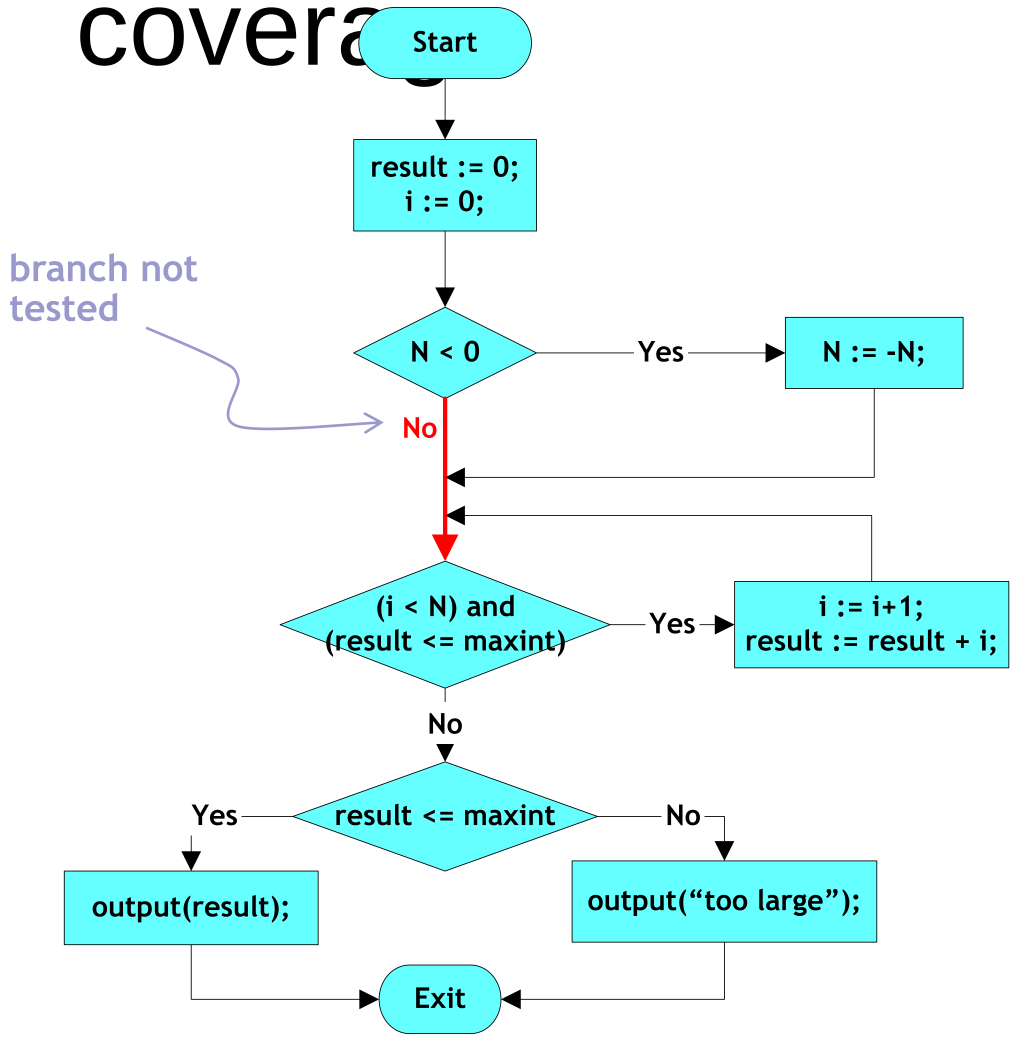
- This is the same as the C_1 but now we must consider test cases that exercise all all possible outcomes of the choices T,T, T,F, F,T, F,F for the predicates P1, and P2 respectively, in the DD-Path graph.



Branch (or decision) coverage

- Branch coverage == decision coverage
- Execute every branch of a program :
  each possible outcome of each decision occurs at least once
- Example:
  - simple decision: IF b THEN s1 ELSE s2
  - multiple decision:
    ```
    CASE   x   OF
    1 :  ….
    2 :  ….
    3 :  ….
    ```
- Stronger than statement coverage
  - IF THEN without ELSE – if the condition is always true all the statements are executed, but branch coverage is not achieved

Example : branch (or decision) coverage



Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

are not sufficient for branch (edge) coverage!

Take:

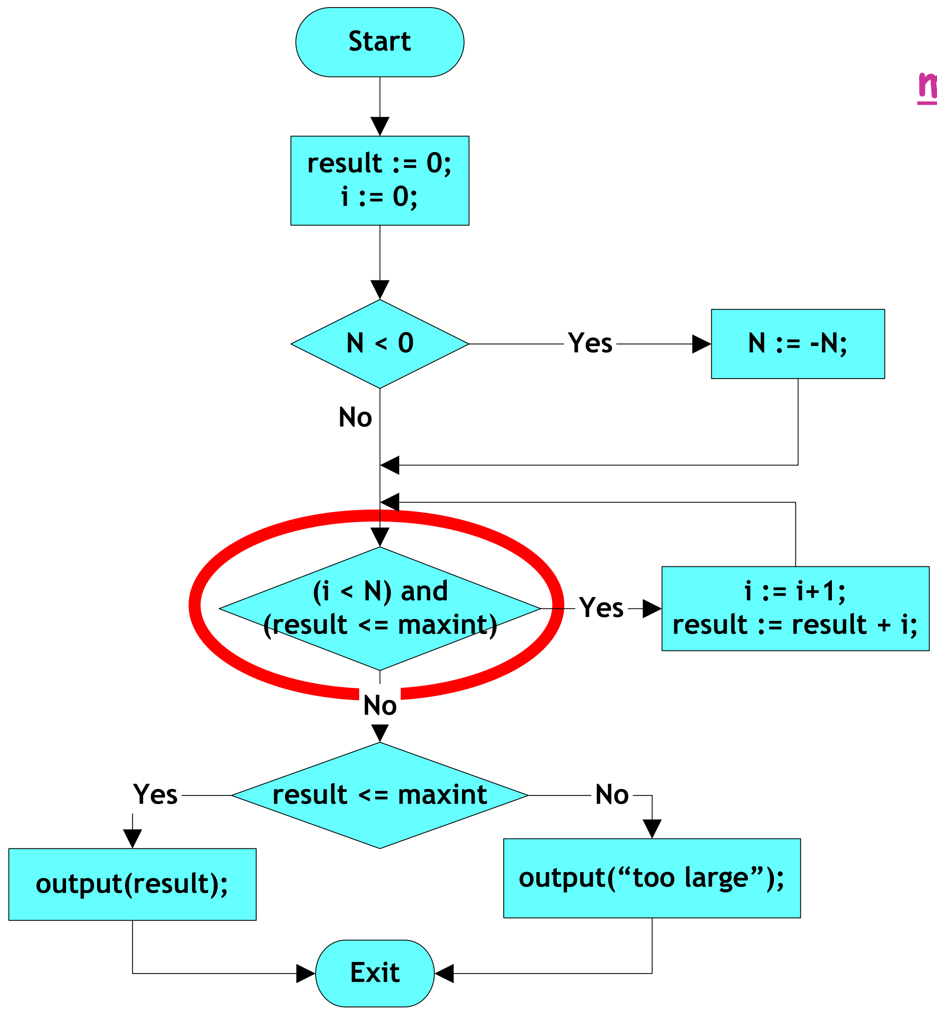
inputs		outputs
maxint	N	result
10	3	6
0	-1	too large

for complete branch (edge) coverage

Condition coverage

- Design test cases such that each possible outcome of each condition in a composite condition occurs at least once
- Example:
  - decision (i < N) AND (result <= maxint) consists of two conditions : (i < N) and (result <= maxint) test cases should be designed such that each gets value true and false at least once
- Last test cases of previous slides already guarantee condition (and branch) coverage

Example : Branch and condition (or condition / decision) coverage



Test cases:

maxint	N	i	result	i<N	result<=maxint
-1	1	0	0	true	false
1	0	0	0	false	true

give condition coverage
for all conditions

But don't preserve
branch coverage

⇓

always take care that
condition coverage
preserves branch coverage :
branch and condition coverage

Multiple Condition Coverage Testing

- Now if we consider that the predicates P1 is a compound predicate (i.e. (A or B)) then Multiple Condition Coverage Testing requires that each possible combination of inputs be tested for each decision.

- Example: “if (A or B)” requires 4 test cases:

  A = True, B = True

  A = True, B = False

  A = False, B = True

  A = False, B = False
- The problem: For n conditions, 2^n test cases are needed, and this grows exponentially with n

Multiple condition coverage

- Design test cases for each combination of conditions
- Example:
  - (i < N) (result <= maxint)

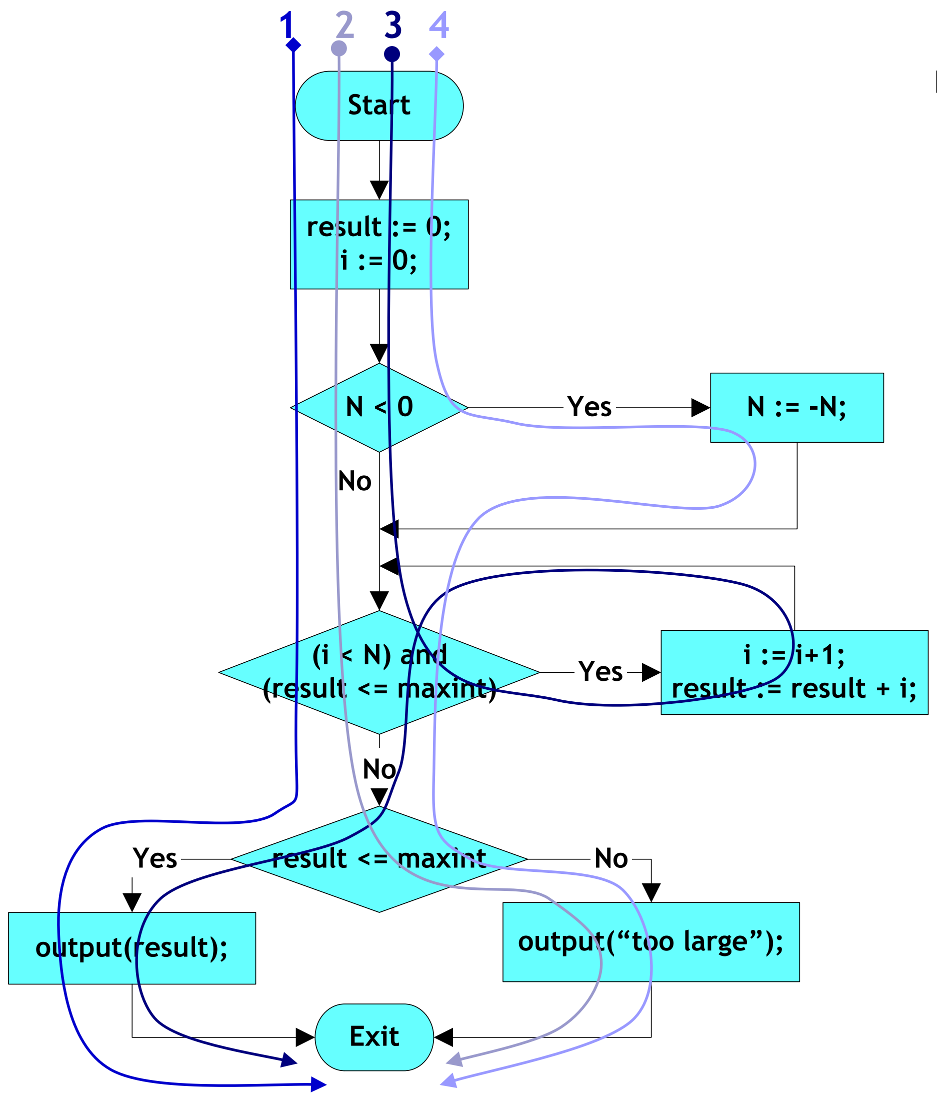
(i < N)	(result <= maxint)
false	false
false	true
true	false
true	true

- Implies branch-, condition-, branch and condition, modified branch/condition coverage
- But : exponential blow-up ($2^{\text{number of conditions}}$)
- Again : some combinations may be infeasible

Independent path (or basis path) coverage

- Obtain a maximal set of linearly independent paths (also called a basis of independent paths)
  - If each path is represented as a vector with the number of times that each edge of the control flow graph is traversed, the paths are linearly independent if it is not possible to express one of them as a liner combination of the others
- Generate a test case for each independent path
- The number of linearly independent paths is given by the McCabe's *cyclomatic complexity* of the program
  - Number of edges - Number of nodes + 2 in the control flow graph
  - Measures the structural complexity of the program
- Problem: some paths may be impossible to execute
- Also called structured testing (see McCabe for details)
- McCabe's argument: this approach produces a number of test cases that is proportional to the complexity of the program (as measured by the cyclomatic complexity), which, in turn, is related to the number of defects expected
- More information:
  - http://www.mccabe.com/iq_research_metrics.htm
  - "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", Arthur H. Watson, Thomas J. McCabe, NIST Special Publication 500-235

Example: Independent path coverage



number of (linearly) independent paths
= cyclomatic complexity
= number of edges - number of nodes + 2
= 12 - 10 + 2
= 4

Test cases

Path	inputs		outputs
	maxint	N	result
1	1	0	0
2	-1	0	too large
3	-1	-1	too large
4	10	1	1

Dependent DD-Path Pairs Coverage Testing C_d

- In simple C_1 coverage criterion we are interested simply to traverse all edges in the DD-Path graph.

- If we enhance this coverage criterion by ensuring that we also traverse dependent pairs of DD-Paths also we may have the chance of revealing more errors that are based on data flow dependencies.

- More specifically, two DD-Paths are said to be dependent iff there is a define/reference relationship between these DD-Paths, in which a variable is defined (receives a value) in one DD-Path and is referenced in the other.

- In C_d testing we are interested on covering all edges of the DD-Path graph *and* all dependent DD-Path pairs.

Loop Coverage

- The simple view of loop testing coverage is that we must devise test cases that exercise the two possible outcomes of the decision of a loop condition that is one to traverse the loop and the other to exit (or not enter) the loop.

- An extension would be to consider a modified boundary value analysis approach where the loop index is given a minimum, minimum +, a nominal, a maximum -, and a maximum value or even robustness testing.

- Once a loop is tested, then the tester can collapse it into a single node to simplify the graph for the next loop tests. In the case of nested loops we start with the inner most loop and we proceed outwards.

- If loops are knotted then we must apply data flow analysis testing techniques.

Statistically Significant Path Coverage Testing

- Exhaustive testing of software is not practical because variable input values and variable sequencing of inputs result in too many possible combinations to test.
- NIST developed techniques for applying statistical methods to derive sample test cases would address how to select the best sample of test cases and would provide a statistical level of confidence or probability that a program implements its functional specification correctly.
- The goal of statistically significant coverage is to develop methods for software testing based on statistical methods, such as Multivariable Testing, Design of Experiments, and Markov Chain usage models, and to develop methods for software testing based on statistical measures and confidence levels.

Source: http://www.itl.nist.gov/div897/ctg/stat/mar98ir.pdf

Basis Path Testing

- We can apply McCabe's Cyclomatic Complexity metric
  - gives an upper bound on number of test cases to ensure edge coverage is satisfied.
  - in practice, it is usually the "lower bound" of the number of test cases due to the presence of loops.

Basis Path Testing - Motivation

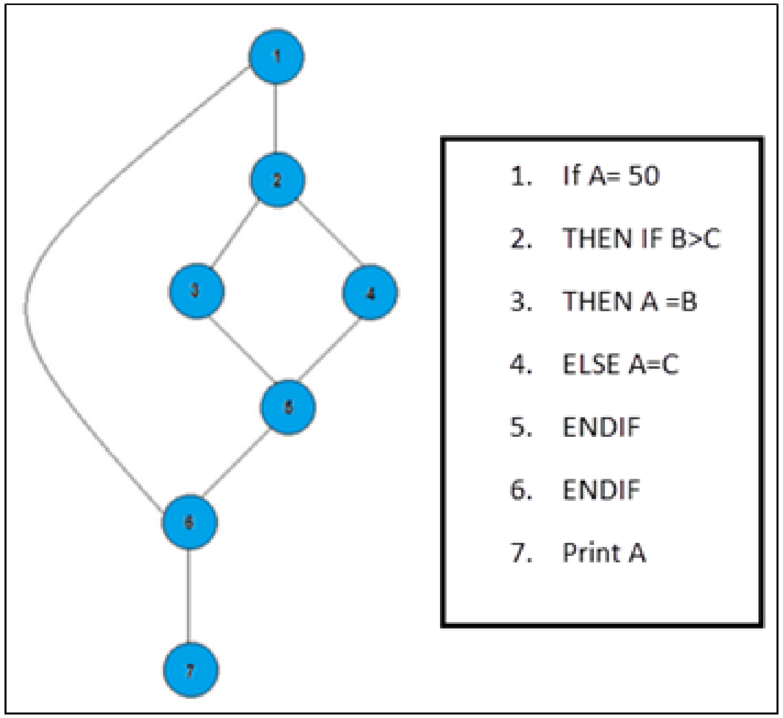
- If we consider the paths in a program graph (or DD-Graph) to form a vector space *V*, we are interested to devise a subset of *V* say *B* that captures the essence of V; that is every element of *V* can be represented as a linear combination of elements of *B.* Addition of paths means that one path is followed by another and multiplication of a number by a path denotes the repetition of a path.

- If such a vector space *B* contains linearly independent paths and forms a "*basis*" for *V* then it certainly captures the essence of *V*.

McCabe Algorithm to Determine Basis Paths

- The algorithm is straightforward:
  - The method begins with the selection of a “baseline path”, which should correspond to a normal execution of a program (from start node to end node, and has as many decisions as possible).
  - The algorithm proceeds by retracing the paths visited and flipping the conditions one at a time.
  - The process repeats up to the point all flips have been considered.
- The objective is to generate test cases that exercise these “basis” paths.

The objective behind basis path in software testing is that it defines the number of independent paths, thus the number of test cases needed can be defined explicitly (maximizes the coverage of each test case).



```
1.  If A= 50
2.  THEN IF B>C
3.  THEN A =B
4.  ELSE A=C
5.  ENDIF
6.  ENDIF
7.  Print A
```

In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output,

Path 1: 1,2,3,5,6, 7
Path 2: 1,2,4,5,6, 7
Path 3: 1, 6, 7

- **Steps for Basis Path testing**

- The basic steps involved in basis path testing include
  - Draw a control graph (to determine different program paths)
  - Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
  - Find a basis set of paths
  - Generate test cases to exercise each path

- **Advantages of Basic Path Testing**

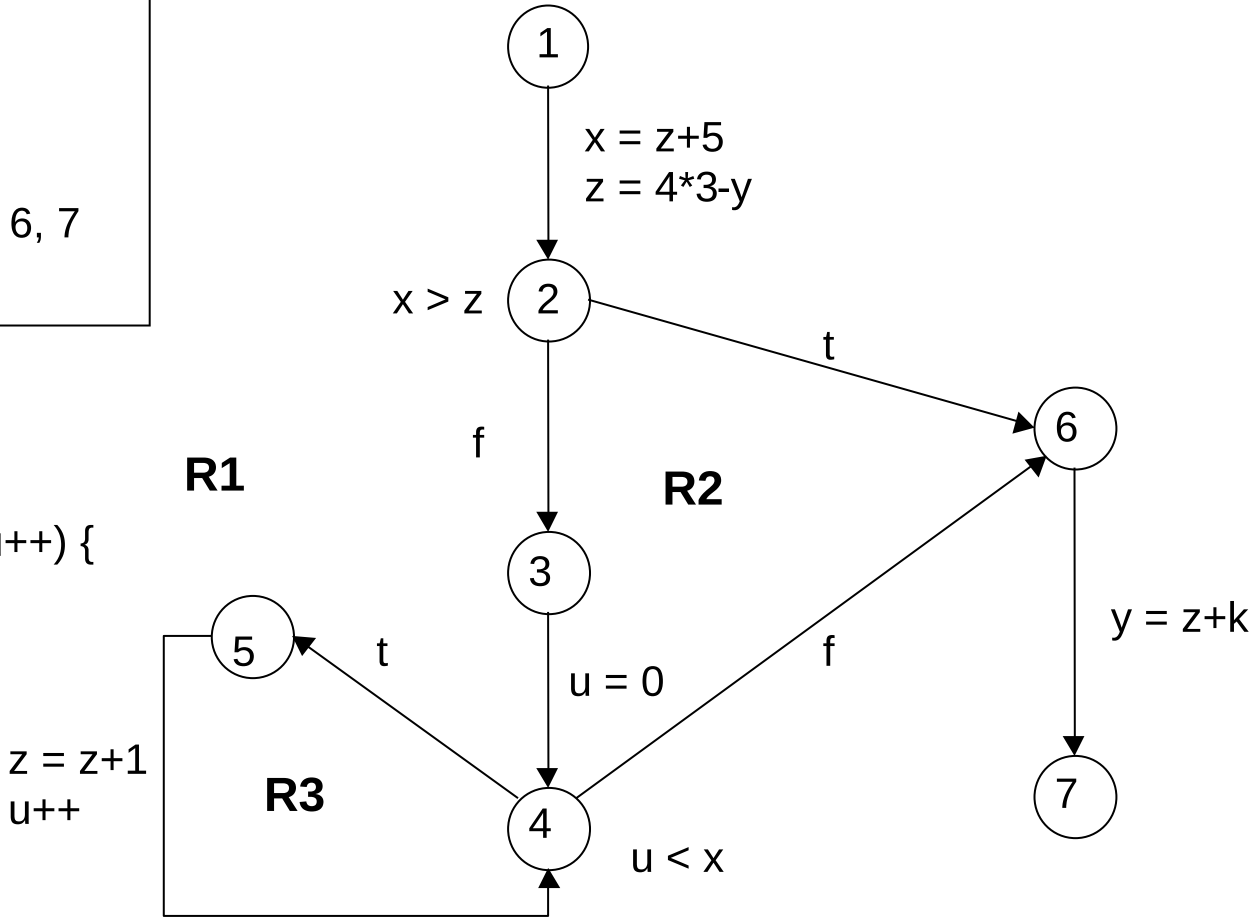
  - It helps to reduce the redundant tests
  - It focuses attention on program logic
  - It helps facilitates analytical versus arbitrary case design
  - Test cases which exercise basis set will execute every statement in a program at least once

Flow Graphs – Use in determining Paths for Testing - Revisited

V(G) = 3

Basis set:
1, 2, 3, 4, 6, 7
1, 2, 3, 4, 5, 4, 6, 7
1, 2, 6, 7

```
x = z+5
z = 4*3-y
if(x > z) goto A;
for( u=0; u < x; u++) {
     z = z+1;
};
A: y = z + k
```

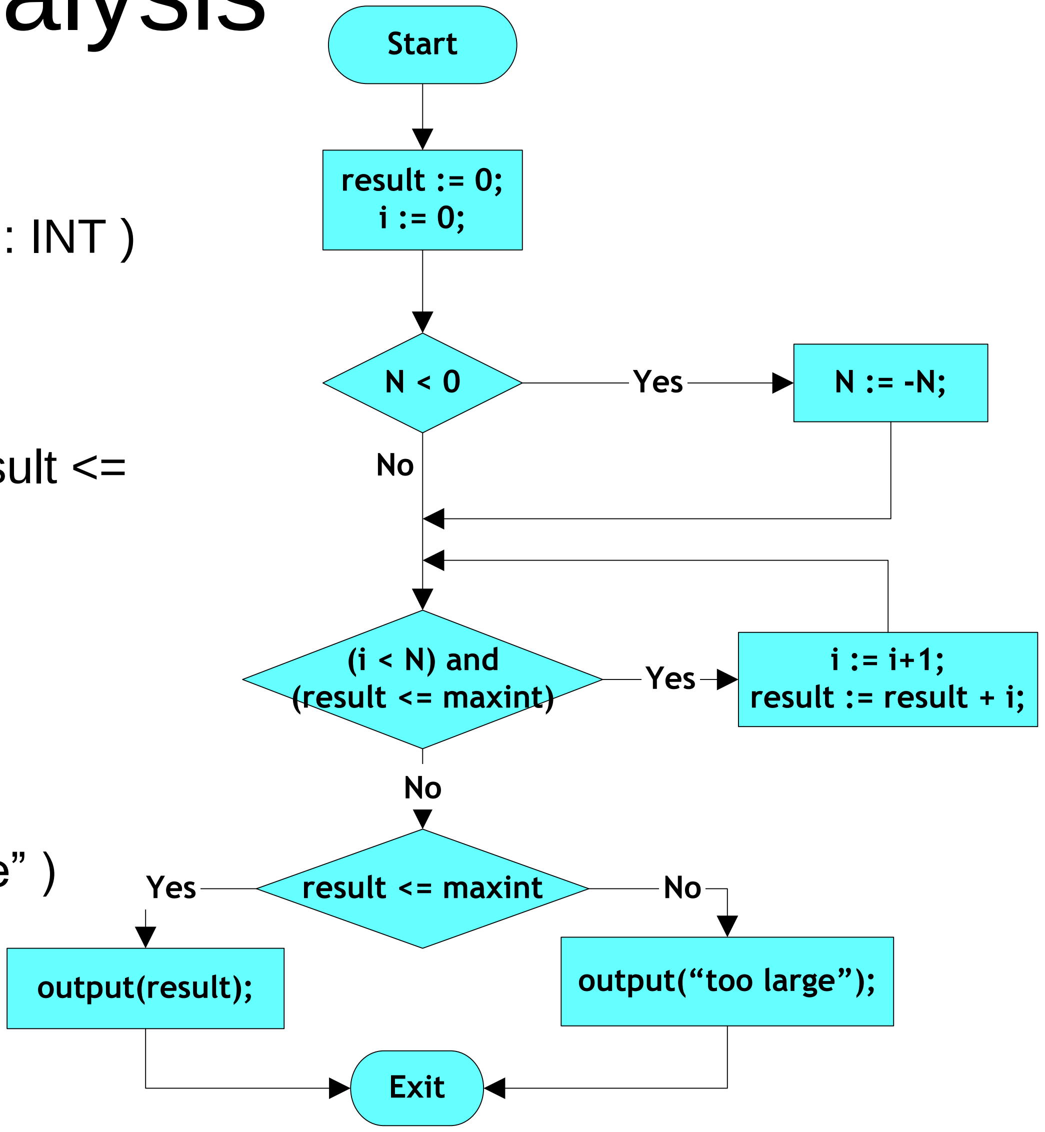


Example of a simple control flowgraph.

Control flow analysis

Example 2

```
1   PROGRAM  sum ( maxint, N : INT )
2     INT    result := 0 ;   i := 0 ;
3     IF   N < 0
4     THEN   N  :=  - N ;
5     WHILE   ( i < N )  AND  ( result <=
    maxint )
6     DO i  :=  i + 1 ;
7            result  :=  result + i ;
8     OD;
9     IF   result <= maxint
10    THEN   OUTPUT ( result )
11    ELSE   OUTPUT ( "too large" )
12  END.
```



Modified Condition/Decision Coverage

- Also known as MC/DC or MCDC
- Design test cases such that
  - every decision in the program has taken all possible outcomes at least once (branch or decision coverage)
  - every condition in a decision in the program has taken all possible outcomes at least once (condition coverage)
  - every condition in a decision has been shown to independently affect that decision's outcome (a condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions)
    - (condition – a Boolean expression containing no Boolean operators)
    - (decision – a Boolean expression composed of conditions and zero or more Boolean operators)
- Created at Boeing, is required for aviation software for the Federal Aviation Administration (FAA) in the USA by RCTA/DO-178B
- Usually considered necessary for adequate testing of critical software

Modified Condition/Decision Coverage

- Designed for languages with logical operators that do not short-circuit, such as Ada and Visual Basic .Net (Or/OrElse, And/AndAlso)
- For languages with logical operators that short-circuit, as in C, C++, Java and C#, MC/DC requires exactly the same test cases as condition/decision coverage (?)
  - the logical operators in these languages only evaluate conditions when their result can affect the encompassing decision
- More information:
  - RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", Radio Technical Commission for Aeronautics, USA, December 1992
  - Chilenski1994 John Joseph Chilenski and Steven P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing", Software Engineering Journal, September 1994, Vol. 9, No. 5, pp.193-200.
  - A Practical Tutorial on Modified Condition / Decision Coverage, NASA / TM-2001-210876 (http://www.faa.gov/certification/aircraft/av-info/software/Research/MCDC%20Tutorial.pdf)

Example: Modified Condition/Decision Coverage

- Consider the following fragment of code:

```
if
    A or (B and C)
then
    do_something;
else
    do_something_else;
end if;
```

- MC/DC may be achieved with the following set of test inputs (note that there are alternative sets of test inputs, which will also achieve MC/DC):

Case	**A**	**B**	**C**	**Outcome**
1	FALSE	FALSE	TRUE	FALSE
2	TRUE	FALSE	TRUE	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

are not evaluated if logical operators short-circuit

cases 2 a 4 are sufficient for branch and condition coverage, but only if logical operators do not short-circuit

- Because:
  - A is shown to independently affect the outcome of the decision condition by case 1 and case 2
  - B is shown to independently affect the outcome of the decision condition by case 1 and case 3
  - C is shown to independently affect the outcome of the decision condition by case 3 and case 4

Path coverage

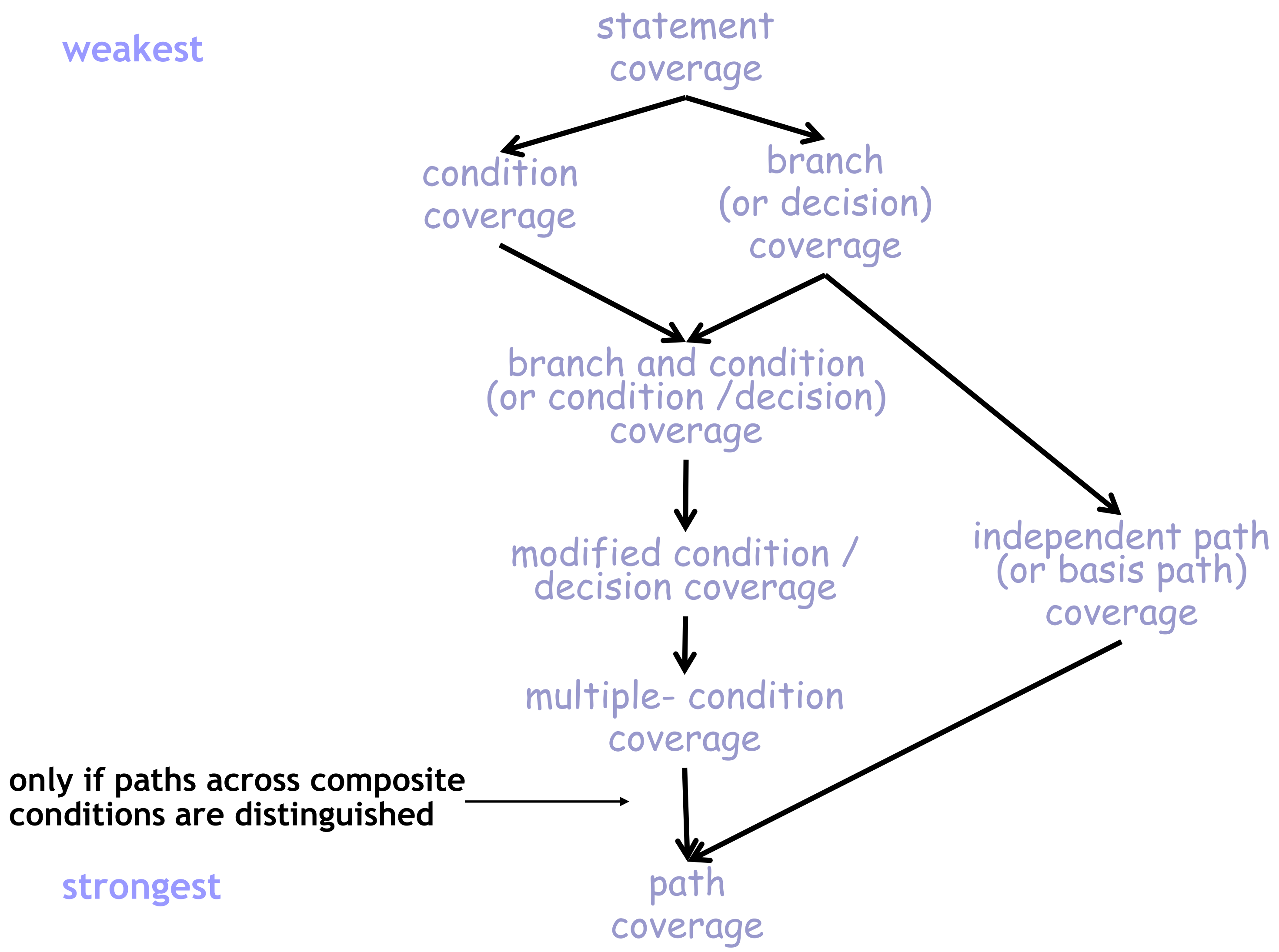
- Execute every possible path of a program,

  i.e., every possible sequence of statements
- Strongest white-box criterion (based on control flow analysis)
- Usually impossible: infinitely many paths (in case of loops)
- So: not a realistic option
- But note : enormous reduction w.r.t. all possible test cases

  (each sequence of statements executed for only one value)

  (doesn't mean exhaustive testing)

White-Box Testing : Overview


weakest
statement coverage
condition coverage
branch (or decision) coverage
branch and condition (or condition /decision) coverage
modified condition / decision coverage
independent path (or basis path) coverage
multiple- condition coverage
only if paths across composite conditions are distinguished
strongest
path coverage

Basic Idea

- Test some paths of program graph:
  - Efficiency: as few paths as possible
  - Completeness: paths form a (mathematical) **basis**
    - Not much overlap among paths
    - As many paths as needed to cover complexity of branching in program graph
- Intuition: all crucial behavior is tested
- Formula
  - CC=E-N+2 or CC=E-N+p Where
  - CC = Cyclomatic Complexity
  - E = the number of edges of the graph
  - N = the number of nodes of the graph
  - *P* = the number of connected components.

(McCabe) Basis Path Testing

- in math, a basis "spans" an entire space, such that everything in the space can be derived from the basis elements.

- the cyclomatic number of a strongly connected directed graph is the number of linearly independent cycles.

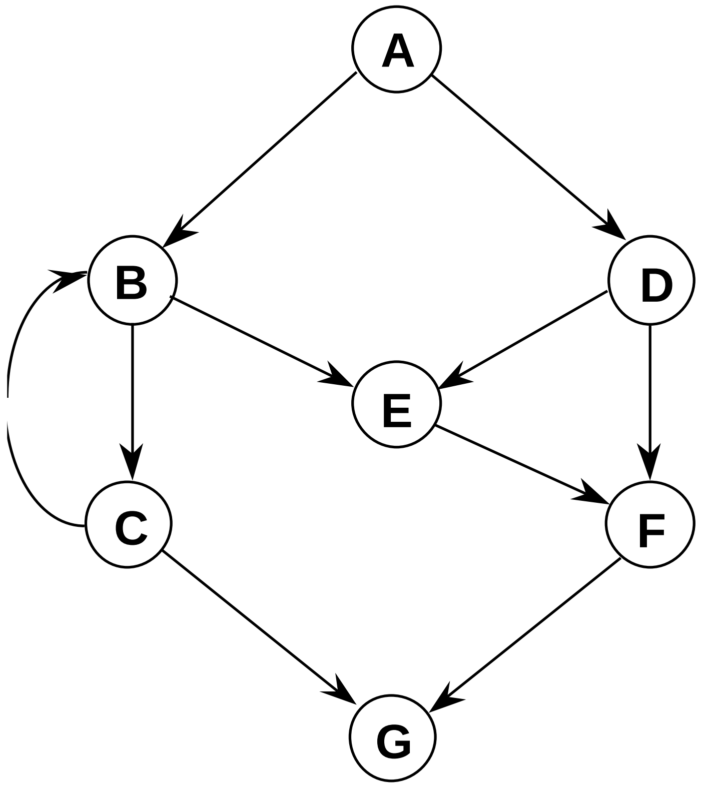
- given a program graph, we can always add an edge from the sink node to the source node to create a strongly connected graph. (assuming single entry, single exit)

- computing V(G) = e - n + p from the modified program graph yields the number of independent paths that must be tested.

- since all other program execution paths are linear combinations of the basis path, it is necessary to test the basis paths. (Some say this is sufficient; want to buy a bridge?)

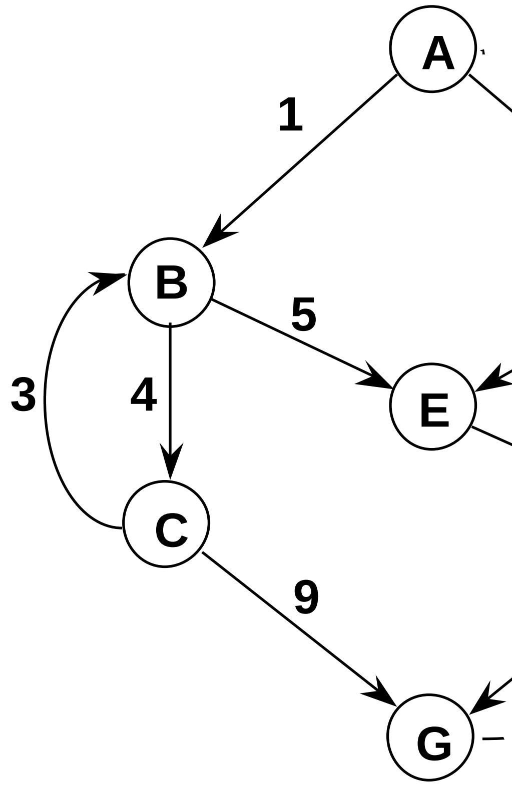
- the next few slides follow McCabe's original example.

McCabe's Control Graph



$$V(G) = 10 - 7 + 2(1)$$
$$= 5$$

Derived, Strongly Connected Graph



McCabe's Baseline Method

To determine a set of basis paths,

1. Pick a "baseline" path that corresponds to normal execution. (The baseline should have as many decisions as possible.)

2. To get succeeding basis paths, retrace the baseline until you reach a decision node. "Flip" the decision(take another alternative) and continue as much of the baseline as possible.

3. Repeat this until all decisions have been flipped. When you reach V(G) basis paths, you're done.

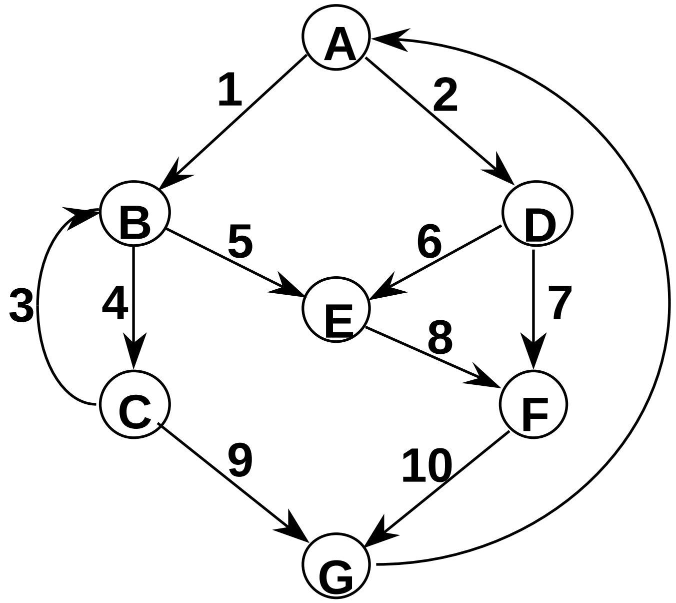
4. If there aren't enough decisions in the first baseline path, find a second baseline and repeat steps 2 and 3.

Following this algorithm, we get basis paths for McCabe's example.

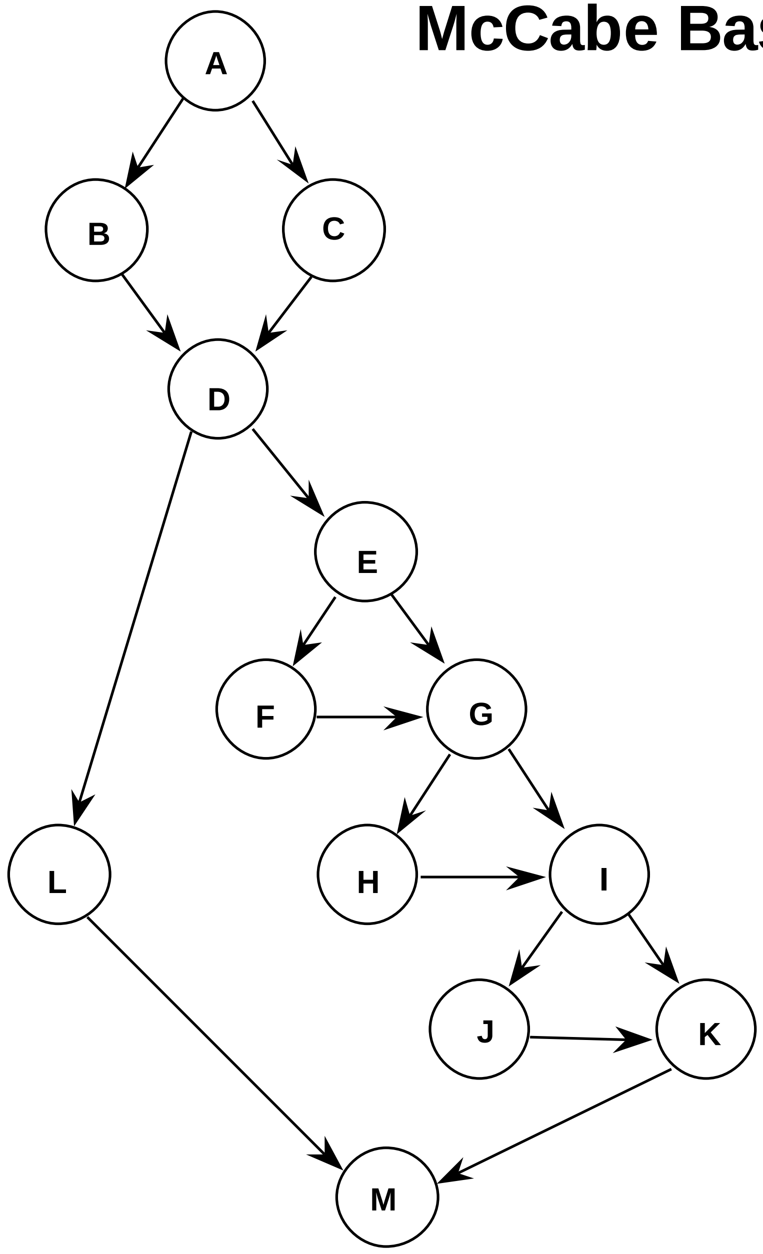
Basis Paths

path \ edges traversed	1	2	3	4	5	6	7	8	9	10
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	1	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	1	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	1	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

ex1 = p2 + p3 - p1
ex2 = 2p2 - p1

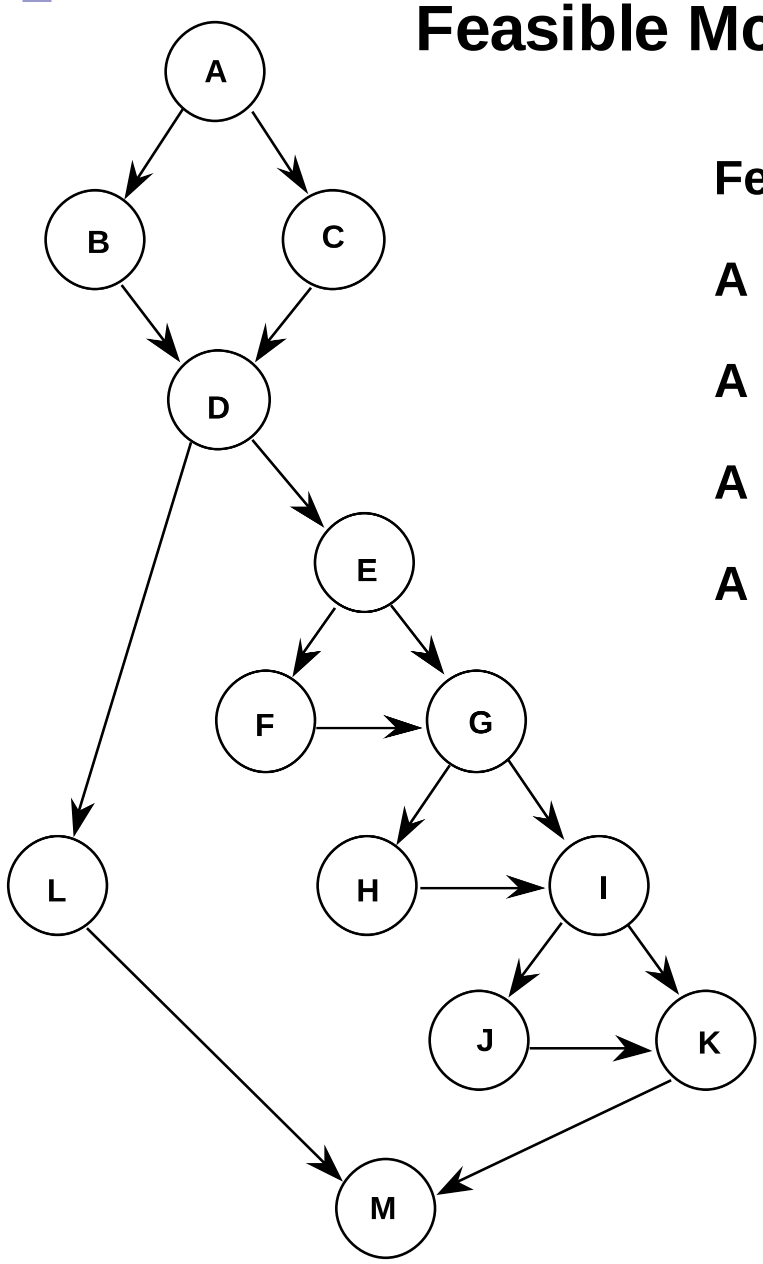


McCabe Basis Paths in the Triangle Program



There are 18 topologically possible paths.

Feasible McCabe Basis Paths



Feasible Basis Path Set B1

A - C - D - L - M - A	Not a triangle
A - B - D - E - F - G - I - K - M - A	Isosceles
A - B - D - E - G - H - I - K - M - A	Equilateral
A - B - D - E - G - I - J - K - M - A	Scalene

Reasoning about source code:

If B Then D, E

IF C Then D, L

IF F Then ~H and ~ J

IF H Then ~J

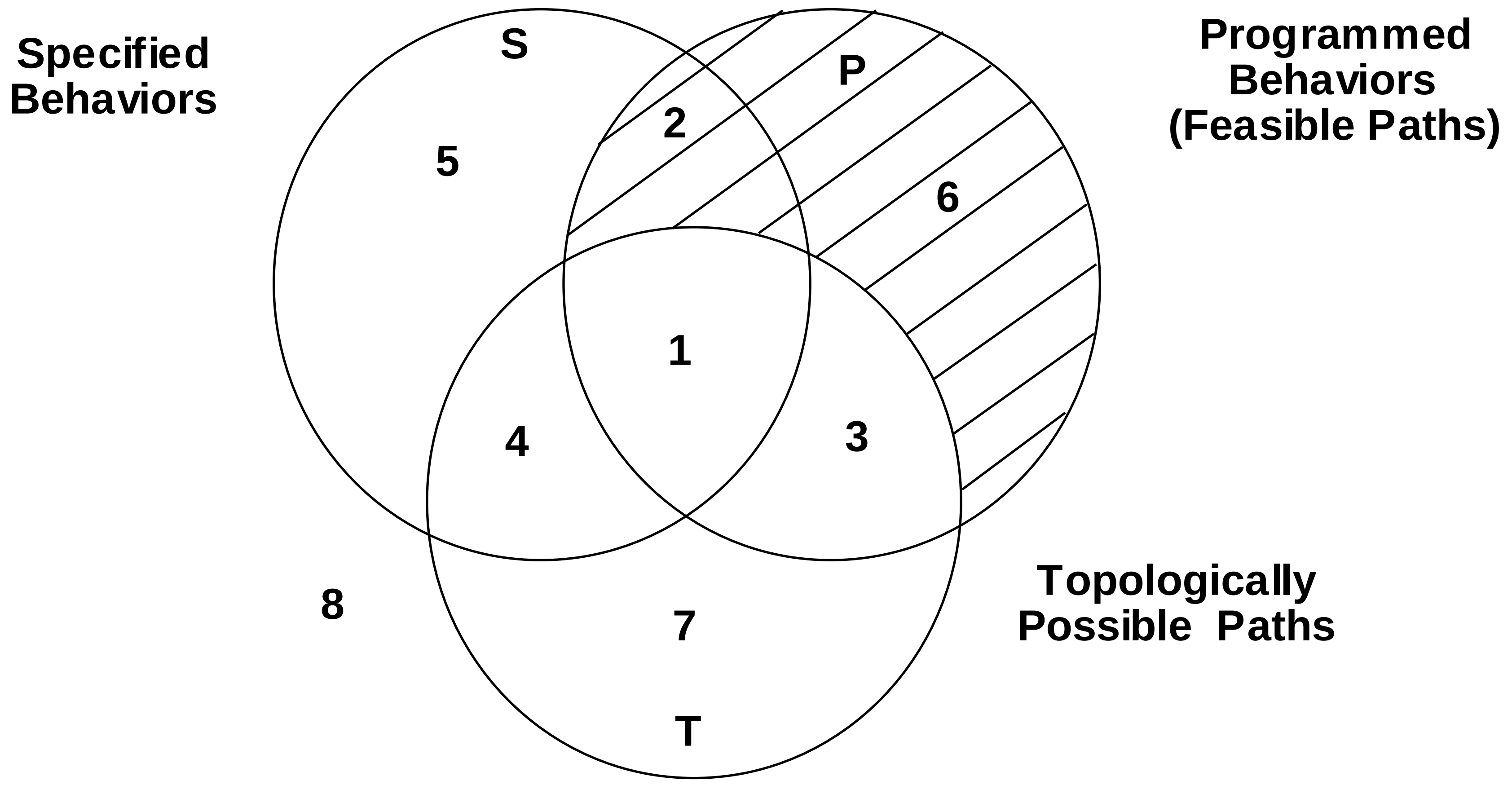
IF E Then ~L

Cons and Pros

1. Linear combinations of execution paths are counter-intuitive.
2. How does the baseline method guarantee feasible basis paths?
3. Given a set of feasible basis paths, is this a sufficient test?

On the positive side,

4. McCabe's approach does address both gaps and redundancies.
5. Essential complexity leads to better programming practices.
6. McCabe proved that violations of the structured programming constructs increase cyclomatic complexity, and violations cannot occur singly.


Specified Behaviors
S
Programmed Behaviors (Feasible Paths)
P
Topologically Possible Paths
T
1
2
3
4
5
6
7
8

Regions 2 and 6 must be empty (every feasible path is topologically possible)
Region 3: unspecified topologically feasible. Check spec?
Region 4: specified, topologically possible, and infeasible. Coding errors?
Region 5: missing behaviors that cannot be detected with path testing
Region 7: technically, not a problem (infeasible). Could become feasible with poor maintenance!

- Questions?

Data flow testing

Data Flow Testing

- Often confused with "dataflow diagrams"
- Main concern: places in a program where data values are defined and used
- Static (compile time) and dynamic (execution time) versions
- Static: Define/Reference Anomalies on a variable that
  - is defined but never used (referenced)
  - is used but never defined
  - is defined more than once
- Starting point is a program, P, with program graph G(P), and the set V of variables in program P.
- "Interesting" data flows are then tested as "mini-functions"

Definitions

Node n $\in$ G(P) is a *defining node* of the variable v $\in$ V, written as DEF(v,n), iff the value of the variable v is defined at the statement fragment corresponding to node n.

Node n $\in$ G(P) is a *usage node* of the variable v $\in$ V, written as USE(v, n), iff the value of the variable v is used at the statement fragment corresponding to node n.

A usage node USE(v, n) is a *predicate use* (denoted as P-use) iff the statement n is a predicate statement; otherwise USE(v, n) is a*computation use*, (denoted C-use).

A *definition-use (sub)path*with respect to a variable v (denoted du-path) is a (sub)path in PATHS(P) such that, for some v $\in$ V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the (sub)path.

A *definition-clear (sub)path*with respect to a variable v (denoted dc-path) is a definition-use (sub)path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the (sub)path is a defining node of v.

Some Definitions

- **Simple path** is a path in which all nodes except possibly the first and the last are distinct
- **Loop free path** is a path in which all nodes are distinct
- **Complete path** is a path from the entry node to the exit node of the CFG
- **Du-Path** with respect to variable x at node n_1 is a path $[n_1, n_2, …. n_k]$ where n_1 has a global definition of x and either
  - Node n_k has a global c-use of x and $[n_1….n_k]$ is def-clear simple path with respect to x or,
  - Node n_k has a p-use of x and $[n_1….nj]$ is a def-clear loop-free path with respect to x

Defining Node

- Input statements
- Assignment statements
- Loop Control statements
- Procedure Calls

Use Node

- Output statements
- Assignment statements
- Conditional statements
- Loop Control statements
- Procedure calls

Predicate Use Out degree ≥ 2

Computation Use Out degree ≤ 1

- Seven data flow testing criteria
  - All-defs
  - All-c-uses
  - All-p-uses
  - All-p-uses/some-c-uses
  - All-c-uses/some-p-uses
  - All-uses
  - All-du-paths

Some more examples of DU

```
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	

Define / Use Information Example

Variable	Defined at	Used at	Comment
locks	9		Declaration
locks	22		READ
locks		23	Predicate use
locks		26	Computation Use
stocks	9		READ
stocks		27	Computation Use
num_locks	26		Assignment
num_locks		26	Computation Use
num_locks		33	WRITE

Lock, Stock, and Barrel Problem Statement

Rifle salespersons in the Arizona Territory sell rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost $45.00, stocks cost $30.00, and barrels cost $25.00. Salespersons must sell at least one complete rifle per month, and production limits are such that the most one salesperson could sell in a month is 70 locks, 80 stocks, and 90 barrels. Each rifle salesperson sends a telegram to the Missouri company with the total order for each town (s)he visits; salespersons visit at least one town per month, but travel difficulties make ten towns the upper limit. At the end of each month, the company computes commissions as follows: 10% on sales up to $1000, 15% on the next $800, and 20% on any sales in excess of $1800. The company has four salespersons. The telegrams from each salesperson are sorted into piles (by person) and at the end of each month a datafile is prepared, containing the sales person's name, followed by one line for each telegram order, showing the number of locks, stocks, and barrels in that order. At the end of the sales data lines, there is an entry of "-1" in the position where the number of stocks would be to signal the end of input for that salesperson. The program produces a monthly sales report that gives the salesperson's name, the total number of locks, stocks, and barrels sold, the salesperson's total dollar sales, and finally his/her commission.

Program Description

Inputs

Variable	Range
(town visit)	1 Š town visit Š 10
locks	1 Š □locks(visit) Š 70
stocks	1 Š □stocks(visit) Š 80
barrels	1 Š □barrels(visit) Š 90

Outputs

Variable	Range
num_locks	1 Š num_locks Š 70
num_stocks	1 Š num_stocks Š 80
num_barrels	1 Š num_barrels Š 90
sales	100 Š sales Š 7800
commission	10 Š commission Š 1420

Functional Decomposition

F(visits) = F(locks, stocks, barrels) = Commission

F1(locks, stocks, barrels) = (num_locks, num_stocks, num_barrels)

F2(num_locks, num_stocks, num_barrels) = sales

F3(sales) = commission

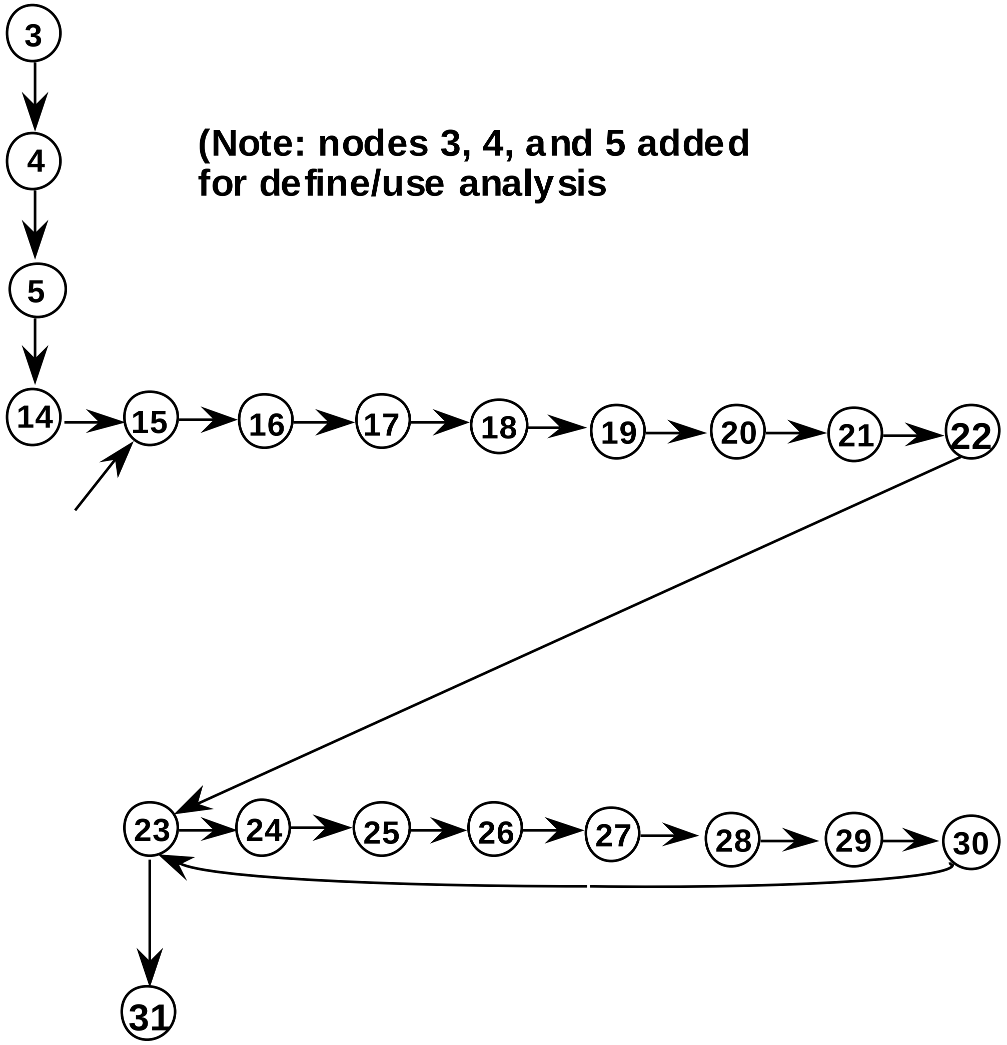
Program Source for Lock, Stock and Barrel

```
program lock_stock_and_barrel
const
    lock_price   = 45.0;
    stock_price  = 30.0;
    barrel_price = 25.0;
type
    STRING_30 = string[30];  {Salesman's Name}
var
    locks, stocks, barrels, num_locks, num_stocks,
    num_barrels, salesman_index, order_index :  INTEGER;
     sales, commission : REAL;
     salesman : STRING_30;

 begin {program lock_stock_and_barrel}
    FOR  salesman_index := 1 TO  4 DO
      BEGIN
        READLN(salesman);
        WRITELN ('Salesman is ', salesman);
        num_locks := 0;
        num_stocks  := 0;
        num_barrels  := 0;
       READ(locks);
       WHILE locks <> -1 DO
         BEGIN
           READLN(stocks, barrels);
           num_locks := num_locks + locks;
           num_stocks := num_stocks + stocks;
           num_barrels := num_barrels + barrels;
           READ(locks);
        END; {WHILE locks}
    READLN;
    WRITELN('Sales for ',salesman);
    WRITELN('Locks sold: ', num_locks);
    WRITELN('Stocks sold: ', num_stocks);
    WRITELN('Barrels sold: ', num_barrels);
    sales := lock_price * num_locks +
     stock_price * num_stocks +
     barrel_price * num_barrels;
    WRITELN('Total sales: ', sales:8:2);
    WRITELN;
    IF (sales > 1800.0) THEN
       BEGIN
         commission := 0.10 * 1000.0
         commission := commission + 0.15 * 800.0
         commission := commission + 0.20 * (sales - 1800.0)
       END;
     ELSE IF (sales > 1000.0) THEN
         BEGIN
            commission := 0.10 * 1000.0
            commission := commission + 0.15 * (sales - 1000.0)
         END;
     ELSE commission := 0.10 * sales;
    WRITELN('Commission is $',commission:6:2);
     END; {FOR salesman}
  end. {program lock_stock_and_barrel}
```

Program Graph for Lock, Stock and Barrel

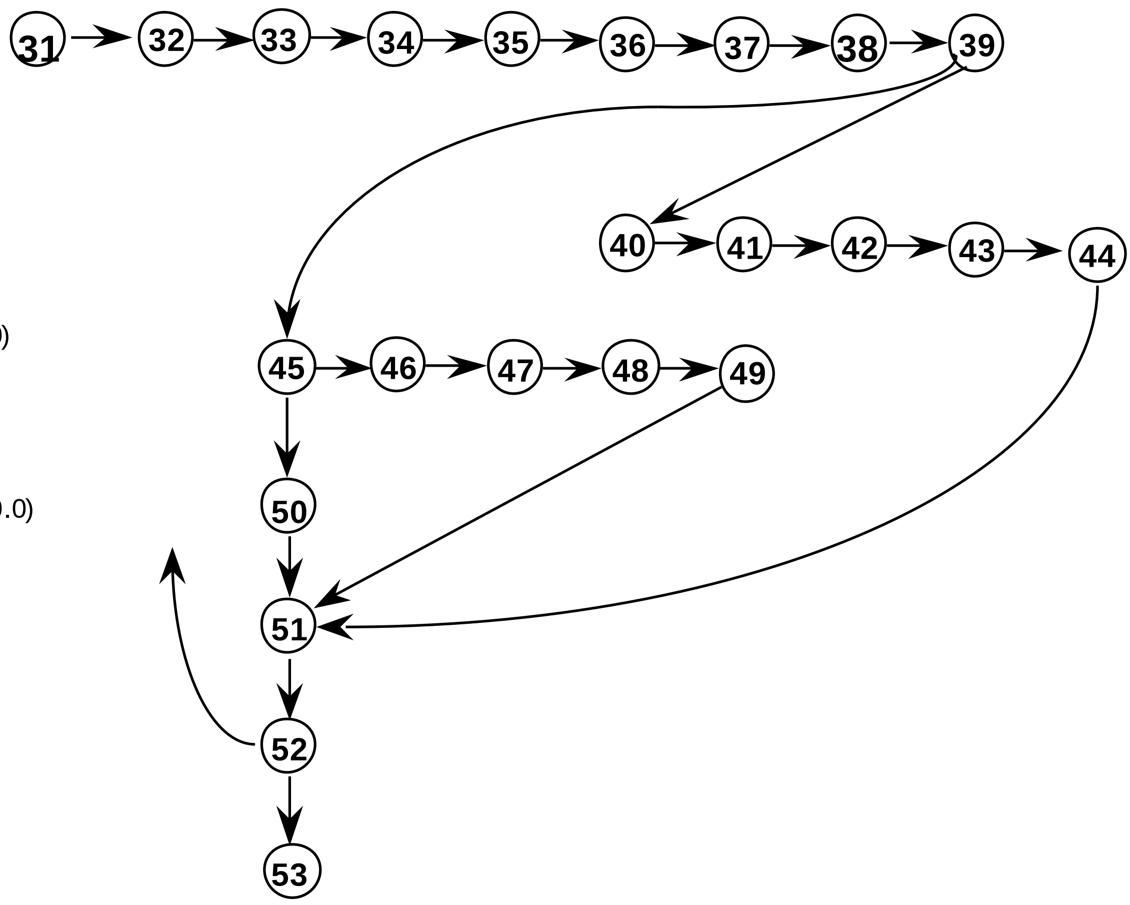
```
1   program lock_stock_and_barrel;
2   const
3      lock_price   = 45.0;
4      stock_price  = 30.0;
5      barrel_price = 25.0;
6   type
7      STRING_30 = string[30];  {Salesman's Name}
8   var
9      locks, stocks, barrels, num_locks, num_stocks,
10     num_barrels, salesman_index, order_index :  INTEGER;
11      sales, commission : REAL;
12      salesman : STRING_30;
13
14  begin {program lock_stock_and_barrel}
15     FOR  salesman_index := 1 TO 4 DO
16       BEGIN
17         READLN(salesman);
18         WRITELN ('Salesman is ', salesman);
19         num_locks := 0;
20         num_stocks := 0;
21         num_barrels := 0;
22        READ(locks);
23        WHILE locks <> -1 DO
24          BEGIN
25            READLN(stocks, barrels);
26            num_locks := num_locks + locks;
27            num_stocks := num_stocks + stocks;
28            num_barrels := num_barrels + barrels;
29            READ(locks);
30         END; {WHILE locks}
```



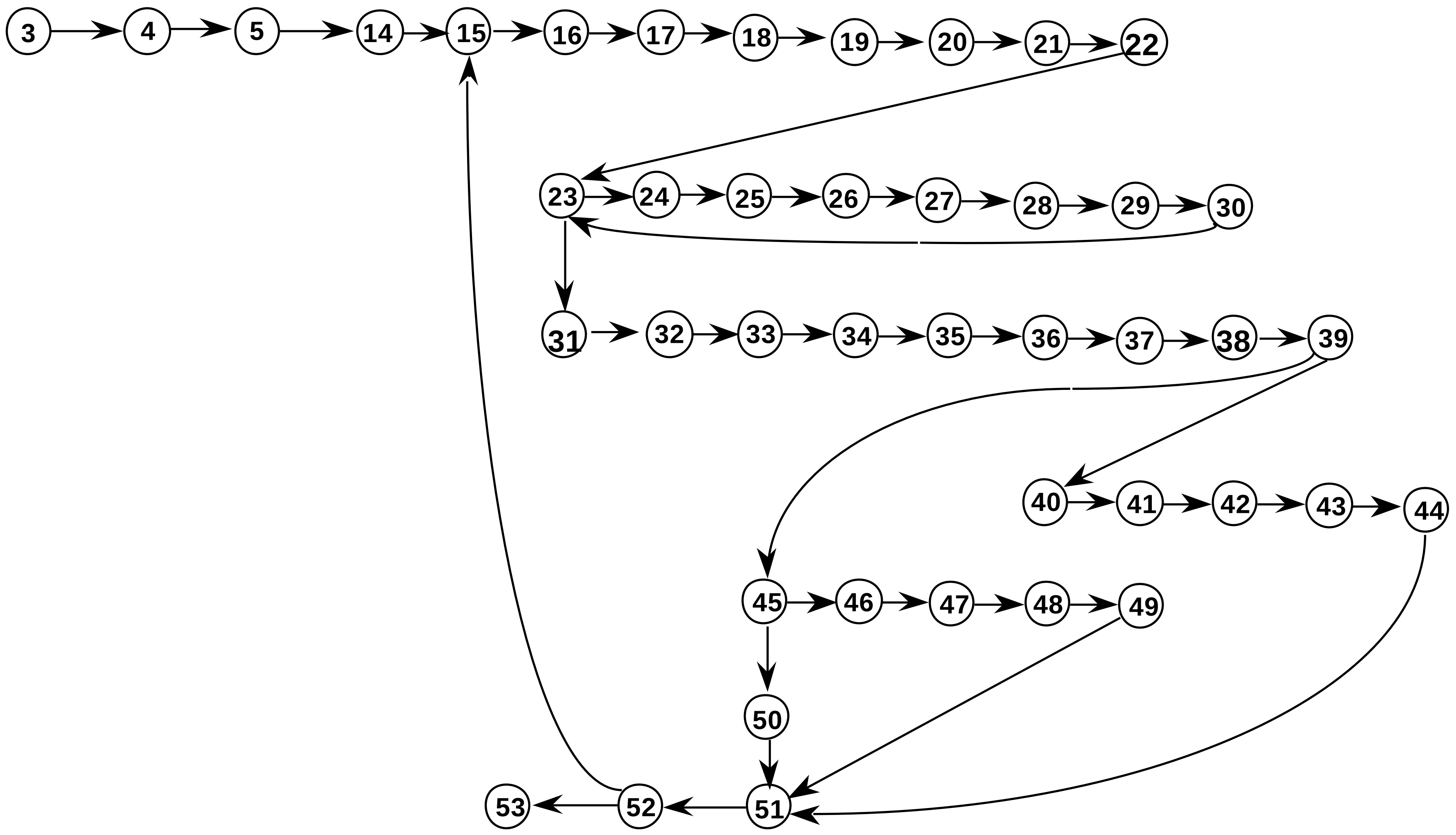
(Note: nodes 3, 4, and 5 added for define/use analysis

Program Graph for Lock, Stock and Barrel

```
31     READLN;
32     WRITELN('Sales for ',salesman);
33     WRITELN('Locks sold: ', num_locks);
34     WRITELN('Stocks sold: ', num_stocks);
35     WRITELN('Barrels sold: ', num_barrels);
36     sales := lock_price * num_locks +
                   stock_price * num_stocks +
                   barrel_price * num_barrels;
37     WRITELN('Total sales: ', sales:8:2);
38     WRITELN;
39     IF (sales > 1800.0)
40        THEN BEGIN
41          commission := 0.10 * 1000.0
42          commission := commission + 0.15 * 800.0
43          commission := commission + 0.20 * (sales - 1800.0)
44        END;
45      ELSE IF (sales > 1000.0)
46          THEN BEGIN
47             commission := 0.10 * 1000.0
48             commission := commission + 0.15 * (sales - 1000.0)
49          END;
50      ELSE commission := 0.10 * sales;
51     WRITELN('Commission is $',commission:6:2);
52      END; {FOR salesman}
53   end. {program lock_stock_and_barrel}
```

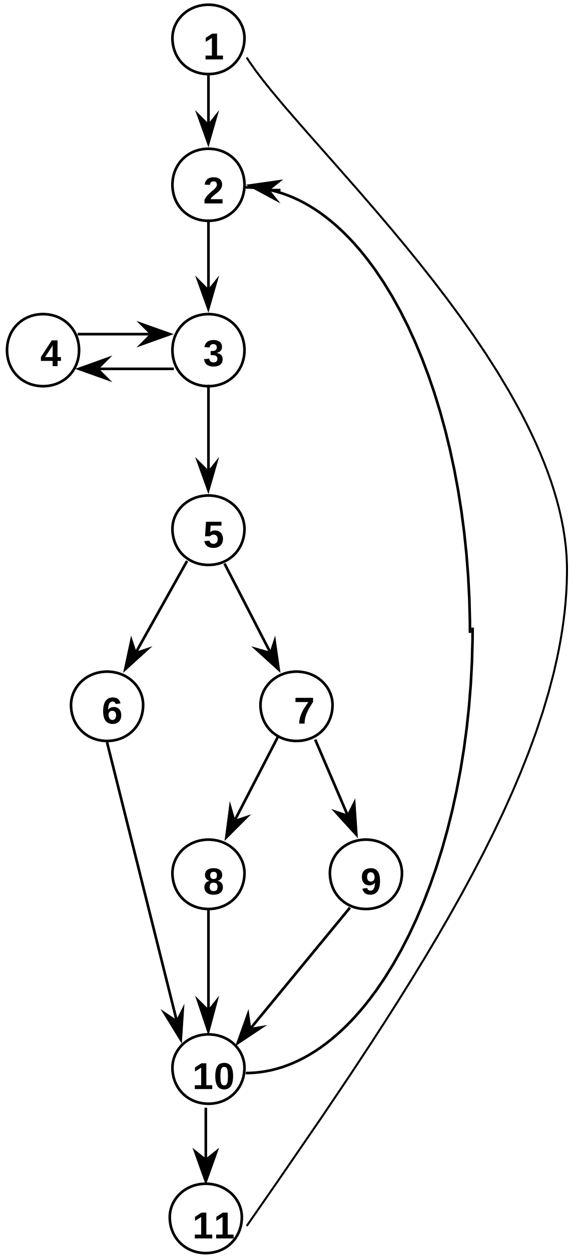


Program Graph for Lock, Stock and Barrel



DD-Path Graph for Lock, Stock and Barrel

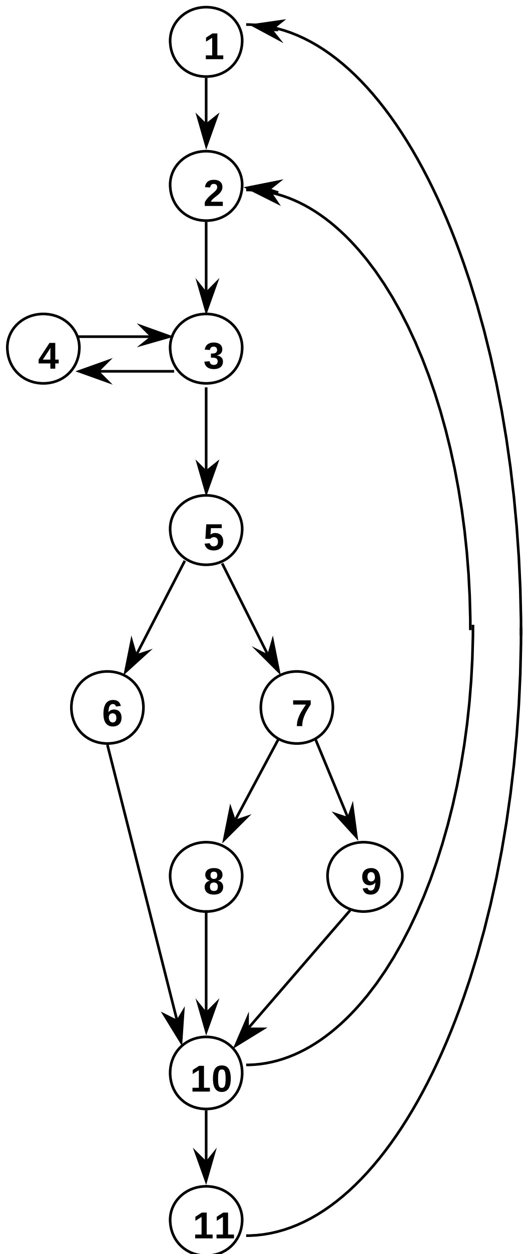
DD-Path	Statements
1	3-5, 14
2	15 - 22
3	23
4	24 - 30
5	31 - 39
6	40 - 44
7	45
8	46 - 49
9	50
10	51 - 52
11	53



McCabe Basis Paths for Lock, Stock and Barrel

$$\begin{aligned} V(G) &= e - n + 2 \\ &= 15 - 11 + 2 \\ &= 6 \end{aligned}$$

1 - 2 - 3 - 5 - 7 - 9 - 10 - 2
1 - 2 - 3 - 5 - 7 - 9 - 10 - 11 - 1
1 - 2 - 3 - 5 - 7 - 8 - 10 - 2
1 - 2 - 3 - 5 - 7 - 8 - 10 - 11
1 - 2 - 3 - 5 - 6 - 10 - 2
1 - 2 - 3 - 4 - 3 - 5 - 6 - 10 - 2



Define/Use Nodes for Lock, Stock and Barrel

d-3 (lock_price)	
d-4 (Stock_price)	
d-5 (barrel_price)	
d-15 (salesman_index),	u-15 (salesman_index)
d-17 (salesman)	
	u-18 (salesman)
d-19 (num_locks)	
d-20 (num_stocks)	
d-21 (num_barrels)	
d-22 (locks)	
	u-23 (locks)
d-25 (stocks), d-25 (barrels)	
d-26 (num_locks)	u-26 (num_locks), u-26 (locks)
d-27 (num_stocks)	u-27 (num_stocks), u-27 (stocks)
d-28 (num_barrels)	u-28 (num_barrels), u-28 (barrels)
d-29 (locks)	
	u-32 (salesman)
	u-33 (num_locks)
	u-34 (num_stocks)
	u-35 (num_barrels)
d-36 (sales)	u-36 (num_locks), u-36 (lock_price)
	u-36 (num_stocks), u-36 (stock_pric
	u-36 (num_barrels), u-36 (barrels)
	u-37 (sales)
	u-39 (sales)
d-41 (commission)	
d-42 (commission)	u-42 (commission)
d-43 (commission)	u-43 (commission), u-43 (sales)
d-47 (commission)	
d-48 (commission)	u-48 (commission), u-48 (sales)
d-50 (commission)	u-50 (sales)
	u-51 (commission)

Define/Use Test Cases

Notation: d-3 (lock_price) means lock_price is defined at node 3

Technique: for a particular variable,

1. **find all its definition and usage nodes, then**
2. **find the feasible paths among these.**
3. **for each path, devise a "suitable" (functional?) set of test cases.**

du-paths for stocks:

d-25 (stocks), u-27 (stocks) are the only define and usage nodes, so there can be only one du-path for stocks:

```
25              READLN(stocks, barrels);
26              num_locks := num_locks + locks;
27              num_stocks := num_stocks + stocks;
```

This path is definition clear (and also pretty boring). Even so, we might want to do traditional equivalence class testing. (What effect does a negative value of stocks have?)

du-paths for locks

there are two definition nodes, d-22 (locks) and d-29 (locks), and two usage nodes, u-23 (locks) and u-26 (locks), so there are four du-paths:

p1 = <22, 23>
p2 = <22, 23, 24, 25, 26>
p3 = <29, 30, 23>
p4 = <29, 30, 23, 24, 25, 26>

```
22          READ(locks);
23          WHILE locks <> -1 DO
24            BEGIN
25              READLN(stocks, barrels);
26              num_locks := num_locks + locks;
27              num_stocks := num_stocks + stocks;
28              num_barrels := num_barrels + barrels;
29              READ(locks);
30            END; {WHILE locks}
```

Part of the "problem" with locks is that the single variable has two roles: it is used in computation and it serves as a loop index. Our test cases will need to exercise each function.

Define / Use Information Example

Variable	Defined at	Used at	Comment
locks	9		Declaration
locks	22		READ
locks		23	Predicate use
locks		26	Computation Use
stocks	9		READ
stocks		27	Computation Use
num_locks	26		Assignment
num_locks		26	Computation Use
num_locks		33	WRITE

Variable	Defined at	Used at	Comment
locks	9		(to compiler)
locks	22		READ
locks		23	predicate use
locks		26	computation use
locks	29		READ
stocks	9		(to compiler)
stocks	25		READ
stocks		27	computation use
num_locks	9		(to compiler)
num_locks	19		assignment
num_locks	26		assignment
num_locks		26	computation use
num_locks		33	WRITE
num_locks		36	computation use
sales	11		(to compiler)
sales	36		assignment
sales		37	WRITE
sales		39	predicate use
sales		43	computation use
sales		45	predicate use
sales		48	computation use
sales		50	computation use
commission	11		(to compiler)
commission	41		assignment
commission	42		assignment
commission		42	computation use
commission	43		assignment
commission		43	computation use
commission	47		assignment
commission	48		assignment
commission		48	computation use
commission	50		assignment
commission		51	WRITE

Du-Path	Variable	Def Node	Use Node
1	locks	22	23
2	locks	22	26
3	locks	29	23
4	locks	29	26
5	stocks	25	27
6	barrels	25	28
7	num_locks	19	26
8	num_locks	19	33
9	num_locks	19	36
10	num_locks	26	33
11	num_locks	26	36
12	num_stocks	20	27
13	num_stocks	20	34
14	num_stocks	20	36
15	num_stocks	27	34
16	num_stocks	27	36
17	num_barrels	21	28
18	num_stocks	21	35
19	num_stocks	21	36
20	num_stocks	28	35
21	num_stocks	28	36
22	sales	36	37
23	sales	36	39
24	sales	36	43
25	sales	36	45
26	sales	36	48
27	sales	36	50
28	commission	41	42
29	commission	42	43
30	commission	43	51
31	commission	47	48
32	commission	48	51
33	commission	50	51

Coverage Metrics Based on du-paths

In the following definitions, T is a set of (sub)paths in the program graph G(P) of a program P, with the set V of variables.

The set T satisfies the *All-Defs* criterion for the program P iff for every variable v∈ V, T contains definition clear (sub)paths from every defining node of v to a use of v.

The set T satisfies the *All-Uses* criterion for the program P iff for every variable v∈ V, T contains definition clear (sub)paths from every defining node of v to every use of v, and to the successor node of each USE(v,n).

The set T satisfies the *All-P-Uses /Some C-Uses* criterion for the program P iff for every variable v∈ V, T contains definition clear (sub)paths from every defining node of v to every predicate us of v, and if a definition of v has no P-uses, there is a definition clear path to at least one computation use..

The set T satisfies the *All-C-Uses /Some P-Uses* criterion for the program P iff for every variable v∈ V, T contains definition clear (sub)paths from every defining node of v to every computation use of v, and if a definition of v has no C-uses, there is a definition clear path to at least one predicate use..

The set T satisfies the *All-DU-paths* criterion for the program P iff for every variable v∈ V, T contains definition clear (sub)paths from every defining node of v to every use of v, and to the successor node of each USE(v,n), and that these paths are either single loop traversals, or they are cycle free.

P-Use=Predicate use (in decision)
C-Use=Computation use

```
1    s = 0;
2     i = 1;
3      while (i ≤ n)
          {
4               s+ = i;
 5                i++
            }
6       Count << s;
7       Count << i;
8       Count << n;
```

```
def 1(s)
def 2 (i)
         use (i)   --- loop
          use (n)   --- loop
   def 4 (s)
   use  4 (s)
   use 4 (i)
        def 5(i)
        use 5 (i)
use 6(s)
use 7 (i)
use 8 (n)
```

```
Procedure SumEven

int n, sum, j;

1      sum = 0;

2          j= 2;

3        n = read ()

4         while (n> 0) do

 5         sum = sum + j

 6         j : = j + 1;

  7        n: = n – 1;

Endwhile

8            Write (sum)
```

Slice for variable j as 6

Slices will be
Slice(2,3,4,6,7)

Slice for variable n
where n is 7

Slice(3,4,7)

Slice Testing

- Often confused with "module execution paths" (McCabe tool)
- Main concern: portions of a program that "contribute" to the value of a variable at some point in the program.
- Nice analogy with history -- a way to separate a complex system into "disjoint" components
- A dynamic construct
- Starting point is a program, P, with program graph G(P), and the set V of variables in program P. Nodes in the program graph are numbered and correspond to statement fragments.
- Definition: The slice on the variable set V at statement fragment n, written S(V, n), is the set of node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n.

Fine Points

- "prior to" is the dynamic part of the definition.

- "contribute" is best understood by extending the Define and Use concepts:

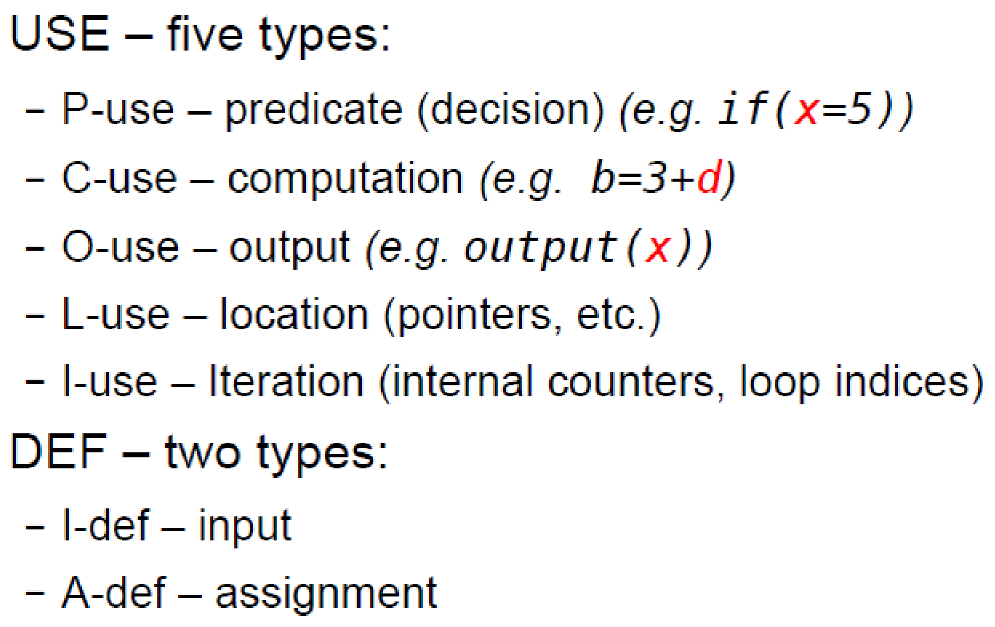
P-use	used in a predicate (decision)
C-use	used in computation
O-use	used for output
L-use	used for location (pointers, subscripts)
I-use	iteration (internal counters, loop indices)
I-def	defined by input
A-def	defined by assignment

- usually, the set V of variables consists of just one element.

- can choose to define a slice as a compileable set of statement fragments -- this extends the meaning of "contribute"

- because slices are sets, we can develop a lattice based on the subse relationship.

Fine points



USE – five types:

- P-use – predicate (decision) *(e.g.* `if(x=5)`*)*
- C-use – computation *(e.g. b=3+d)*
- O-use – output *(e.g.* `output(x)`*)*
- L-use – location (pointers, etc.)
- I-use – Iteration (internal counters, loop indices)

DEF – two types:

- I-def – input
- A-def – assignment

Program Source for Lock, Stock and Barrel

```
program lock_stock_and_barrel
const
    lock_price   = 45.0;
    stock_price  = 30.0;
    barrel_price = 25.0;
type
    STRING_30 = string[30];  {Salesman's Name}
var
    locks, stocks, barrels, num_locks, num_stocks,
    num_barrels, salesman_index, order_index :  INTEGER;
     sales, commission : REAL;
     salesman : STRING_30;

 begin {program lock_stock_and_barrel}
    FOR  salesman_index := 1 TO 4 DO
      BEGIN
        READLN(salesman);
        WRITELN ('Salesman is ', salesman);
        num_locks := 0;
        num_stocks := 0;
        num_barrels := 0;
       READ(locks);
       WHILE locks <> -1 DO
         BEGIN
           READLN(stocks, barrels);
           num_locks := num_locks + locks;
           num_stocks := num_stocks + stocks;
           num_barrels := num_barrels + barrels;
           READ(locks);
        END; {WHILE locks}
  READLN;
  WRITELN('Sales for ',salesman);
  WRITELN('Locks sold: ', num_locks);
  WRITELN('Stocks sold: ', num_stocks);
  WRITELN('Barrels sold: ', num_barrels);
  sales := lock_price * num_locks +
  stock_price * num_stocks +
  barrel_price * num_barrels;
  WRITELN('Total sales: ', sales:8:2);
  WRITELN;
  IF (sales > 1800.0) THEN
     BEGIN
       commission := 0.10 * 1000.0
       commission := commission + 0.15 * 800.0
       commission := commission + 0.20 * (sales - 1800.0)
     END;
   ELSE IF (sales > 1000.0) THEN
       BEGIN
          commission := 0.10 * 1000.0
          commission := commission + 0.15 * (sales - 1000.0)
       END;
   ELSE commission := 0.10 * sales;
  WRITELN('Commission is $',commission:6:2);
   END; {FOR salesman}
end. {program lock_stock_and_barrel}
```

Sample Slices

```
1   program lock_stock_and_barrel
2   const
3      lock_price   = 45.0;
4      stock_price  = 30.0;
5      barrel_price = 25.0;
6   type
7      STRING_30 = string[30];  {Salesman's Name}
8   var
9      locks, stocks, barrels, num_locks, num_stocks,
10     num_barrels, salesman_index, order_index :  INTEGER;
11      sales, commission : REAL;
12      salesman : STRING_30;
13
14  begin {program lock_stock_and_barrel}
15     FOR  salesman_index := 1 TO 4 DO
16       BEGIN
17         READLN(salesman);
18         WRITELN ('Salesman is ', salesman);
19         num_locks  := 0;
20         num_stocks  := 0;
21         num_barrels := 0;
22        READ(locks);
23        WHILE locks <> -1 DO
24          BEGIN
25            READLN(stocks, barrels);
26            num_locks := num_locks + locks;
27            num_stocks := num_stocks + stocks;
28            num_barrels := num_barrels + barrels;
29            READ(locks);
30         END; {WHILE locks}
```

S4: S(locks, 22) = {22}
S5: S(locks, 23) = {22, 23, 24, 29, 30}
S6: S(locks, 26) = {22, 23, 24, 29, 30}
S7: S(locks, 29) = {29}

S5: S(locks, 23) = {22, 23, 24, 29, 30}

```
22        READ(locks);
23        WHILE locks <> -1 DO
24          BEGIN
25            READLN(stocks, barrels);
26            num_locks := num_locks + locks;
27            num_stocks := num_stocks + stocks;
28            num_barrels := num_barrels + barrels;
29           READ(locks);
30         END; {WHILE locks}
```

P-use at node 23 and a C-use at node 26, and has two definitions, the I-defs at nodes 22 and 29.

Program Slices in Lock, Stock, and Barrel

S(salesman, 15) = ø
S(salesman, 30) = {15}
S(num_locks, 17) = ø
S(num_locks, 24) = {17, 20, 27?}
S(num_locks, 31) = {17, 20, 24, 27}
S(num_locks, 34) = {17, 20, 24, 27}
S(num_stocks, 18) = ø
S(num_stocks, 25) = {18, 23}
S(num_stocks, 32) = {18, 23, 25}
S(num_stocks, 34) = {18, 23, 25}
S(num_barrels, 19) = ø
S(num_barrels, 26) = {19,23}
S(num_barrels, 33) = {19, 23, 26}
S(num_barrels, 34) = {19, 23, 26}
S(locks, 20) = ø
S(locks, 24) = {20, 27?}
S(stocks, 25) = {23}
S(barrels, 26) = {23}
S(lock_price, 34) = {3}
S(stock_price, 34) = {4}
S(barrel_price, 34) = {5}
S(sales, 34) = {3, 4, 5, 17, 18, 19, 20, 23, 24, 25, 26, 27}
S(sales, 35) = {3, 4, 5, 17, 18, 19, 20, 23, 24, 25, 26, 27}
S(sales, 36) = {3, 4, 5, 17, 18, 19, 20, 23, 24, 25, 26, 27}
S(sales, 42) = {3, 4, 5, 17, 18, 19, 20, 23, 24, 25, 26, 27}
S(sales, 47) = {3, 4, 5, 17, 18, 19, 20, 23, 24, 25, 26, 27}
S(commission, 38) = ø
S(commission, 39) = {38 }
S(commission, 40) = {3, 4, 5,17,18,19, 20, 23, 24, 25, 26, 27, 34, 38, 39}
S(commission, 44) = ø
S(commission, 45) = {3, 4, 5,17,18,19, 20, 23, 24, 25, 26, 27, 34, 44}
S(commission, 47) = {3, 4, 5,17,18,19, 20, 23, 24, 25, 26, 27, 34}
S(commission, 48) = {3, 4, 5,17,18,19, 20, 23, 24, 25, 26, 27, 34, 38, 39, 40, 44,45,47

Developing a Lattice of Slices

S34: S(commission, 41) = {41}

S35: S(commission, 42) = {41, 42}

S36: S(commission, 43) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 36, 41, 42, 43}

S37: S(commission, 47) = {47}

S38: S(commission, 48) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 47, 48}

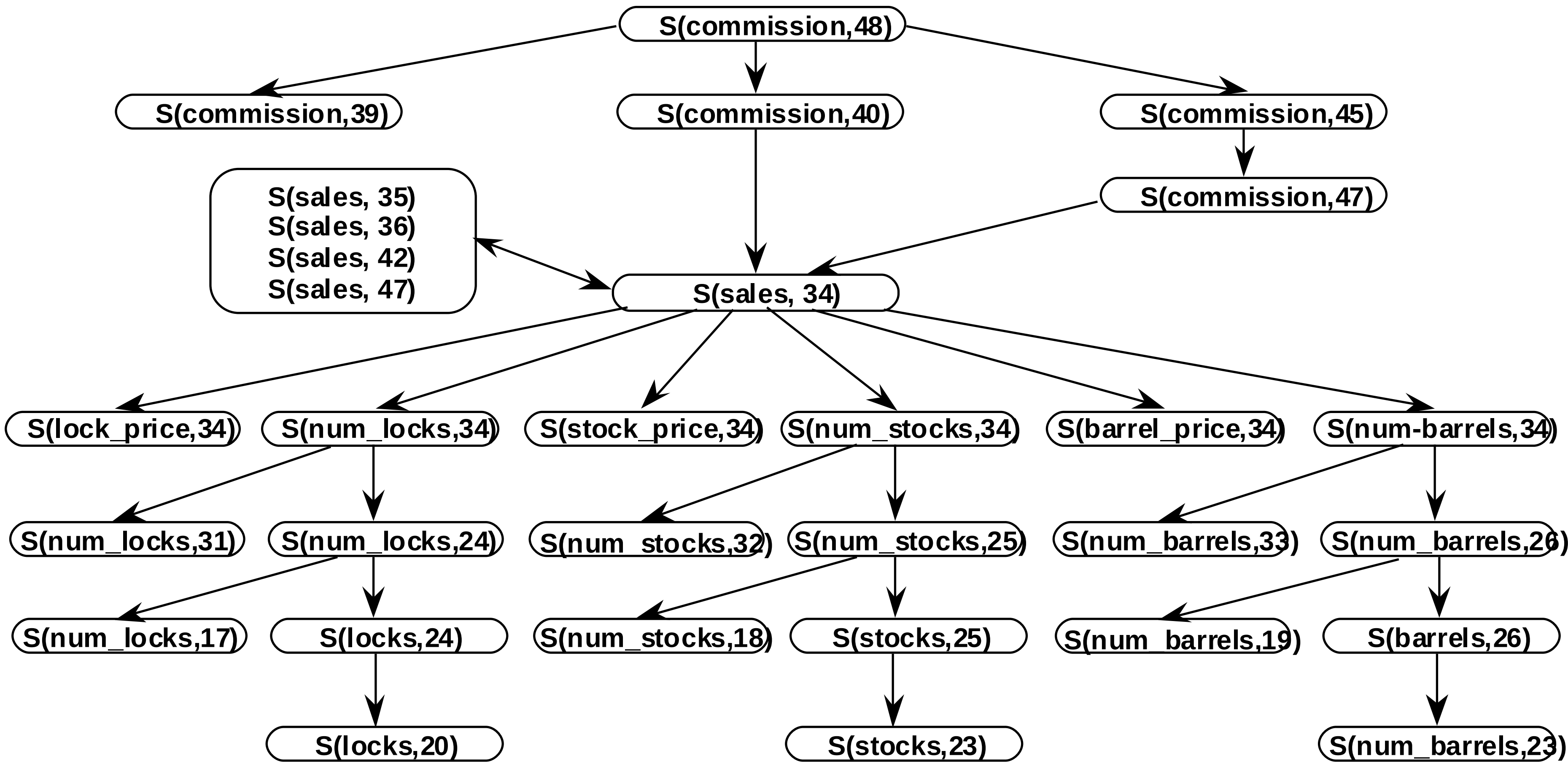
S39: S(commission, 50) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 50}

S40: S(commission, 51) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 41, 42, 43, 47, 48, 5

$$\begin{tikzcd}
S_{34} & & \\
S_{35} \arrow[u] & S_{37} & \\
S_{36} \arrow[u] & S_{38} \arrow[u] & S_{39} \\
 & S_{40} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Equivalent Slices	Test Objective
1, 2, 3	salesman read and written correctly
4	locks read correctly
5, 6	locks sentinel correct
7	additional locks read correctly
8, 9 (10, 11)	iterative read of stocks correct
(8, 9) 10, 11	iterative read of barrels correct
12	num_locks initialized correctly
13, 14, 15	num_locks computed correctly
16	num_stocks initialized correctly
17, 18, 19	num_stocks computed correctly
20	num_barrels initialized correctly
21, 22, 23	num_barrels computed correctly
24	lock_price constant definition correct
25	stock_price constant definition correct
26	barrel_price constant definition correct
27, 28, 29, 30, 31, 32, 33	sales computed correctly
34	commission on first 1000 correct for sales > 1800
35	commission on next 800 correct for sales > 1800
36	commission on excess over 1800 correct for sales > 1800
37	commission on first 1000 correct for 1000 <sales <1800
38	commission on excess over 1000 correct for 1000 <sales <1800
39	commission on sales < 1000 correct
40	commission written correctly

Lattice of Slices



When should testing stop?

1. when you run out of time.
2. when continued testing causes no new failures.
3. when continued testing reveals no new faults.
4. when you can't think of any new test cases.
5. when you reach a point of diminishing returns.
6. when mandated coverage has been attained.
7. when all faults have been removed.

Slice-Based Testing Definitions:

- Given a program P, and a program graph G(P) in which statements and statement fragments are numbered, and a set V of variables in P, the ***slice on the variable set V at statement fragment n***, written S(V,n), is the set node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n
- The idea of slices is to separate a program into components that have some useful meaning
- We will include CONST declarations in slices
- Five forms of usage nodes
    - P-use (used in a predicate (decision))
    - C-use (used in computation)
    - O-use (used for output, e.g. writeln())
    - L-use (used for location, e.g. pointers)
    - I-use (iteration, e.g. internal counters)
- Two forms of definition nodes
    - I-def (defined by input, e.g. readln())
    - A-def (defined by assignment)
- For now, we presume that the slice S(V,n) is a slice on one variable, that is, the set V consists of a single variable, v
- If statement fragment n (in S(V,n)) is a defining node for v, then n is included in the slice
- If statement fragment n (in S(V,n)) is a usage node for v, then n is not included in the slice
- P-uses and C-uses of other variables are included to the extent that their execution affects the value of the variable v
- O-use, L-use, and I-use nodes are excluded from slices
- Consider making slices compliable