UNIT II

Control Flow Graphs and Path Testing

Introduction

Path Testing Definition

A family of structural test techniques based on judiciously selecting a set of test paths through the programs.

- ✓ **Goal:** Pick enough paths to assure that every source statement is executed at least once.
- ✓ It is a measure of thoroughness of **code coverage.**
- ✓ It is used most for unit testing on new software.
- ✓ Its effectiveness reduces as the software size increases.
- ✓ Path testing concepts are used **in** and **along** with other testing techniques

Code Coverage: During unit testing: # stmts executed at least once / total # stmts

Control Flow Graphs and Path Testing

Path Testing contd..

Assumptions:

- Software takes a different path than intended due to some error.
- Specifications are correct and achievable.
- Data definition & access are correct

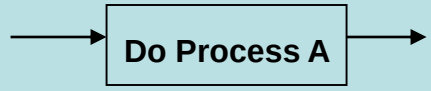
Observations

- Structured programming languages need less of path testing.
- Assembly language, Cobol, Fortran, Basic & similar languages make path testing necessary.

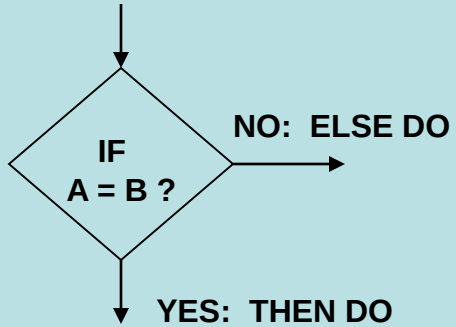
Control Flow Graph

A simplified, abstract, and graphical representation of a program's control structure using process blocks, decisions and junctions.

Process Block

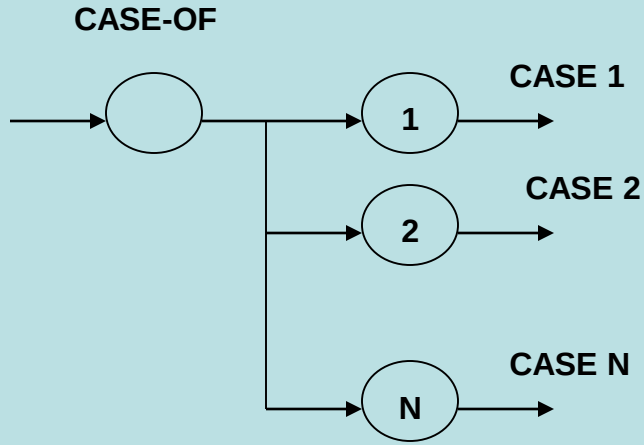


Decisions



Junctions

Case Statement



Control Flow Graph Elements

Control Flow Graph Elements:

Process Block:

- A sequence of program statements uninterrupted by decisions or junctions with a single entry and single exit.

Junction:

- A point in the program where control flow can merge (into a node of the graph)
- Examples: target of GOTO, Jump, Continue

Decisions:

- A program point at which the control flow can diverge *(based on evaluation of a condition)*.
- Examples: IF stmt. Conditional branch and Jump instruction.

Case Statements:

- A Multi-way branch or decision.
- Examples: In assembly language: jump addresses table, Multiple GOTOs, Case/Switch

- For test design, Case statement and decision are similar.

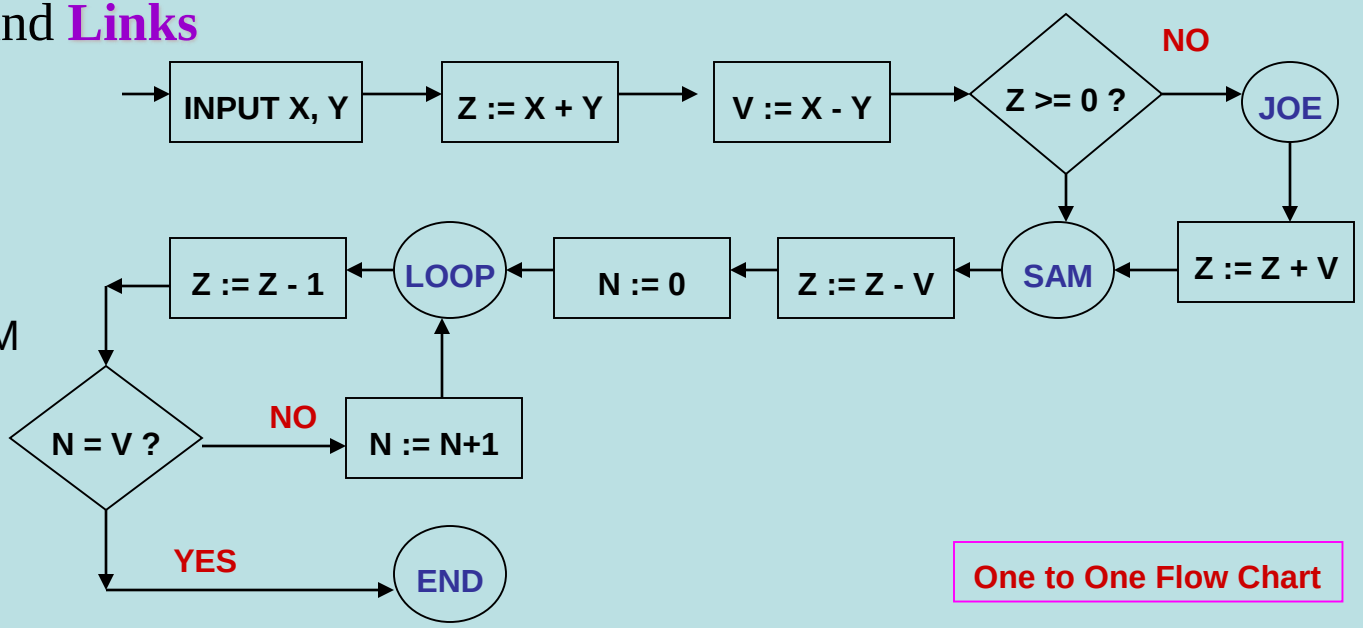
Control Flow Graph Vs Flow Charts

Control Flow Graph	Flow Chart
Compact representation of the program	Usually a multi-page description
Focuses on Inputs, Outputs, and the control flow into and out of the block.	Focuses on the process steps inside
Inside details of a process block are not shown	Every part of the process block are drawn

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```

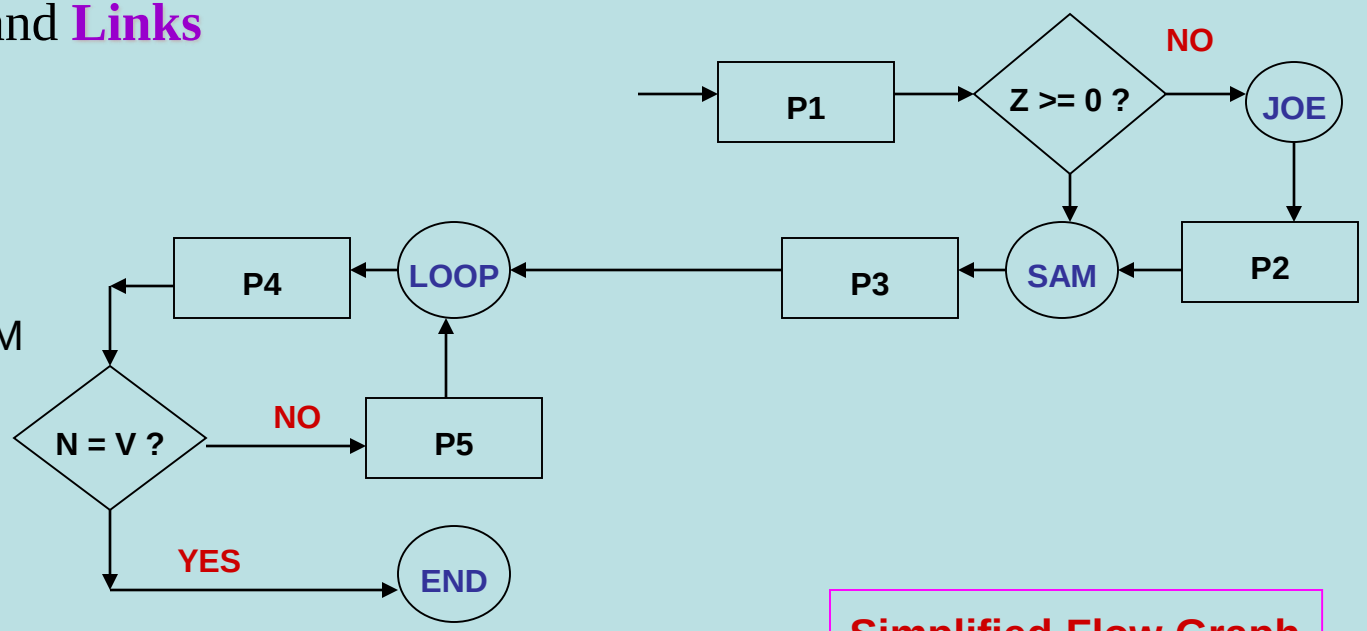


One to One Flow Chart

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```



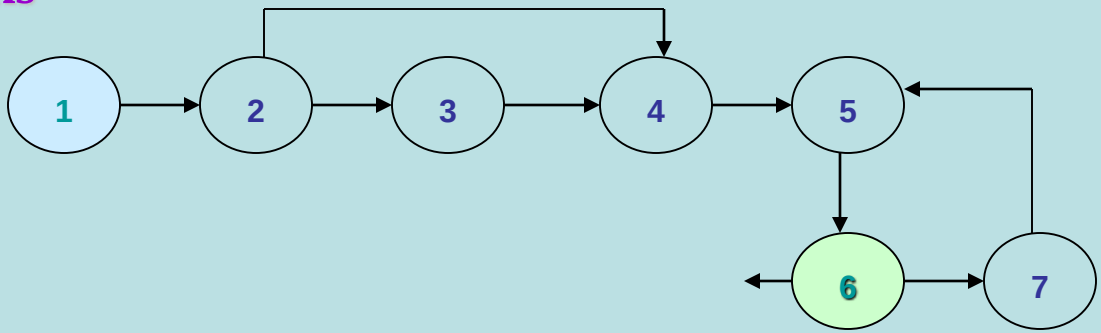
Simplified Flow Graph

Control Flow Graphs and Path Testing

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

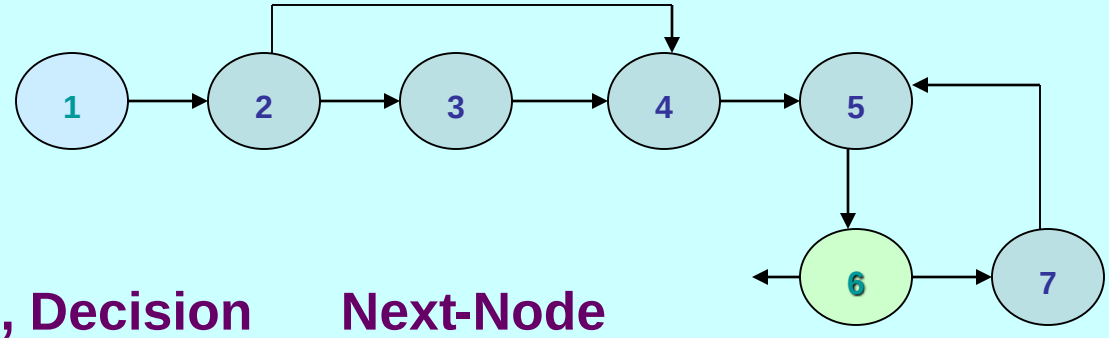
```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```



Simplified Flow Graph

Control Flow Graphs and Path Testing

Linked List Notation of a Control Flow Graph



Node	Processing, label, Decision	Next-Node
1	(BEGIN; INPUT X, Y; Z := X+Y ; V := X-Y)	: 2
2	(Z >= 0 ?)	: 4 (TRUE)
		: 3 (FALSE)
3	(JOE: Z := Z + V)	: 4
4	(SAM: Z := Z – V; N := 0)	: 5
5	(LOOP; Z := Z -1)	: 6
6	(N = V ?)	: 7 (FALSE)
		: END (TRUE)
7	(N := N + 1)	: 5

Path Testing Concepts

1. **Path** is a sequence of statements starting at an entry, junction or decision and ending at another, or possibly the same junction or decision or an exit point.

   Link is a single process (*block*) in between two nodes.

   Node is a junction or decision.

   Segment is a sequence of links. A path consists of many segments.

   Path segment is a succession of consecutive links that belongs to the same path. (3,4,5)

   Length of a path is measured by # of links in the path or # of nodes traversed.

   Name of a path is the set of the names of the nodes along the path. (1,2,3 4,5, 6)
   (1,2,3,4, 5,6,7, 5,6,7, 5,6)

   Path-Testing Path is an “entry to exit” path through a processing block.

2. Entry / Exit for a routines, process blocks and nodes.

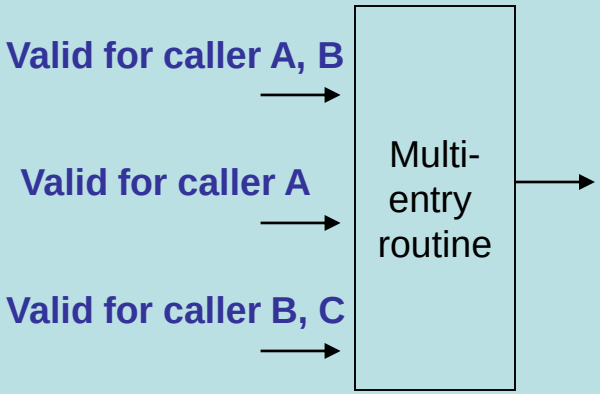
Single entry and single exit routines are preferable.

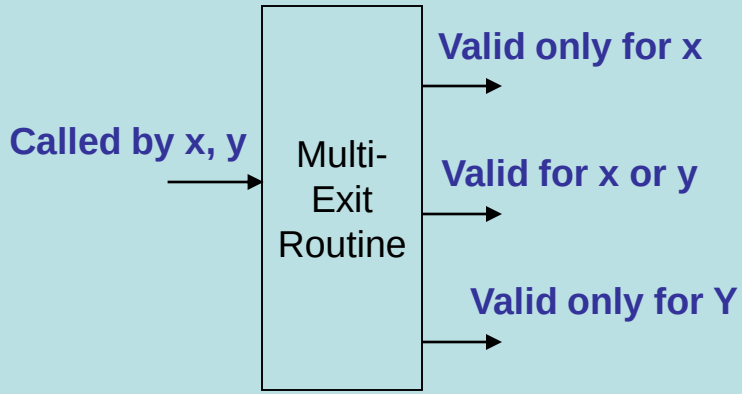
Path Testing Concepts..

Multi-entry / Multi-exit routines: (ill-formed)

- A Weak approach: Hence, convert it to single-entry / single-exit routine.
- Integration issues:

  Large # of inter-process interfaces. Creates problem in Integration.
  More # test cases and also a formal treatment is more difficult.



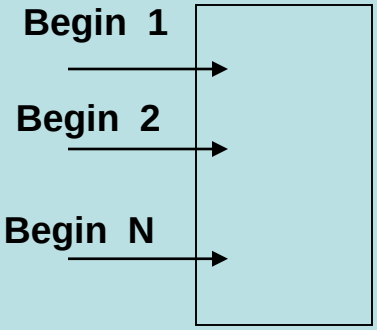


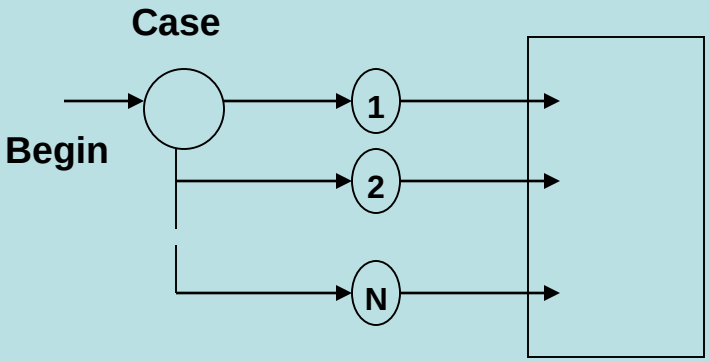
- Theoretical and tools based issues
  - A good formal basis does not exist.
  - Tools may fail to generate important test cases.

Path Testing Concepts contd..

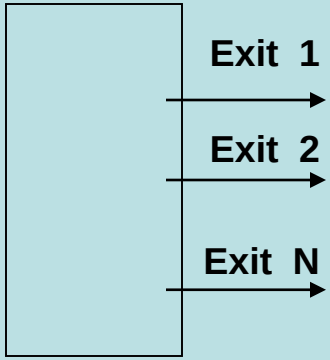
Convert a multi-entry / exit routine to a single entry / exit routine:

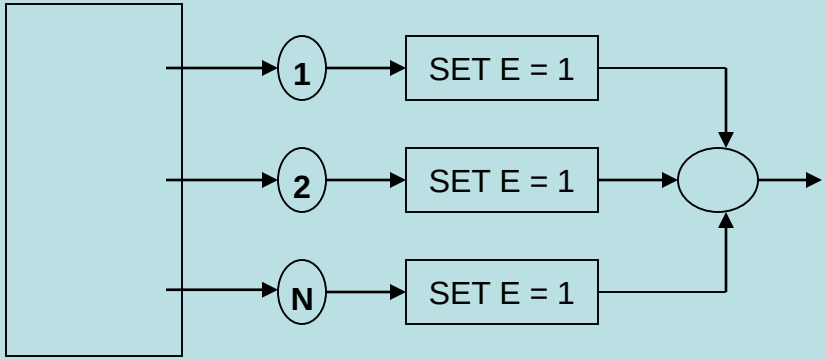
- Use an entry parameter and a case statement at the entry => single-entry





- Merge all exits to Single-exit point after setting one exit parameter to a value.





Path Testing Concepts

3. Fundamental Path Selection Criteria

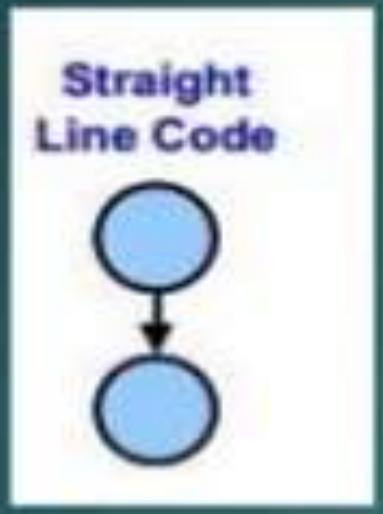
A minimal set of paths to be able to do complete testing.

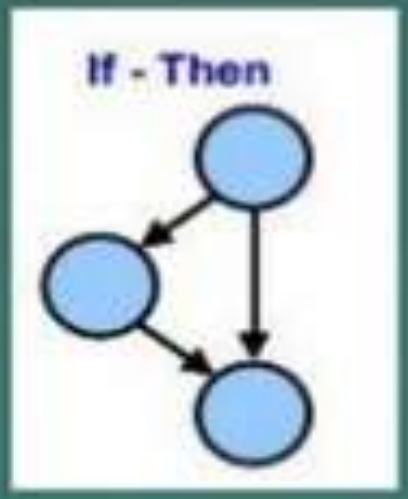
- **Complete Path Testing prescriptions:**
  1. Exercise every path from entry to exit.
  2. Exercise every statement or instruction at least once.
  3. Exercise every branch and case statement in each direction, at least once.

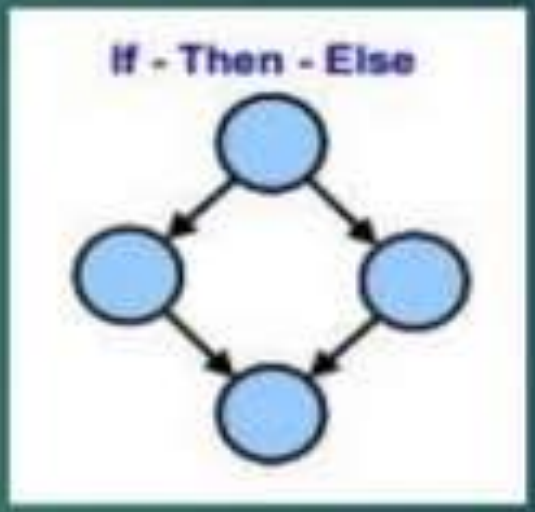
Basic path testing approach

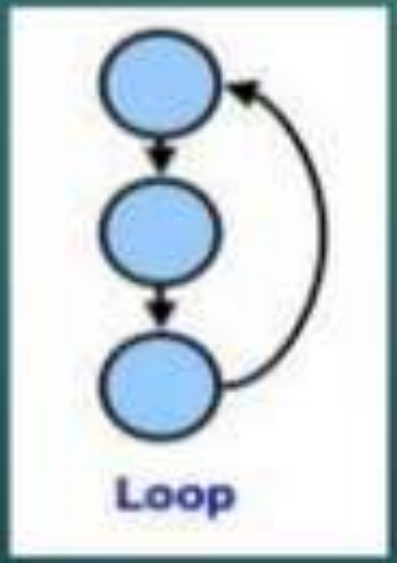
- Step 1: Draw a control flow graph.
- Step 2: Determine Cyclomatic complexity.
- Step 3: Find a basis set of paths.
- Step 4: Generate test cases for each path.

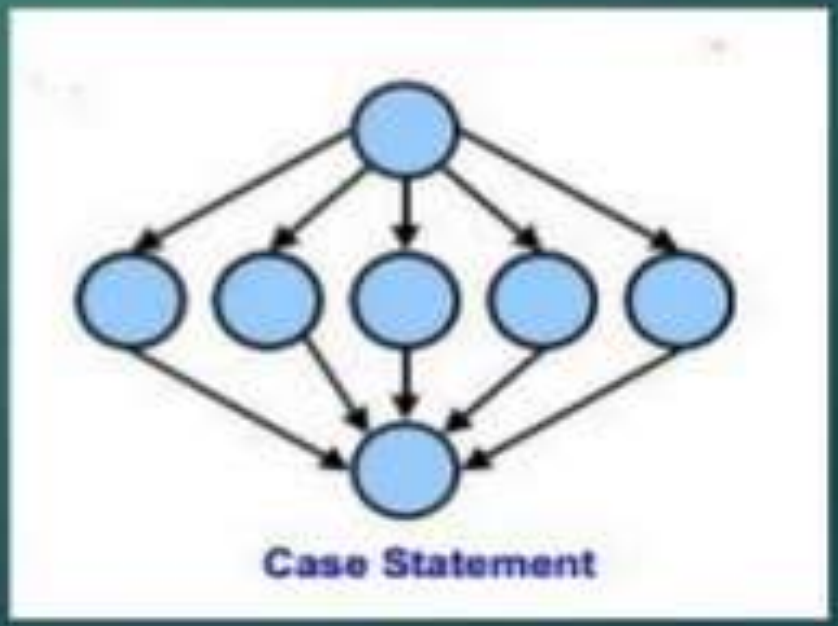
Basic control flow graph structures:





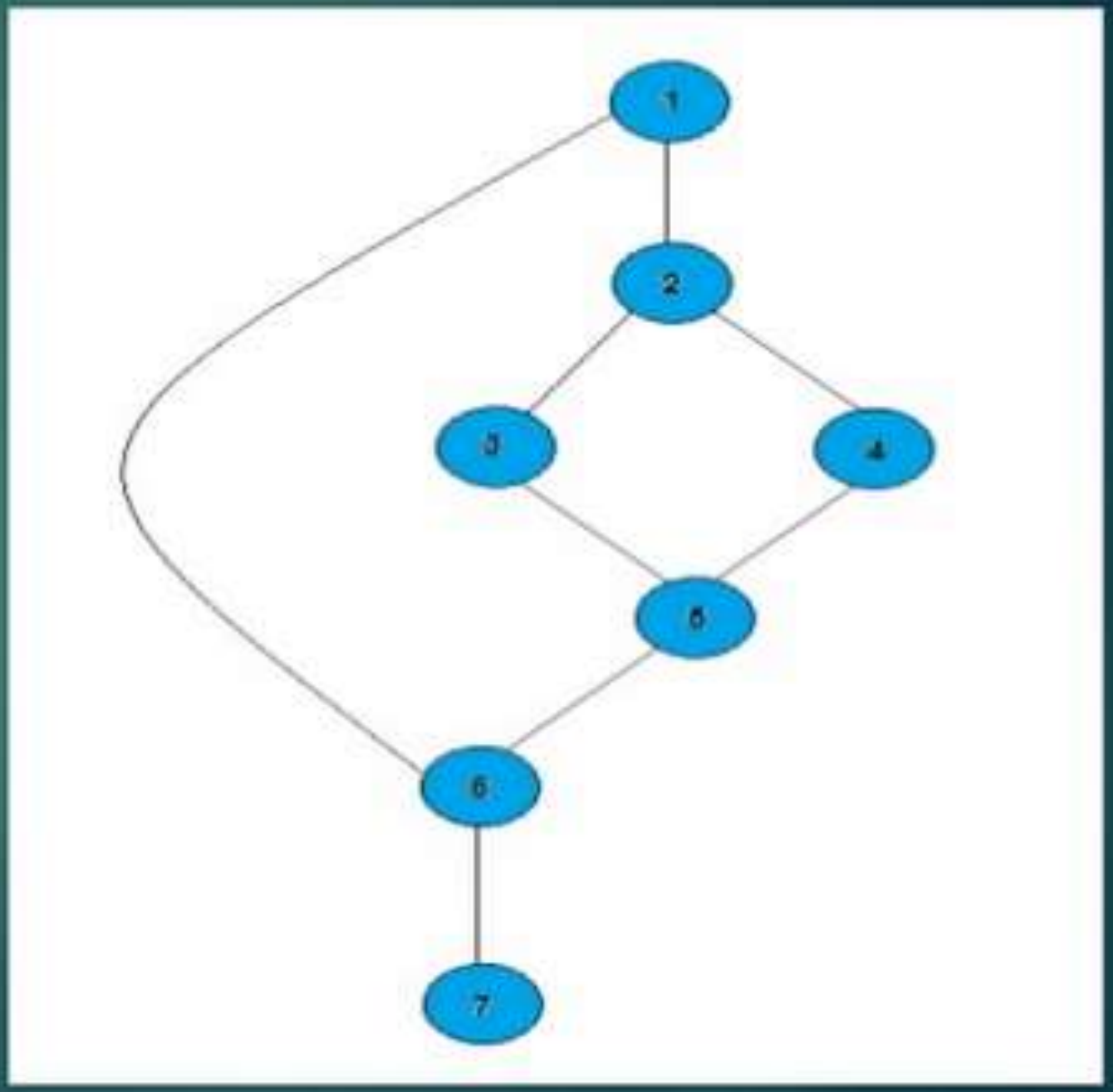






Draw a control flow graph

- 1: IF A = 100
- 2: THEN IF B > C
- 3: THEN A = B
- 4: ELSEA= C
- 5: ENDIF
- 6: ENDIF
- 7: Print A



Determine Cyclomatic complexity

There are several methods:

1. Cyclomatic complexity = edges - nodes + 2p

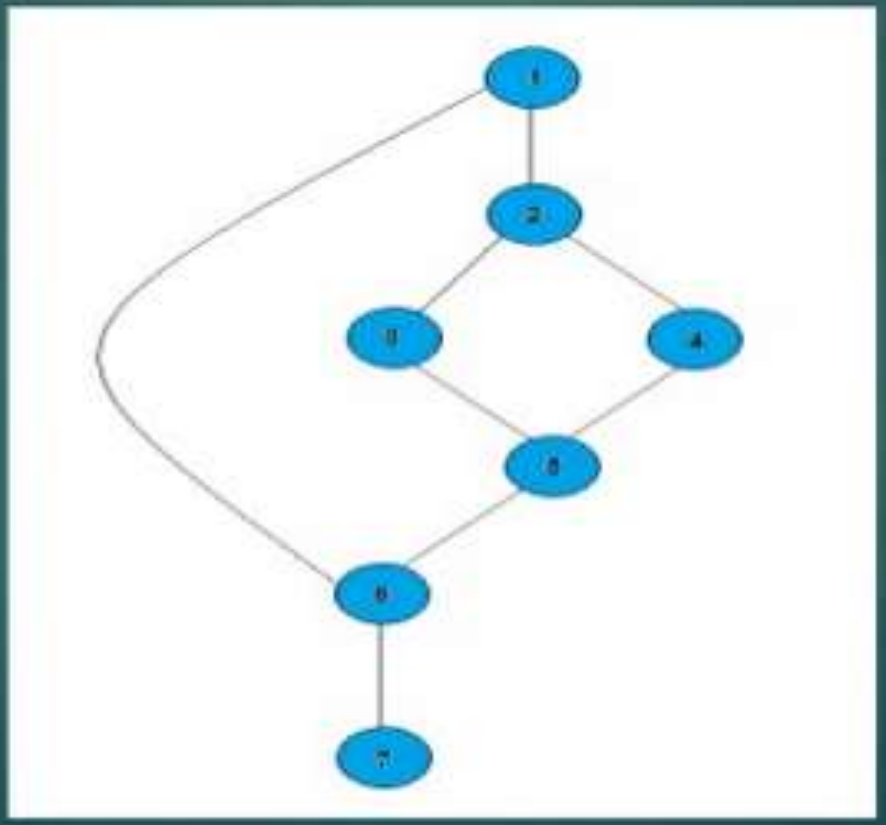
2. Cyclomatic complexity= Number of Predicate Nodes + 1

3. Cyclomatic complexity =number of regions in the control flow graph

Determine Cyclomatic complexity

Cyclomatic complexity = edges - nodes + 2p

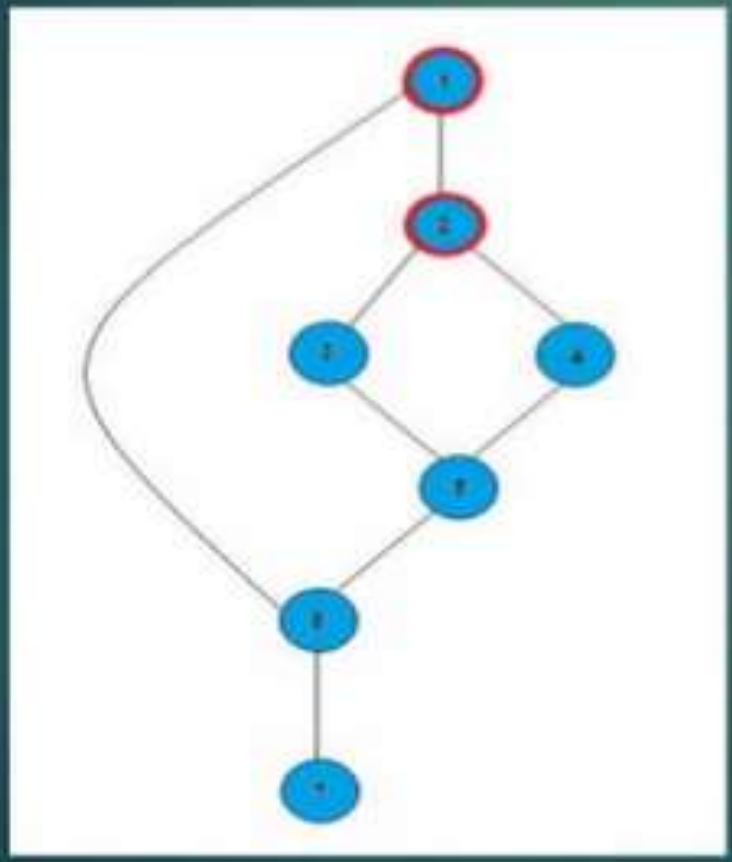
- p = number of unconnected parts of the graph.



Cyclomatic complexity = 8-7+ 2*1= 3.

Determine Cyclomatic complexity

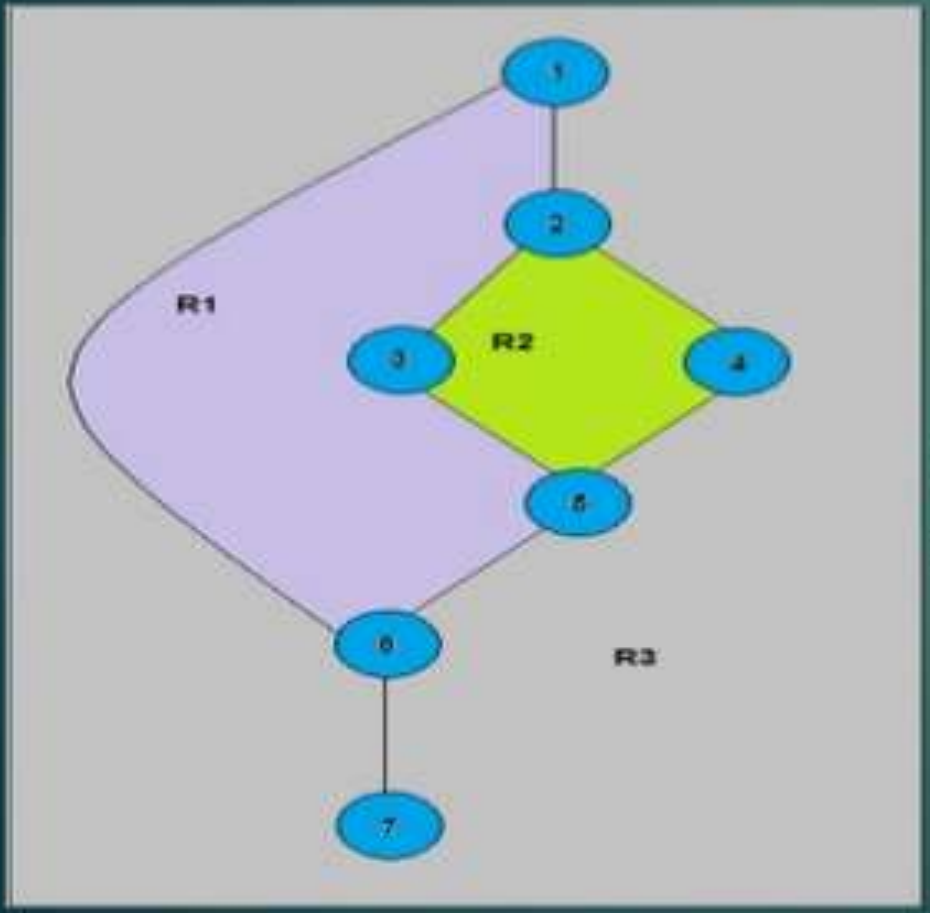
Cyclomatic complexity= Number of Predicate Nodes + 1



Cyclomatic complexity = 2+1= 3.

Determine Cyclomatic complexity

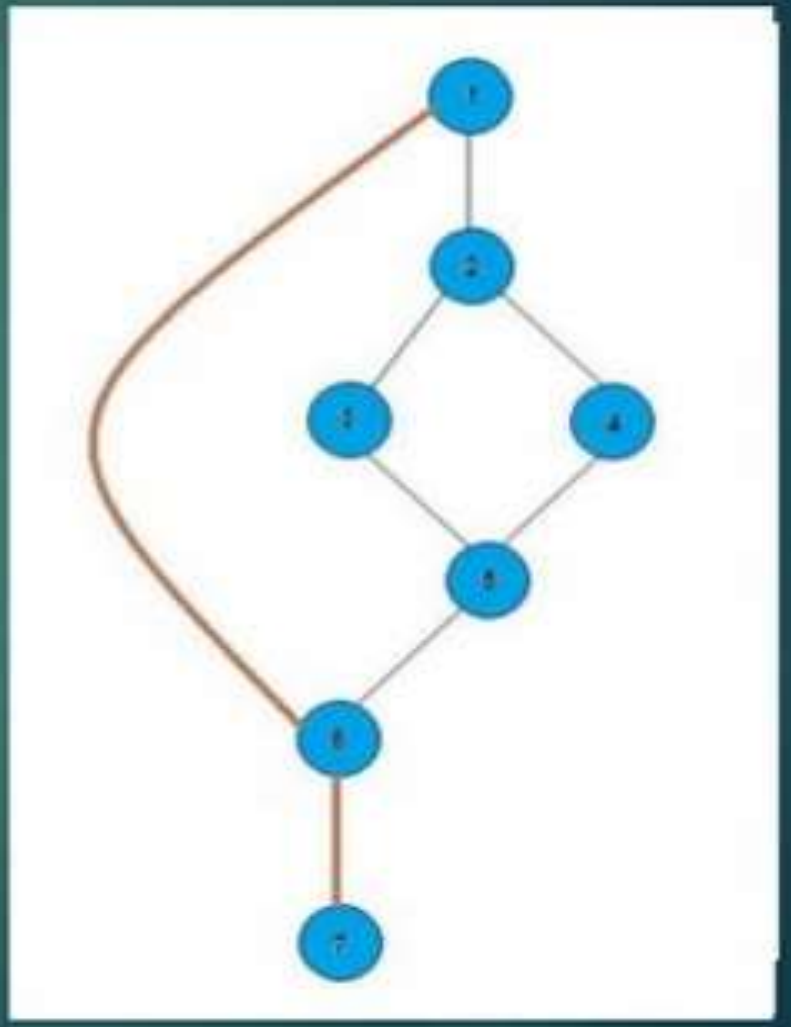
Cyclomatic complexity =number of regions in the control flow graph



Cyclomatic complexity = 3

Find a basis set of paths

- **Path 1**: 1, 2, 3, 5, 6, 7.
- **Path 2**: 1, 2, 4, 5, 6, 7.
- **Path 3**: 1, 6, 7.



Path Testing Concepts

Path Testing Criteria :

1. **Path Testing (P_∞):**

   Execute all possible control flow paths thru the program; but typically restricted to entry-exit paths.

   Implies 100% path coverage.

2. **Statement Testing (P_1)** :

   Execute all statements in the program at least once under the some test.
   100% statement coverage => 100% node coverage.
   Denoted by **C1**
   C1 is a minimum testing requirement in the IEEE unit test standard: ANSI 87B.

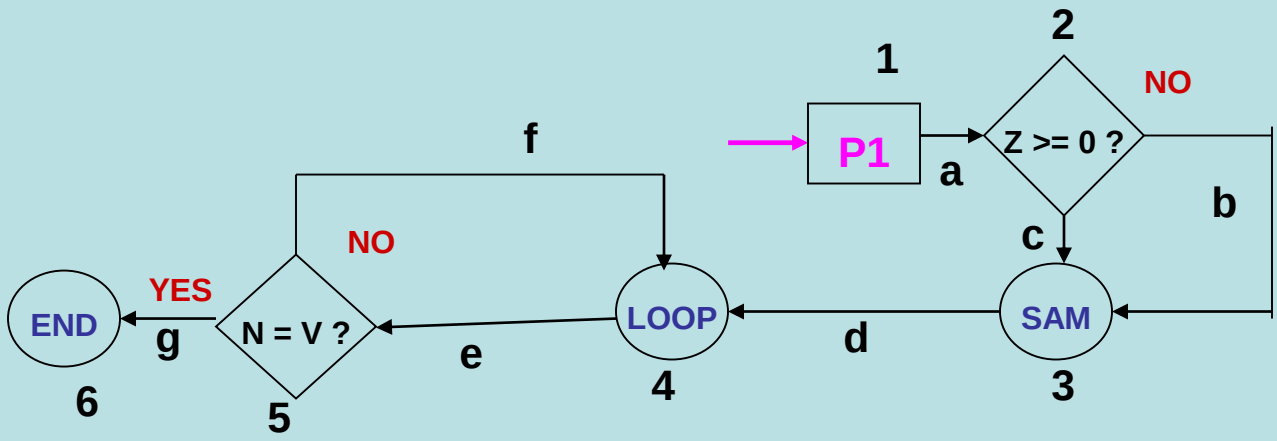
3. **Branch Testing (P_2)** :

   Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
   Denoted by **C2**
   Objective: 100% branch coverage and 100% Link coverage.

   For well structured software, branch testing & coverage include statement coverage

Control Flow Graphs and Path Testing

Picking enough (the fewest) paths for achieving C1+C2



1. Does every decision have Y & N (C2)?
2. Are call cases of case statement marked (C2)?
3. Is every three way branch covered (C2)?
4. Is every link covered at least once (C1)?

Make small changes in the path changing only 1 link or node at a time.

Paths	Decisions		Process-link						
	2	5	a	b	c	d	e	f	g
abdeg	No	Y	Y	Y		Y	Y		Y
acdeg	Y	Y	Y		Y	Y	Y		Y
abdefeg	No	Y	Y	Y		Y	Y	Y	Y
acdefeg	Y	No	Y		Y	Y	Y	Y	Y

Revised path selection Rules

1. Pick the simplest and functionally sensible entry/exit path
2. Pick additional paths as small variations from previous paths. (pick those with no loops, shorter paths, simple and meaningful)
3. Pick additional paths but without an obvious functional meaning (only to achieve C1+C2 coverage).
4. Be comfortable with the chosen paths to achieve C1+C2

Effectiveness of Path Testing

- Path testing catches ~65% of Unit Test Bugs ie., ~35% of all bugs.
- More effective for unstructured than structured software.

- **Limitations**
  - Path testing may not do expected coverage if bugs occur.
  - Path testing may not reveal totally wrong or missing functions.
  - Unit-level path testing may not catch interface errors among routines.
  - Data base and data flow errors may not be caught.
  - Unit-level path testing cannot reveal bugs in a routine due to another.
  - Not all initialization errors are caught by path testing.
  - Specification errors cannot be caught.

- **A lot of work**
  - Creating flow graph, selecting paths for coverage, finding input data values to force these paths, setting up loop cases & combinations.
- Careful, systematic, test design will catch as many bugs as the act of testing.

Test design process at all levels at least as effective at catching bugs as is running the test designed by that process.

Predicates, Predicate Expressions

Path

- A sequence of process links (& nodes)

Predicate

- The logical function evaluated at a decision : True or False. *(Binary , boolean)*

Compound Predicate

- Two or more predicates combined with AND, OR etc.

Path Predicate

- Every path corresponds to a succession of True/False values for the predicates traversed on that path.

- A predicate associated with a path.
  " X > 0 is True " AND "W is either negative or equal to 122" is True

Predicates, Predicate Expressions…

Predicate Interpretation

- The symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation**.

  An input vector is a set of inputs to a routine arranged as a one dimensional array.

- Predicate interpretation may or may not depend on the path.

- Path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

Predicates, Predicate Expressions…

Process Dependency

- An **input variable** is **independent** of the processing if its value does not change as a result of processing.
- An **input variable** is **process dependent** if its value changes as a result of processing.
- A **predicate** is **process dependent** if its truth value can change as a result of processing.
- A **predicate** is **process independent** if its truth value does not change as a result of processing.
- Process dependence of a predicate doesn't follow from process dependence of variables
- If all the input variables (on which a predicate is based) are process independent, then **predicate is process independent**.

Predicates, Predicate Expressions…

Correlation

- Two **input variables** are **correlated** if every combination of their values cannot be specified independently.
- **Variables** whose values can be specified independently without restriction are **uncorrelated**.
- A pair of predicates whose outcomes depend on one or more variables in common are **correlated predicates**.
- Every path through a routine is **achievable** only if all predicates in that routine are **uncorrelated**.
- If a routine has a loop, then at least one decision's predicate must be process dependent. Otherwise, there is an input value which the routine loops indefinitely.

Predicates, Predicate Expressions…

Path Predicate Expression

- Every selected path leads to an associated boolean expression, called the **path predicate expression**, which characterizes the input values (if any) that will cause that path to be traversed.
- Select an entry/exit path. Write down un-interpreted predicates for the decisions along the path. If there are iterations, note also the value of loop-control variable for that pass. Converting these into predicates that contain only input variables, we get a set of boolean expressions called path predicate expression.
- Example (inputs being numerical values):

If $X_5 > 0$ .OR. $X_6 < 0$ then

$X_1 + 3X_2 + 17 >= 0$
$X_3 = 17$
$X_4 – X_1 >= 14\ X_2$

Predicates, Predicate Expressions…

A: $X_5 > 0$
B: $X_1 + 3X_2 + 17 >= 0$
C: $X_3 = 17$
D: $X_4 – X_1 >= 14\ X_2$

E: $X_6 < 0$
F: $X_1 + 3X_2 + 17 >= 0$
G: $X_3 = 17$
H: $X_4 – X_1 >= 14\ X_2$

Converting into the predicate expression form:

$$A\ B\ C\ D + E\ B\ C\ D \Rightarrow (A + E)\ B\ C\ D$$

If we take the alternative path for the expression: D then

$$(A + E)\ B\ C\ \overline{D}$$

Predicates, Predicate Expressions…

Predicate Coverage:

- Look at **examples** & possibility of bugs: A B C D A + B + C + D
  - Due to semantics of the evaluation of logic expressions in the languages, the entire expression may not be always evaluated.
  - A bug may not be detected.
  - A wrong path may be taken if there is a bug.
- Realize that on our achieving C2, the program could still hide some control flow bugs.
- **Predicate coverage:**
  - If all possible combinations of truth values corresponding to selected path have been explored under some test, we say predicate coverage has been achieved.
  - Stronger than branch coverage.
  - If all possible combinations of all predicates under all interpretations are covered, we have the equivalent of total path testing.

4. Testing of Paths involving loops

Bugs in iterative statements apparently are not discovered by C1+C2.
But by testing at the boundaries of loop variable.

Types of Iterative statements

1. **Single loop statement.**
2. **Nested loops.**
3. **Concatenated Loops.**
4. **Horrible Loops**

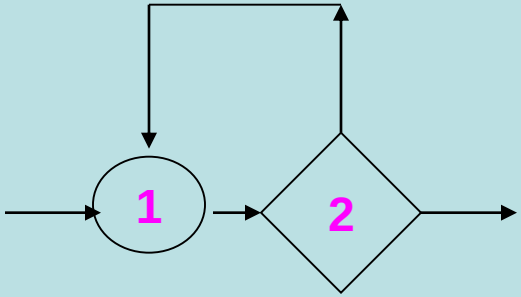
Let us denote the Minimum # of iterations by n_{min}
the Maximum # of iterations by n_{max}
the value of loop control variable by V
the #of test cases by T
the # of iterations carried out by n

- Later, we analyze the Loop-Testing times

Testing of path involving loops…

1. Testing a Single Loop Statement (three cases)

Case 1. $n_{min} = 0$, $n_{max} = N$, no excluded values

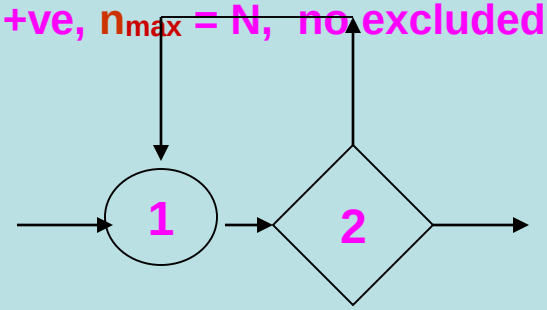


1. Bypass the loop.
   If you can't, there is a bug, $n_{min} \neq 0$ or a wrong case.
2. Could the value of loop (control) variable V be negative?
   could it appear to specify a –ve n ?
3. Try one pass through the loop statement: n = 1
4. Try two passes through the loop statement: n = 2
   To detect initialization data flow anomalies:
   Variable defined & not used in the loop, or
   Initialized in the loop & used outside the loop.
5. Try n = typical number of iterations : $n_{min} < n < n_{max}$
6. Try n = n_{max} -1

7. Try n = n_{max}
8. Try n = n_{max} + 1.
   What prevents V (& n) from having this value?
   What happens if it is forced?

Testing of path involving loops…

Case 2.Single loop, nonzero minimum,No excluded values(**n_{min} = +ve, n_{max} = N, no excluded values)**



1. Try **n_{min} - 1**
   Could the value of loop (control) variable V be < **n_{min}**?
   What prevents that ?
2. Try **n_{min}**
3. Try **n_{min} + 1**

4. Once, unless covered by a previous test.
5. Twice, unless covered by a previous test.

4. Try n = typical number of iterations : **n_{min}** < n < **n_{max}**
5. Try n = **n_{max}** -1

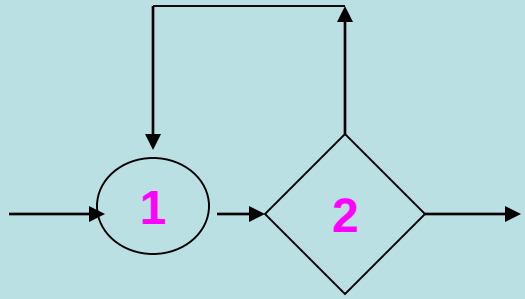
6. Try n = **n_{max}**
7. Try n = **n_{max}** + 1.
   What prevents V (& n) from having this value?
   What happens if it is forced?

Ref Boris Beizer

Note: only a case of no iterations, n = 0 is not there!

Path Testing Concepts…

Case 3. Single loop with excluded values



1. Treat this as single loops with excluded values as two sets.

2. Example:

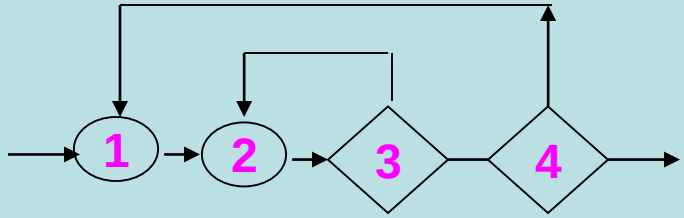
V = 1 to 20 excluding 7,8,9 and10

Test cases to attempt are for:

V = 0,1,2,4,6,7 and V = 10,11,15,19,20,21

(underlined cased are not supposed to work)

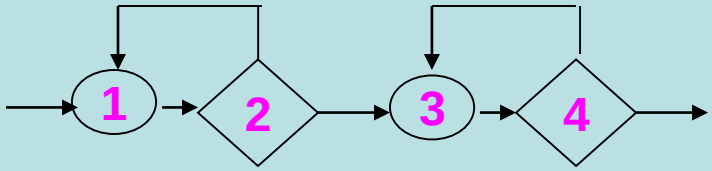
Testing of path involving loops…



- Compromise on # test cases for processing time.
- Expand tests for solving potential problems associated with initialization of variables and with excluded combinations and ranges.
- Apply Huang's twice thorough theorem to catch data initialization problems.

Testing of path involving loops…

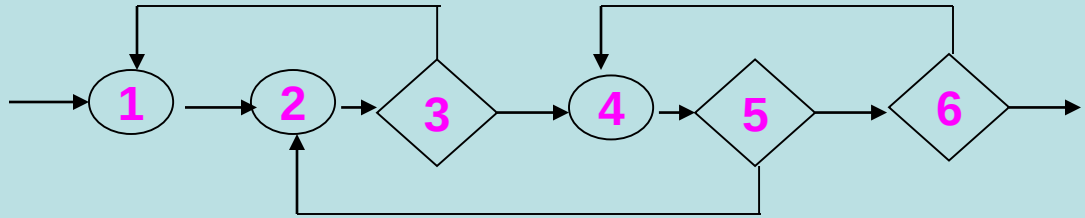
3. Testing Concatenated Loop Statements



- Two loops are concatenated if it's possible to reach one after exiting the other while still on the path from entrance to exit.
- If these are independent of each other, treat them as independent loops.
- If their iteration values are inter-dependent & these are same path, treat these like a nested loop.
- Processing times are additive.

Testing of path involving loops…

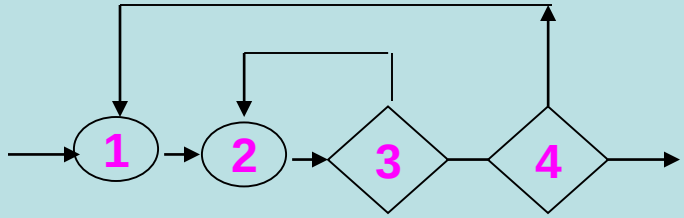
4.Testing Horrible Loops



- Avoid these.
- Even after applying some techniques of paths, resulting test cases not definitive.
- Too many test cases.
- Thinking required to check end points etc. is unique for each program.
- Jumps in & out of loops and intersecting loops etc, makes test case selection an ugly task.
- etc. etc.

Testing of path involving loops…

2. Testing a Nested Loop Statement



- Multiplying # of tests for each nested loop => very large # of tests
- A test selection technique:

1. Start at the inner-most loop. Set all outer-loops to Min iteration parameter values: **V_{min}**.
2. Test the **V_{min}, V_{min} + 1, typical V, V_{max} - 1, V_{max}** for the inner-most loop. Hold the outer-loops to **V_{min}**. Expand tests are required for out-of-range & excluded values.
3. If you have done with outer most loop, Go To step 5. Else, move out one loop and do step 2 with all other loops set to **typical values**.
4. Do the five cases for all loops in the nest simultaneously.

- Assignment: check # test cases = 12 for nesting = 2, 16 for 3, 19 for 4.

Testing of path involving loops…

Loop Testing Times

- Longer testing time for all loops if all the extreme cases are to be tested.
- Unreasonably long test execution times indicate bugs in the s/w or specs.

  Case: Testing nested loops with combination of extreme values leads to long test times.

  - Show that it's due to incorrect specs and fix the specs.
  - Prove that combined extreme cases cannot occur in the real world. Cut-off those tests.

  - Put in limits and checks to prevent the combined extreme cases.
  - Test with the extreme-value combinations, but use different numbers.

- The test time problem is solved by rescaling the test limit values.

  - Can be achieved through a separate compile, by patching, by setting parameter values etc..

Testing blindness

- **coming to the right path** – even thru a wrong decision (at a predicate). Due to the interaction of some statements makes a buggy predicate work, and the bug is not detected by the selected input values.

- **calculating wrong number of tests** at a predicate by ignoring the # of paths to arrive at it.

- **Cannot be detected by path testing and need other strategies**

Testing blindness

- **Assignment blinding**: A buggy Predicate seems to work correctly as the specific value chosen in an assignment statement works with both the correct & buggy predicate.

Correct	Buggy	
X := 7	X := 7	
IF Y > 0 THEN …	IF X + Y > 0 THEN …	**(check for Y=1)**

- **Equality blinding**:
  - When the path selected by a prior predicate results in a value that works both for the correct & buggy predicate.

Correct	Buggy	
IF Y = 2 THEN …	IF Y = 2 THEN ..	
IF X + Y > 3 THEN …	IF X > 1 THEN …	**(check for any X>1)**

- **Self-blinding**
  - When a buggy predicate is a multiple of the correct one and the result is indistinguishable along that path.

Correct	Buggy	
X := A	X := A	
IF X - 1 > 0 THEN …	IF X + A -2 > 0 THEN …	**(check for any X,A)**

Achievable Paths

1. Objective is to select & test just enough paths to achieve a satisfactory notion of test completeness such as C1 + C2.

2. Extract the program's control flow graph & select a set of tentative covering paths.

3. For a path in that set, interpret the predicates.

4. Trace the path through, multiplying the individual compound predicates to achieve a boolean expression. Example: (A + BC) (D + E)

5. Multiply & obtain **sum-of-products** form of the **path predicate expression**:
   AD + AE + BCD + BCE

6. Each product term denotes a set of inequalities that, if solved, will yield an input vector that will drive the routine along the selected path.

7. A set of input values for that path is found when any of the inequality sets is solved.

   A solution found => **path is achievable**. Otherwise the path is **unachievable.**

Application of Path Testing

1. Integration, Coverage, and Paths in Called Components

- Mainly used in Unit testing, especially new software.
- **In an Idealistic bottom-up integration test process** – integrating one component at a time. Use stubs for lower level component (sub-routines), test interfaces and then replace stubs by real subroutines.
- **In reality,** integration proceeds in associated blocks of components. Stubs may be avoided. Need to think about paths inside the subroutine.

To achieve C1 or C2 coverage:

- Predicate interpretation may require us to treat a subroutine as an in-line-code.
- Sensitization becomes more difficult.
- Selected path may be unachievable as the called components' processing may block it.

Weaknesses of Path testing:

- It assumes that effective testing can be done one level at a time without bothering what happens at lower levels.
- predicate coverage problems & blinding.

Implementation & Application of Path Testing

2. Application of path testing to **New Code**

- Do Path Tests for C1 + C2 coverage
- Use the procedure similar to the idealistic bottom-up integration testing, using a mechanized test suite.
- A path blocked or not achievable could mean a bug.
- When a bug occurs the path may be blocked.

Implementation & Application of Path Testing

3. Application of path testing to **Maintenance**

- Path testing is applied first to the modified component.
- Use the procedure similar to the idealistic bottom-up integration testing, but without using stubs.
- Select paths to achieve C2 over the changed code.
- Newer and more effective strategies could emerge to provide coverage in maintenance phase.

Implementation & Application of Path Testing

4. Application of path testing to **Rehosting**

- Path testing with C1 + C2 coverage is a powerful tool for rehosting old software.
- Software is rehosted as it's no more cost effective to support the application environment.
- Use path testing in conjunction with automatic or semiautomatic structural test generators.

Implementation & Application of Path Testing

Application of path testing to **Rehosting..**

Process of path testing during rehosting

- A translator from the old to the new environment is created & tested. Rehosting process is to catch bugs in the translator software.
- A complete C1 + C2 coverage path test suite is created for the old software. Tests are run in the old environment. The outcomes become the specifications for the rehosted software.
- Another translator may be needed to adapt the tests & outcomes to the new environment.
- The cost of the process is high, but it avoids risks associated with rewriting the code.
- Once it runs on new environment, it can be optimized or enhanced for new functionalities (which were not possible in the old environment.)

Data - Flow Testing - Basics

Data - Flow Testing - Basics

- In Data flow testing concentrate on the usage of variable.
- The importance of analyzing the use of variables in programs has been recognized for a long time.
- Variables have been seen as the main areas where a program can be tested structurally.
- Statements where variable receives values
- Statements where values are used

➢ **Anomalies with variables**

  - "A variable that is defined but never used (referenced).
  - A variable that is used but never defined.
  - A variable that is defined twice before it is used."

- Data Flow Testing (DFT) uses Control Flow Graph (CFG) to explore dataflow anomalies.

Data - Flow Testing - Basics

Definition:

- Data-flow testing is a white box testing technique that can be used to detect improper use of data values due to coding errors.
- Data Flow Testing is a family of test strategies based on selecting paths through the program's control flow in order to explore the sequence of events related to the status of data objects.
- Testing focuses on the points at which variables receive values and the points at which these values are used.

Example:

Pick enough paths to assure that every data item has been initialized prior to its use, or that all objects have been used for something.

➢Advantages of Data Flow Testing:

- Data Flow testing helps us to pinpoint any of the following issues:
- A variable that is declared but never used within the program.
- A variable that is used but never declared.
- A variable that is defined multiple times before it is used.
- De-allocating a variable before it is used.

- **Motivation:**
  - it is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test,
  - one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.

DATA FLOW MACHINES: There are two types of data flow machines with different architectures.

(1) Von Neumann machines

(2) Multi-instruction, multi-data machines (MIMD).

(1). Von Neumann Machine Architecture:

- Most computers today are von-neumann machines.
- This architecture features interchangeable storage of instructions and data in the same memory units.

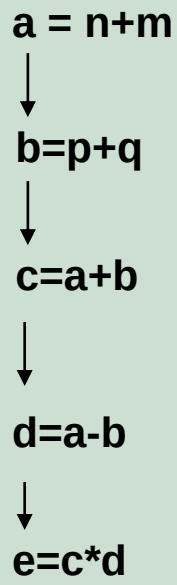
Program Control flow with Von Neumann's paradigm

Given m, n, p, q, find e.

e = (m+n+p+q) * (m+n-p-q)

```
a := m + n
b := p + q
c := a + b
d := a - b

e := c * d
```



Multiple representations of control flow graphs possible.

Data - Flow Testing - Basics

•The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:

1. Fetch instruction from memory
2. Interpret instruction
3. Fetch operands
4. Process or Execute
5. Store result
6. Increment program counter
7. GOTO 1

•**Multi-instruction, Multi-data machines (MIMD) Architecture:**

•These machines can fetch several instructions and objects in parallel.

•They can also do arithmetic and logical operations simultaneously on different data objects.

•The decision of how to sequence them depends on the compiler.

•**Multi-instruction, Multi-data machines (MIMD) Architecture:**

•These machines can fetch several instructions and objects in parallel.

•They can also do arithmetic and logical operations simultaneously on different data objects.

•The decision of how to sequence them depends on the compiler.

Data - Flow Testing - Basics

Data Object State and Usage: Data Objects can be created, killed and used.

They can be used in two distinct ways:

- ✓ In a Calculation
- ✓ As a part of a Control Flow Predicate.

The following symbols denote these possibilities:

- ✓**Defined:** d - defined, created, initialized etc
- ✓**Killed or undefined:** k - killed, undefined, released etc
- ✓**Usage:** u - used for something (c - used in Calculations, p - used in a predicate)

1. **Defined (d):**

- An object is defined explicitly when it appears in a data declaration.
- Or implicitly when it appears on the left hand side of the assignment.
- It is also to be used to mean that a file has been opened.
- A dynamically allocated object has been allocated.
- Something is pushed on to the stack.
- A record written.

2. Killed or Undefined (k):

- An object is killed on undefined when it is released or otherwise made unavailable.
- When its contents are no longer known with certitude (with aboslute certainity / perfectness).
- Release of dynamically allocated objects back to the availability pool.
- Return of records.
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefined A

3. Usage (u):

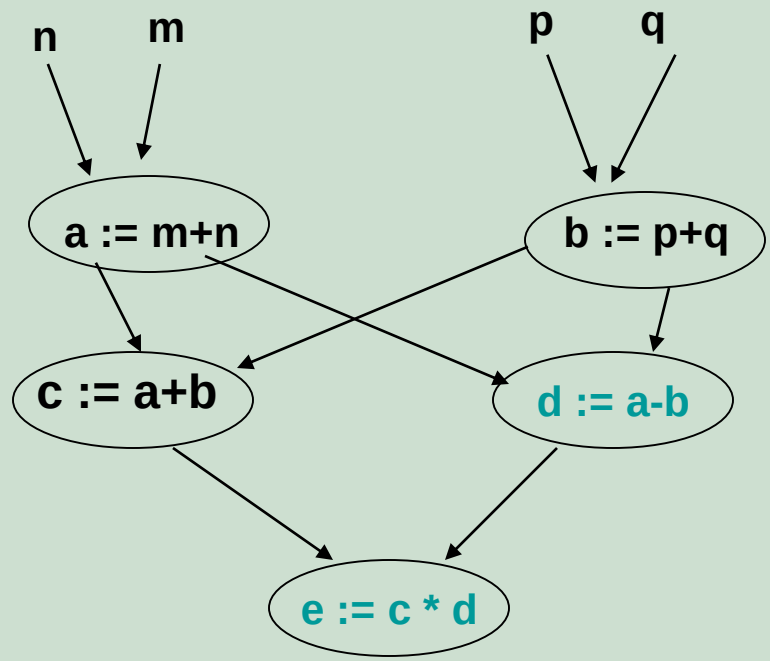
• A variable is used for computation (c) when it appears on the right hand side of an assignment statement.

• A file record is read or written.

•It is used in a Predicate (p) when it appears directly in a predicate.

Program Flow using Data Flow Machines paradigm

```
BEGIN
 PAR DO
        READ m, n, n, p, q
END PAR
PAR DO
        a := m+n
        b := p+q
END PAR
PAR DO
        c := a+b
        d := a-b
END PAR
PAR DO
        e := c * d
END PAR
END
```



The interrelations among the data items remain same.

Actions on data objects

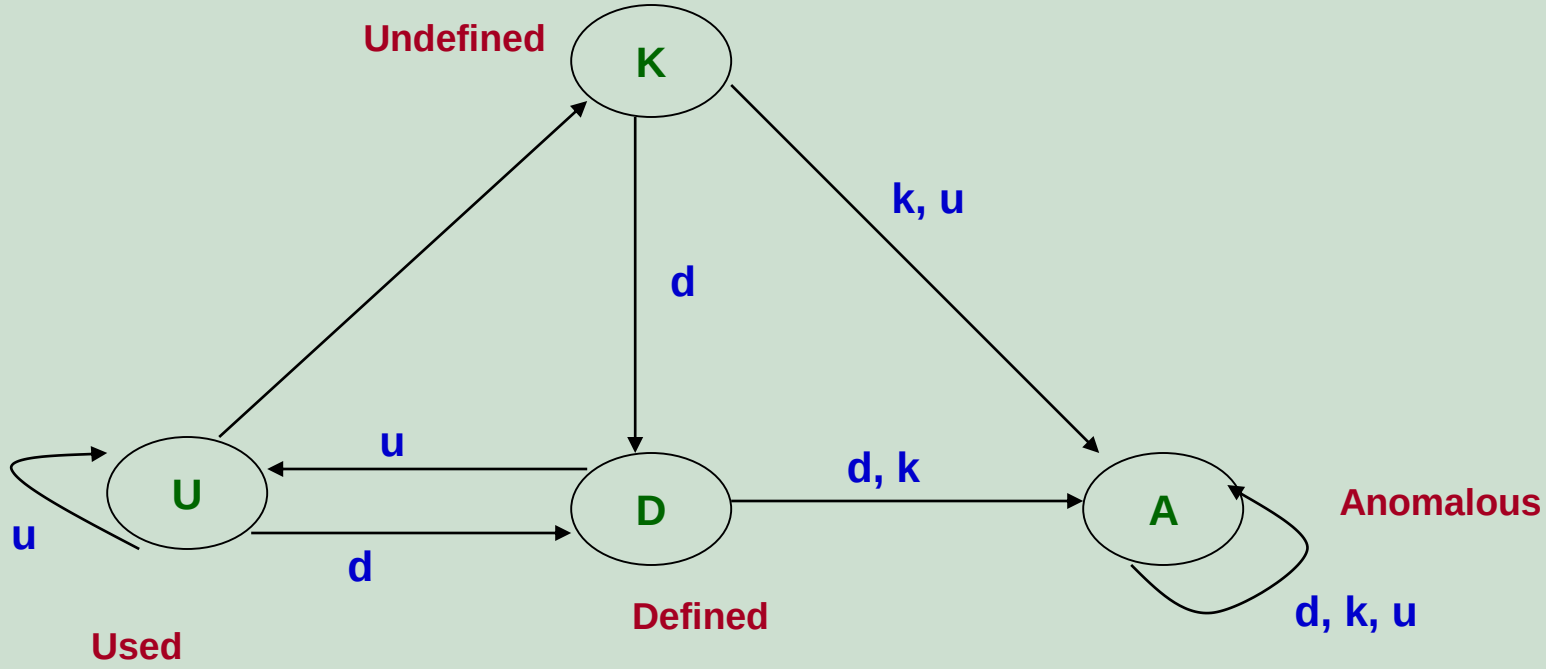
-	no action from START to this point From this point till the EXIT
- d	normal
- u	anomaly
- k	anomaly
k-	normal
u -	normal - possibly an anomaly
d -	possibly anomalous

Data Flow Anomaly State graph

- Data Object State
  - K, D, U, A
- Processing Step
  - k, d, u

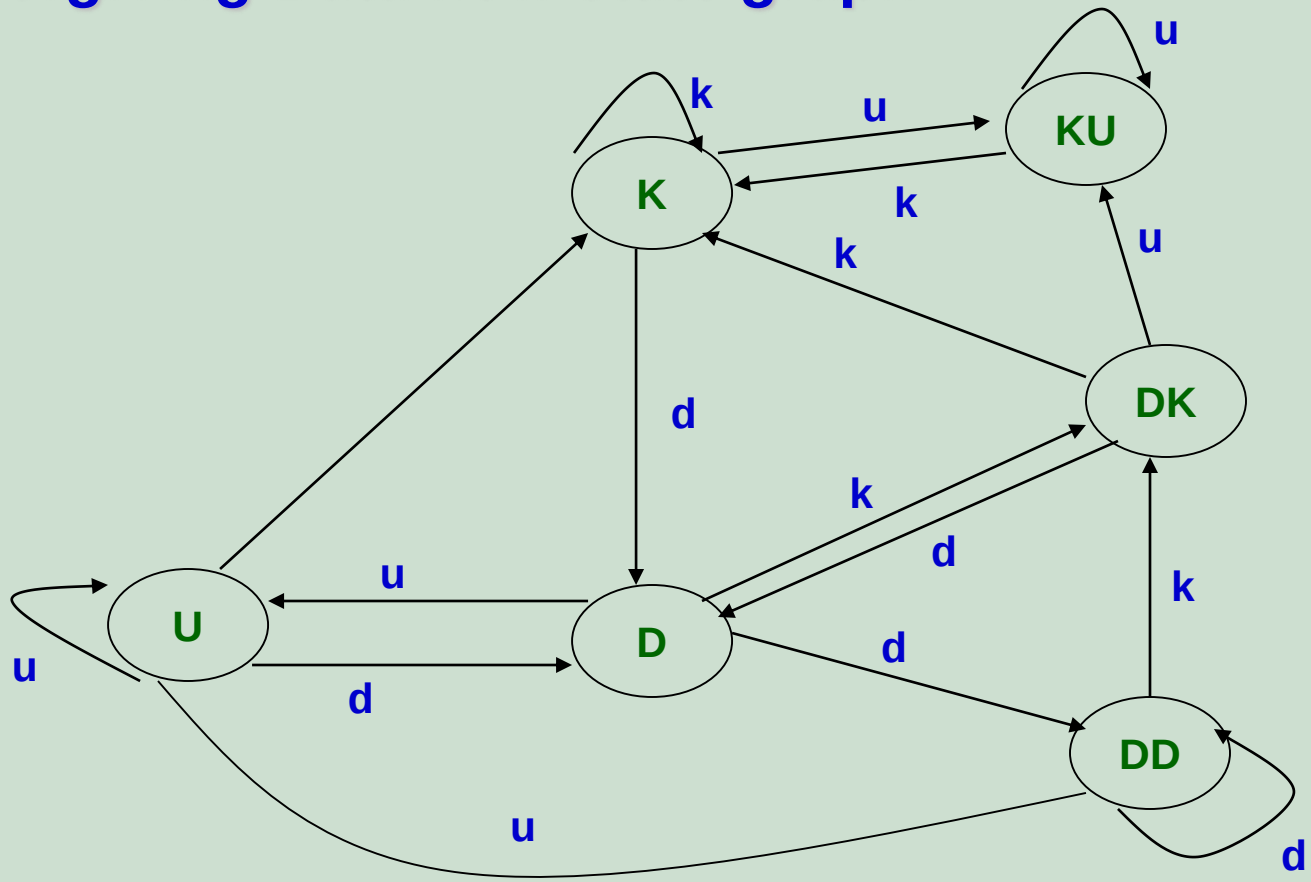
Data Flow Anomaly State graph

- **Object state**
- **Unforgiving Data flow state graph**



Data Flow Anomaly State graph

Forgiving Data flow state graph



A ⇒ DD, DK, KU

Data Flow State Graphs

- **Differ in processing of anomalies**
- **Choice depends on Application, language, context**

Static vs Dynamic Anomaly Detection

- Static analysis of data flows
- Dynamic analysis
  Intermediate data values

Insufficiency of Static Analysis (for Data flow)

1. Validation of Dead Variables
2. Validation of pointers in Arrays
3. Validation of pointers for Records & pointers
1. Dynamic addresses for dynamic subroutine calls
2. Identifying False anomaly on an unachievable path
1. Recoverable anomalies & Alternate state graph
2. Concurrency, Interrupts, System Issues

Data Flow Model

- Based on CFG
- CFG annotated with program actions
- link weights : dk, dp, du etc..
- Not same as DFG
- For each variable and data object

Procedure to Build:

1. Entry & Exit nodes
1. Unique node identification
1. Weights on out link
2. Predicated nodes
3. Sequence of links
    1. Join
    2. Concatenate weights
    3. The converse

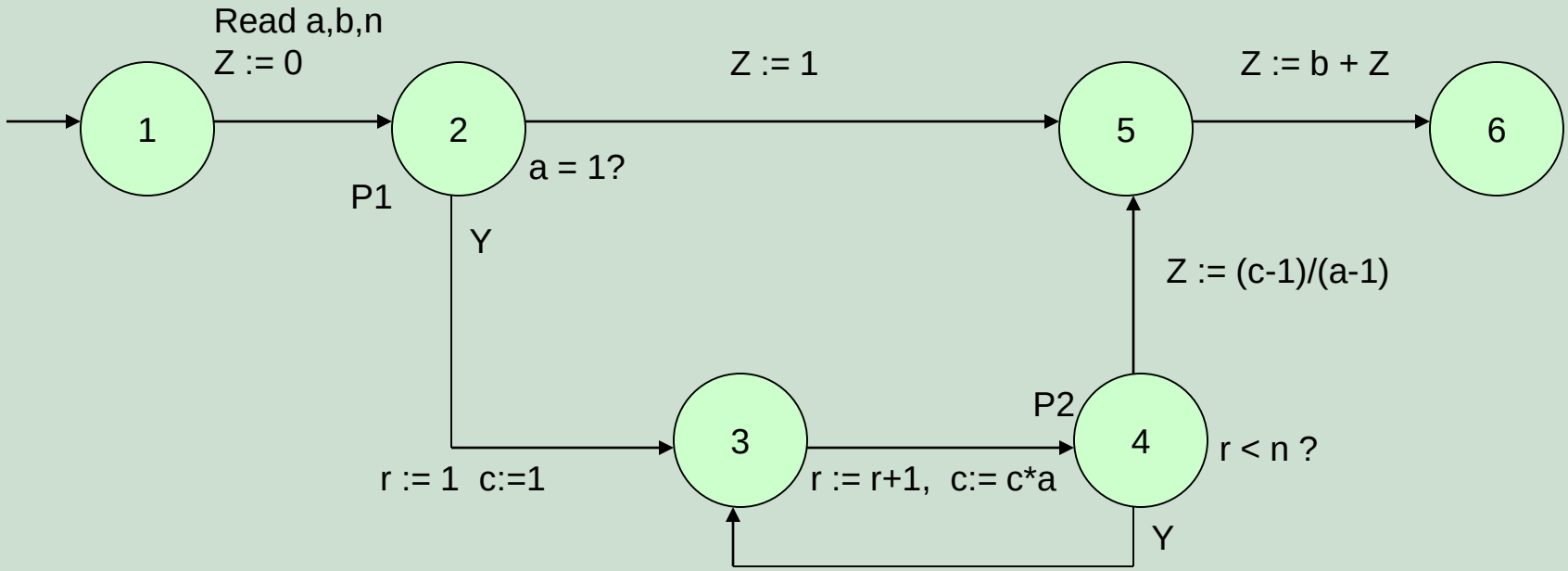
Example:

$$Z = b + \frac{a^n - 1}{a - 1}$$

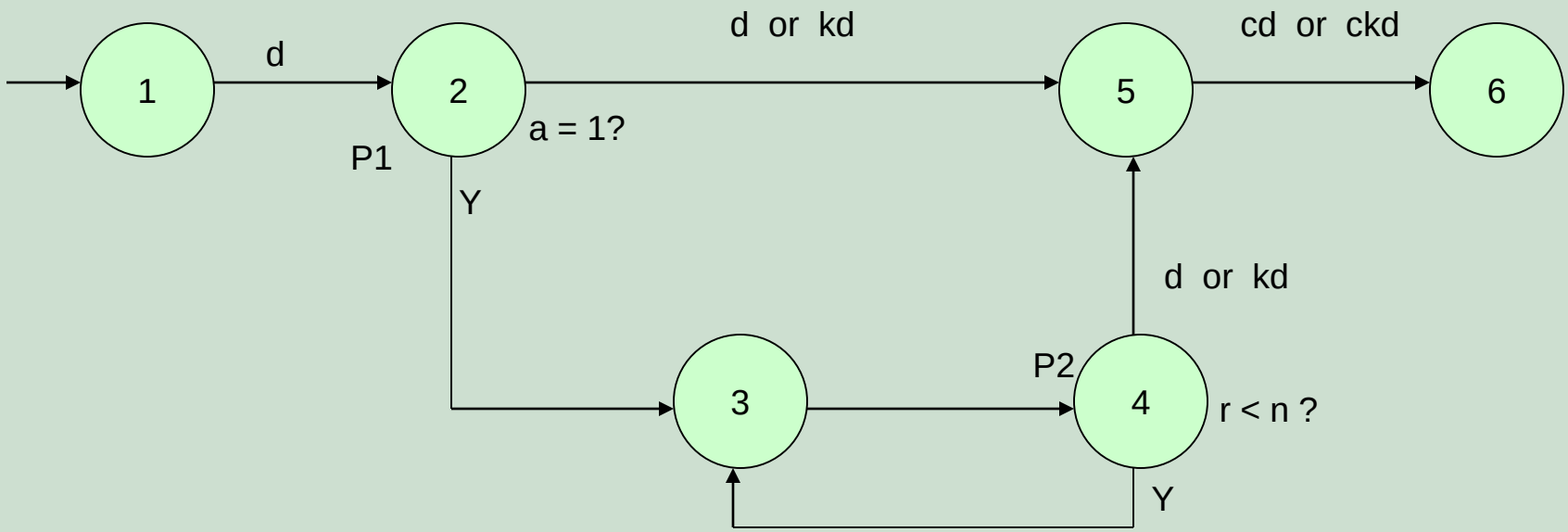
```
START
INPUT a, b, n
Z := 0
IF   a = 1  THEN    Z  :=  1
GOTO  DONE1
r := 1    c := 1
POWER:
    c :=  c * a
    r  :=  r  + 1
    IF   r <= n   THEN     GO TO POWER
    Z := (c – 1) / (a – 1)
DONE1:
Z := b + Z

END
```

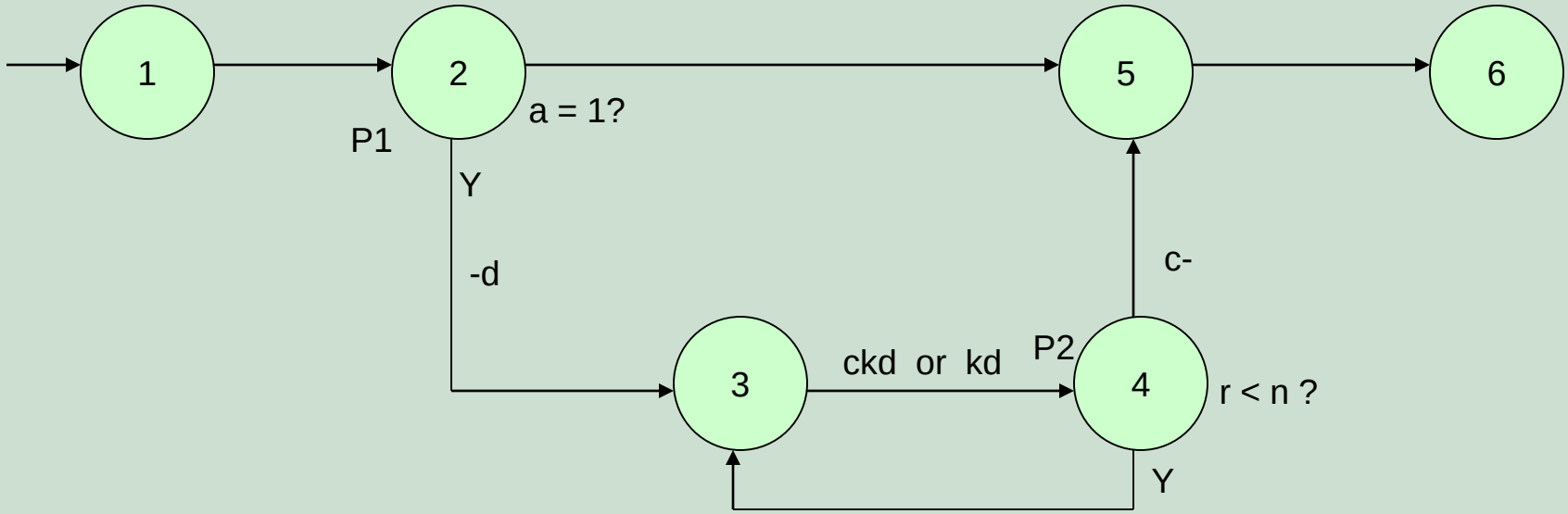
CFG for the Example



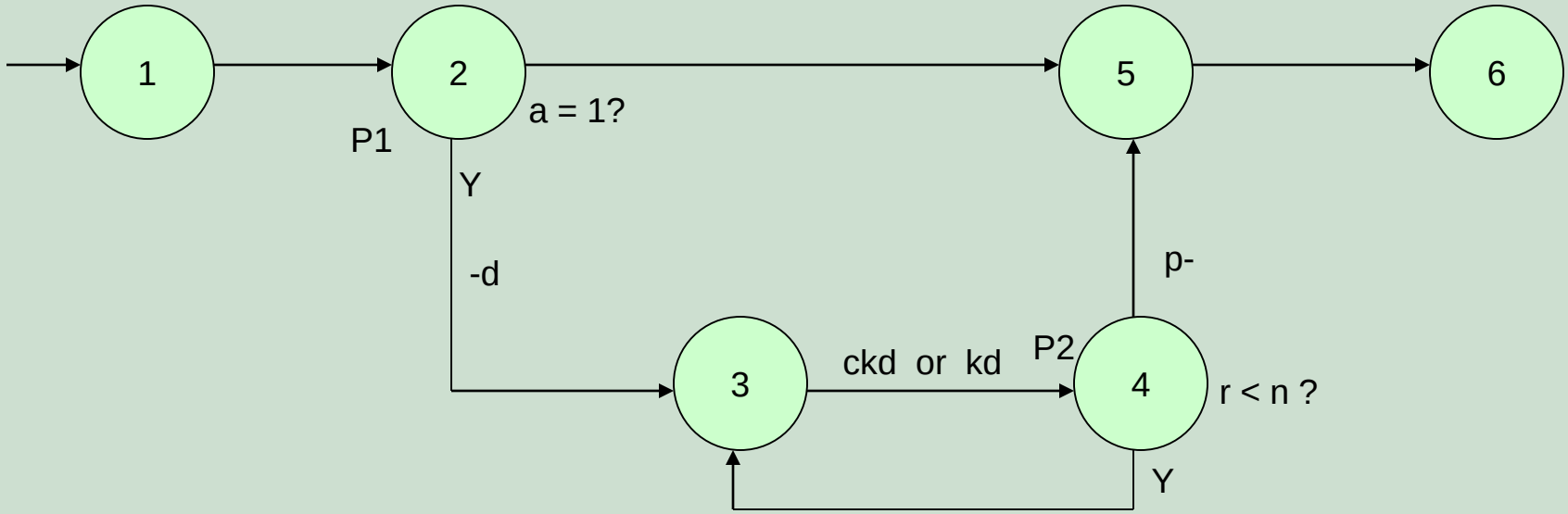
CFG annotated – Data Flow Model for Z



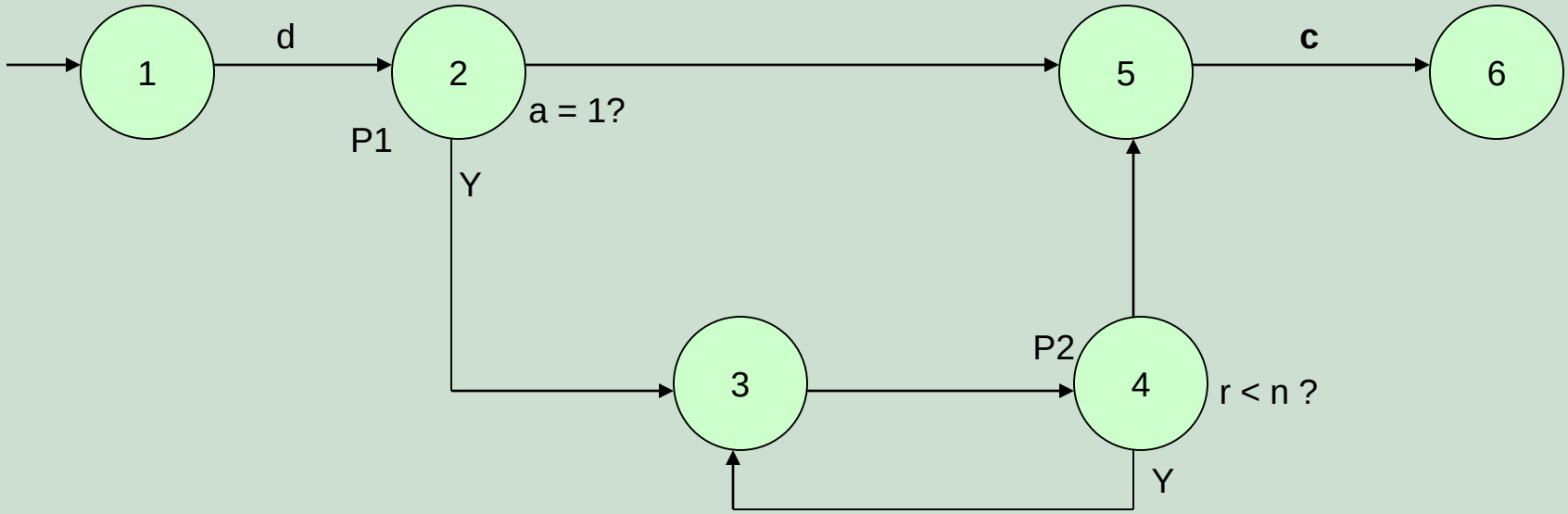
CFG annotated – Data Flow Model for c



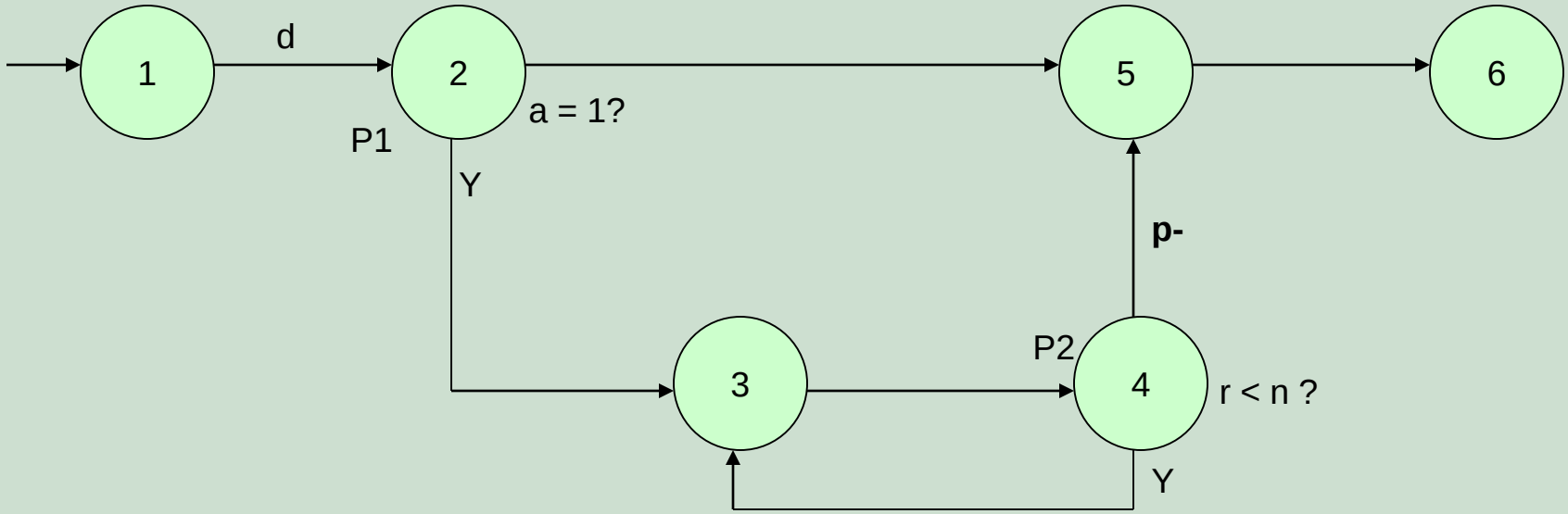
CFG annotated – Data Flow Model for **r**



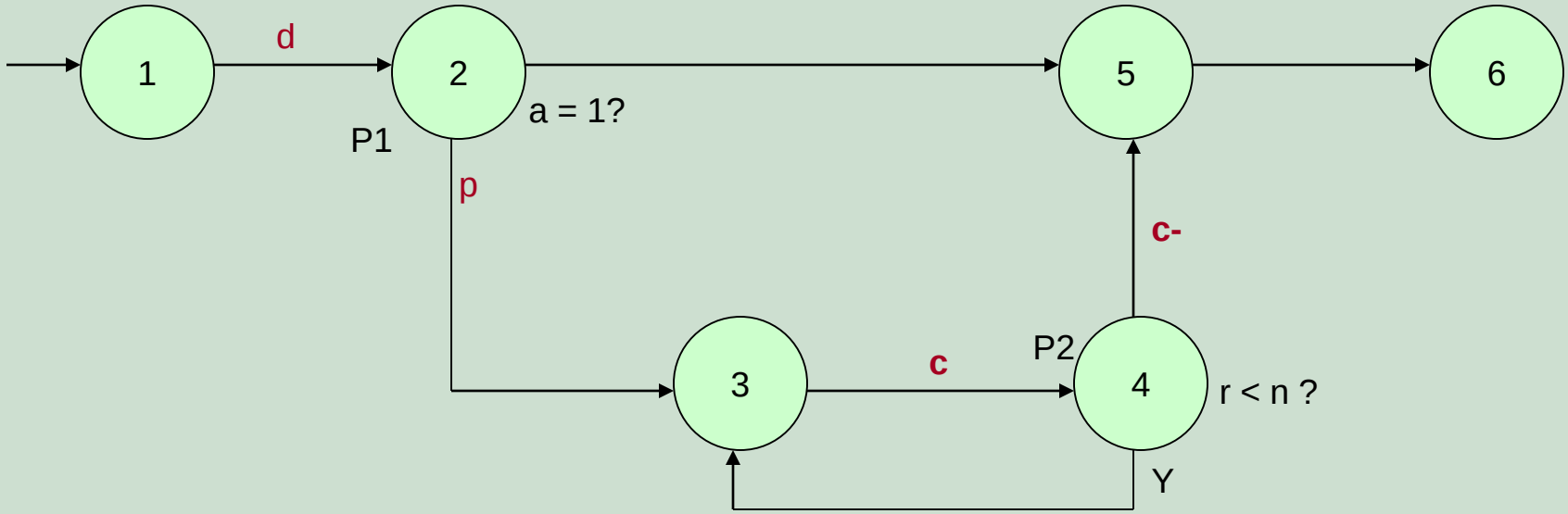
CFG annotated – Data Flow Model for **b**



CFG annotated – Data Flow Model for **n**



CFG annotated – Data Flow Model for **a**



Programmers and Logic

- "Logic" is one of the most often used words In programmers' vocabularies but one of their least used techniques.
- Boolean Algebra is being the simplest form of logic.
- Boolean Algebra is to logic as arithmetic is to mathematics.

Decision Tables

- A decision table is a table that consists of four areas called the condition stub, the condition entry, the action stub and the action entry.
- Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will take place.

An example of a Decision Table

CONDITION ENTRY

	RULE 1	RULE 2	RULE 3	RULE 4
CONDITION 1	YES	YES	NO	NO
CONDITION 2	YES	I	NO	I
CONDITION 3	NO	YES	NO	I
CONDITION 4	NO	YES	NO	YES
ACTION 1	YES	YES	NO	NO
ACTION 2	NO	NO	YES	NO
ACTION 3	NO	NO	NO	YES

ACTION ENTRY

LOGIC-BASED TESTING

- The condition stub is a list of names of conditions.
- A rule specifies whether a condition should or should not be met for the rule to be satisfied. “yes” means that the conditions must be met, and “No” means that conditions must not be met, and “I” means that the conditions plays no part in the rule, or it is immaterial to the rule.
- The action stub names the actions the routine will take or initiate if the rule is satisfied.
- If the action entry is “YES” , the action will take place; if “NO” , the action will not take place.
- Some of the rules are not specified by the decision table for which a default action to be taken are called Default Rules.

LOGIC-BASED TESTING

- Action 1 will take place if condition 1 and 2 are met and if condition 3 and 4 are not met (rule 1) or if condition 1,3 and 4 are met (rule 2)

	RULE 1	RULE 2	RULE 3	RULE 4
CONDITION 1	YES	YES	NO	NO
CONDITION 2	YES	I	NO	I
CONDITION 3	NO	YES	NO	I
CONDITION 4	NO	YES	NO	YES
DEFAULT ACTION	YES	YES	YES	YES

Decision Table Processors

- Decision tables can be automatically translated into code and as such are a higher order language.
- The decision table translator checks the source decision table for consistency and completeness and fills in any required default rules.
- Decision tables as a source language have the virtue of clarity, direct correspondence to specifications, and maintainability.

Decision tables as a Basis for Test Case Design

- If a specification is given as a decision table, it follows that decision tables should be used for test case design.
- If a program's logic is implemented as a decision table, decision table should also be used as a basis for test design.
- It is not always possible or desirable to implement the program as a decision table because the program's logical behavior is only part of its behavior. The program interfaces with other programs, there are restrictions or the decision table language may not have needed features.
- The use of a decision table model to design tests is warranted when:
- The specification is given as a decision table or can be easily converted into one.
- The order in which the predicates are to be evaluated does not affect interpretation of the rules or the resulting action.
- Once a rule is satisfied and an action is selected, no other rule need be examined.
- If several actions can result from satisfying rule, the order in which the actions are executed doesn't matter.

Expansion of Immaterial Cases

- Improperly specified immaterial entries (I) cause most decision-table contradictions.
- If a condition's truth value is immaterial in a rule, satisfying the rule doesnot depend on the condition. It doesn't mean that the case is impossible.
- For example,
- Rule 1: " if the persons are male and over 30, then they shall receive a 15% raise"
- Rule 2: "but if the persons are female, then they shall receive a 10% raise."
- The above rules state that age is material for a male's raise, but immaterial for determining a female's raise.

Expansion of Immaterial Cases

	Rule 2.1	Rule 2.2	Rule 4.1	Rule 4.2	Rule 4.3	Rule 4.4
CONDITION1	YES	YES	NO	NO	NO	NO
CONDITION2	YES	NO	YES	YES	NO	NO
CONDITION3	YES	YES	YES	NO	NO	YES
CONDITION4	YES	YES	YES	YES	YES	YES

LOGIC-BASED TESTING

The expansion of an Inconsistent Specification

	Rule 1	Rule 2
Condition 1	Yes	Yes
Condition 2	I	No
Condition 3	Yes	I
Condition 4	No	No
Action 1	Yes	No
Action 2	No	yes

Rule 1.1	Rule 1.2	Rule 2.1	Rule 2.2
Yes	Yes	Yes	Yes
Yes	No	No	No
Yes	Yes	Yes	No
No	No	No	No
Yes	Yes	No	No
No	No	Yes	Yes

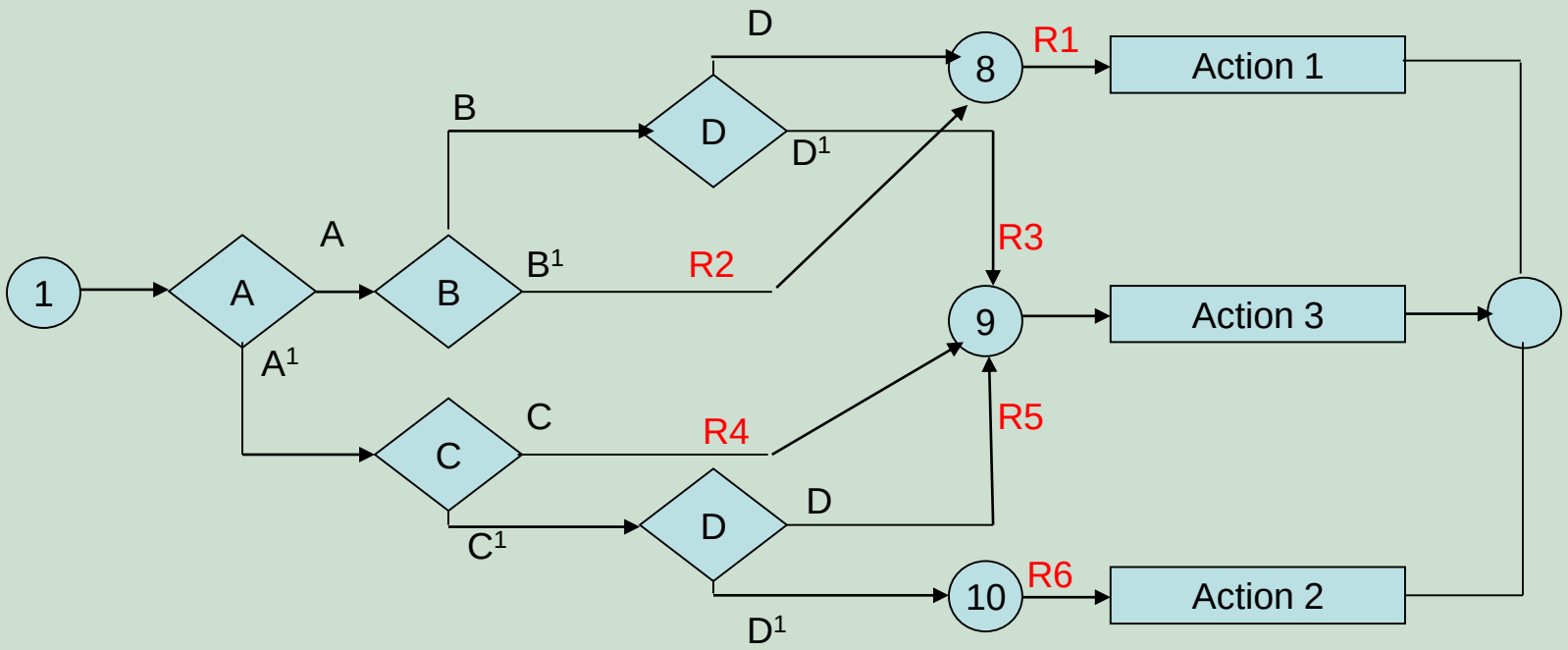
LOGIC-BASED TESTING

Test Case Design

- Test case design by decision tables begins with examining the specification's consistency and completeness.
- This is done by expanding all immaterial cases and checking the expanded tables.
- Once the specification have been verified, the objective of the test case is to show that the implementation provides the correct action for all combinations of predicate values.

Decision tables and Structure

- Decision tables can also be used to examine a program's structure.



LOGIC-BASED TESTING

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6
Condition A	Yes	Yes	Yes	No	No	No
Condition B	Yes	No	Yes	I	I	I
Condition C	I	I	I	Yes	No	No
Condition D	Yes	I	No	I	Yes	No
Action 1	Yes	Yes	No	No	No	No
Action 2	No	No	Yes	Yes	Yes	No
Action 3	No	No	No	No	No	Yes

The decision table corresponding to the previous decision tree