VALIDATION TOOLS

- o TEST DATA GENERATORS
- o EXECUTION FLOW SUMMARIZERS
- o FILE COMPARATORS

Berkeley Pascal PXP -- Version 2.12 (5/11/83)

Mon Apr 9 15:17 1984 pascflow.p

Profiled Mon Apr 9 15:20 1984

```
        1.--|program primes(input, output);

{Prints all prime numbers between 3 and MAXPRIME.
Uses Sieve of Eratosthenes method}

            |const
            |   MAXPRIME = 700;

            |type
            |   boolvec = array [1..MAXPRIME] of boolean;

            |var
            |   primes : boolvec;
            |   i, j, k : 1..MAXPRIME;

            |begin
            |   for i := 1 to MAXPRIME do
      700.--|      if odd(i) then
         350.--|      primes[i] := false
      350.--|      else
         350.--|      primes[i] := true;
            |   i := 3;
            |   k := trunc(sqrt(MAXPRIME) );
            |   while i <= k do begin
         8.--|      j := i + i;
               |      while j <= MAXPRIME do begin
         688.--|         primes[j] := true;
               |         j := j + i
               |      end;
               |      i := i + 2;
               |      while primes[i] and (i <= k) do
           4.--|         i := i + 2
            |   end;
            |   i := 3;
            |   while i <= MAXPRIME do begin
      349.--|      if not primes[i] then
         124.--|         writeln(i, 'is prime');
               |      i := i + 2
            |   end
            |end.
```

WHITE BOX TESTING

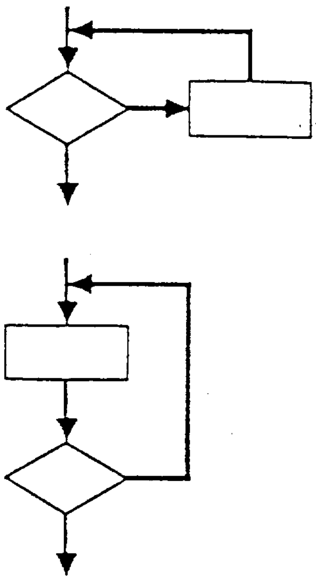
- o STRUCTURAL TESTING

WHITE BOX TESTING

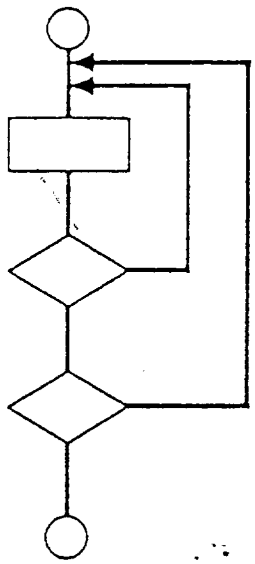
- DERIVE FLOW GRAPH
- DETERMINE CYCLOMATIC COMPLEXITY
- DETERMINE A BASIS SET OF INDEP PATHS
- PREPARE TEST CASES TO FORCE
  EXECUTION OF BASIS SET

WHITE BOX TESTING

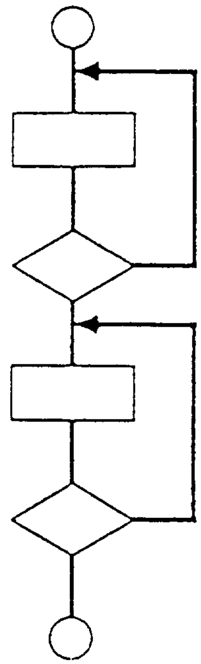
- o LOOP TESTING
  - SIMPLE
  - NESTED
  - CONCATENATED
  - UNSTRUCTURED



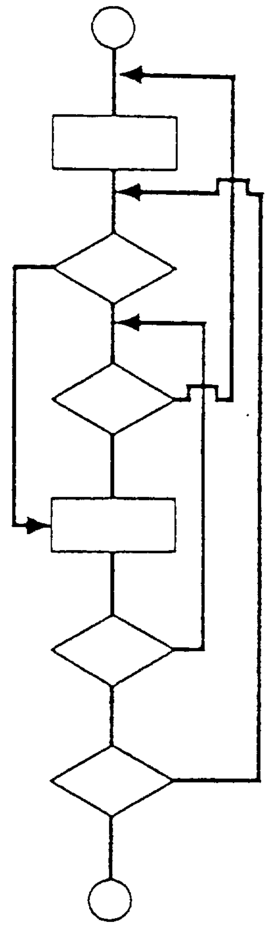
Simple loops



Nested loops



Concatenated
loops

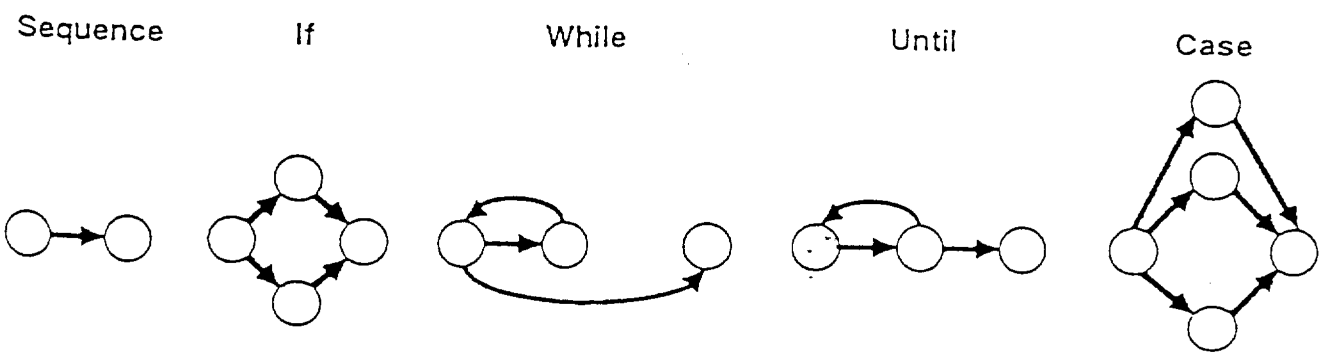


Unstructured
loops

STRUCTURAL TESTING

- PATHS TO EXERCISE
- DERIVE TEST DATA TO EXERCISE PATHS
- TEST COVERAGE CRITERION
- EXECUTE TEST CASES
- MEASURE TEST COVERAGE ACHIEVED

The structured constructs in flow graph form:



where each circle represents one or more
nonbranching PDL or source code statements

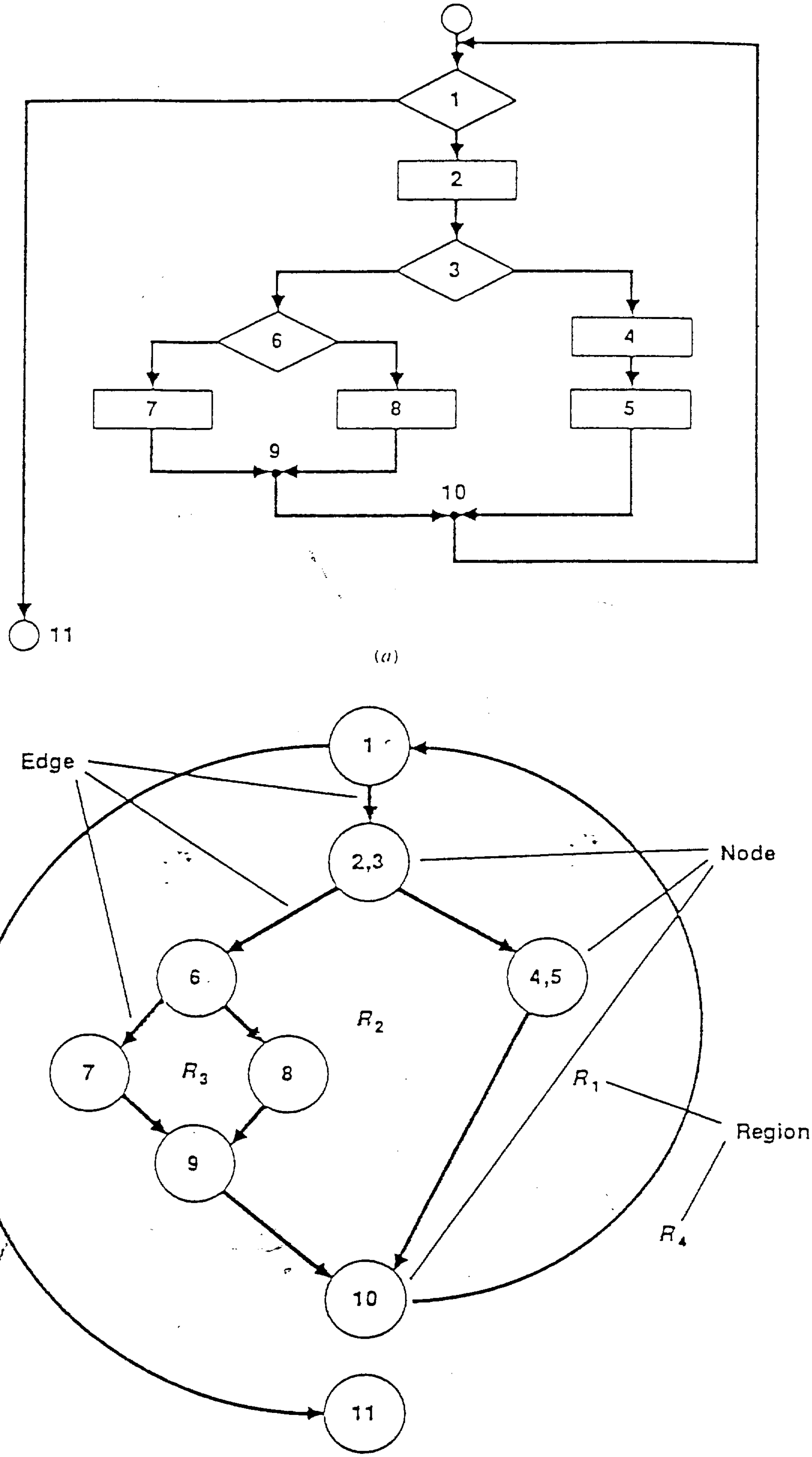
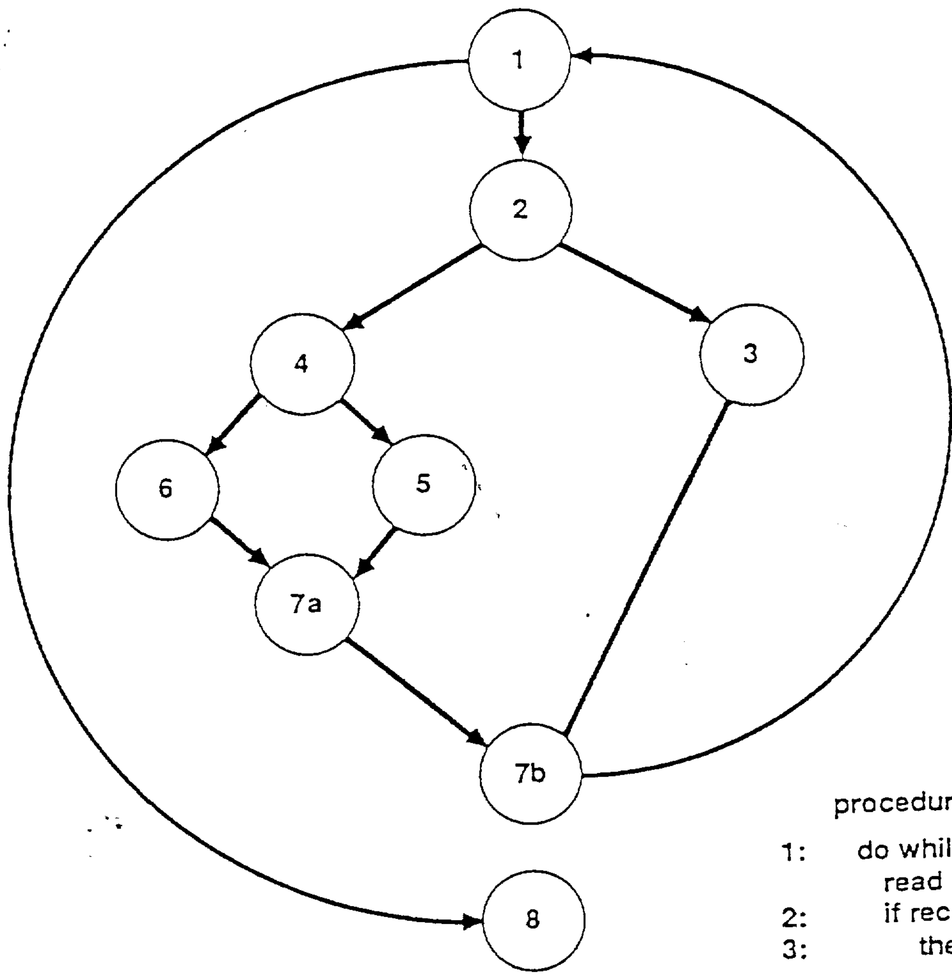


FIGURE 13.4
(*a*) Flowchart. (*b*) Flow graph.

Flow graph



PDL

```
procedure: sort
1:    do while records remain
         read record;
2:       if record field 1 = 0
3:          then process record;
               store in buffer;
               increment counter;
4:          elseif record field 2 = 0
5:             then reset counter;
6:             else process record;
               store in file;
7a:         endif
         endif
7b:   enddo
8:    end
```

FIGURE 13.5
Translating PDL to flow graph.

```
    .
    .
    .
IF a OR b
   then procedure x
   else procedure y
ENDIF
```

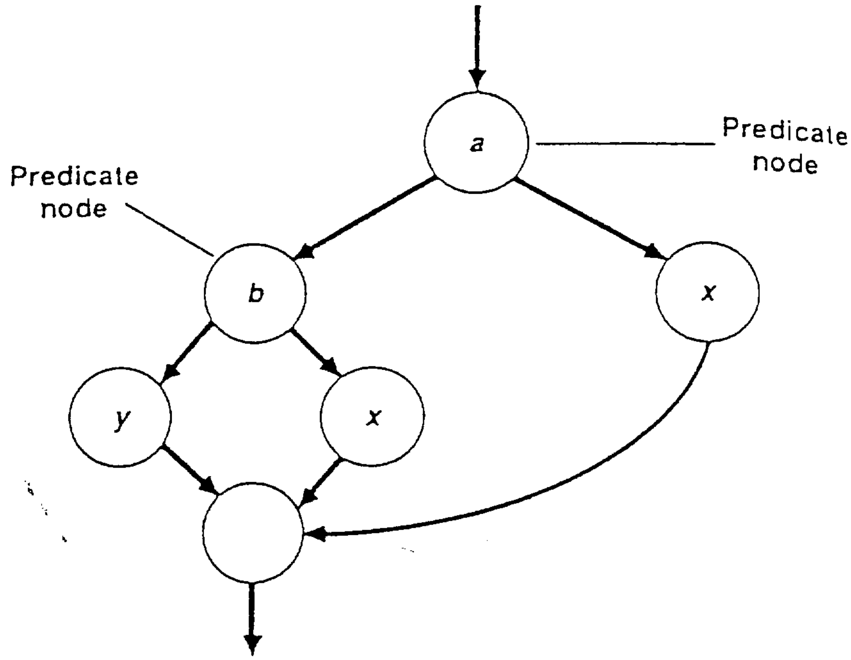


FIGURE 13.6
Compound logic.

```
PROCEDURE average;

    * This procedure computes the average of 100 or fewer
      numbers that lie bounding values; it also computes the
      total input and the total valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100]  IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum  IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[ i ] <> -999 and total.input < 100
        increment total.input by 1;
        IF value[ i ] >= minimum AND value[ i ] <= maximum
            THEN increment total.valid by 1;
                    sum = sum + value[ i ];
            ELSE skip
        ENDIF
        increment i by 1;
    ENDDO
    IF total.valid > 0
        THEN average = sum / total.valid;
        ELSE average = -999;
    ENDIF
END average
```

FIGURE 13.7

PDL for test case design.

```
PROCEDURE average;

    * This procedure computes the average of 100 or fewer
      numbers that lie bounding values; it also computes the
      total input and the total valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100]  IS SCALAR ARRAY;
    TYPE average, total.input, total.valid,
        minimum, maximum, sum  IS SCALAR;
    TYPE I IS INTEGER;
       i = 1;
(1)    total.input = total.valid = 0;
       sum = 0;
    DO WHILE value[ i ] <> -999 (2) and total.input < 100 (3)
    (4) increment total.input by 1;
        IF value[ i ] >= minimum (5) AND value[ i ] <= maximum (6)
        (7)  THEN increment total.valid by 1;
                   sum = sum + value[ i ];
             ELSE skip
    (8) ENDIF
        increment i by 1;
(9) ENDDO
    IF total.valid > 0 (10)
    (11) THEN average = sum / total.valid;
    (12) ELSE average = -999;
(13) ENDIF
END average
```

FIGURE 13.8
Identifying nodes.

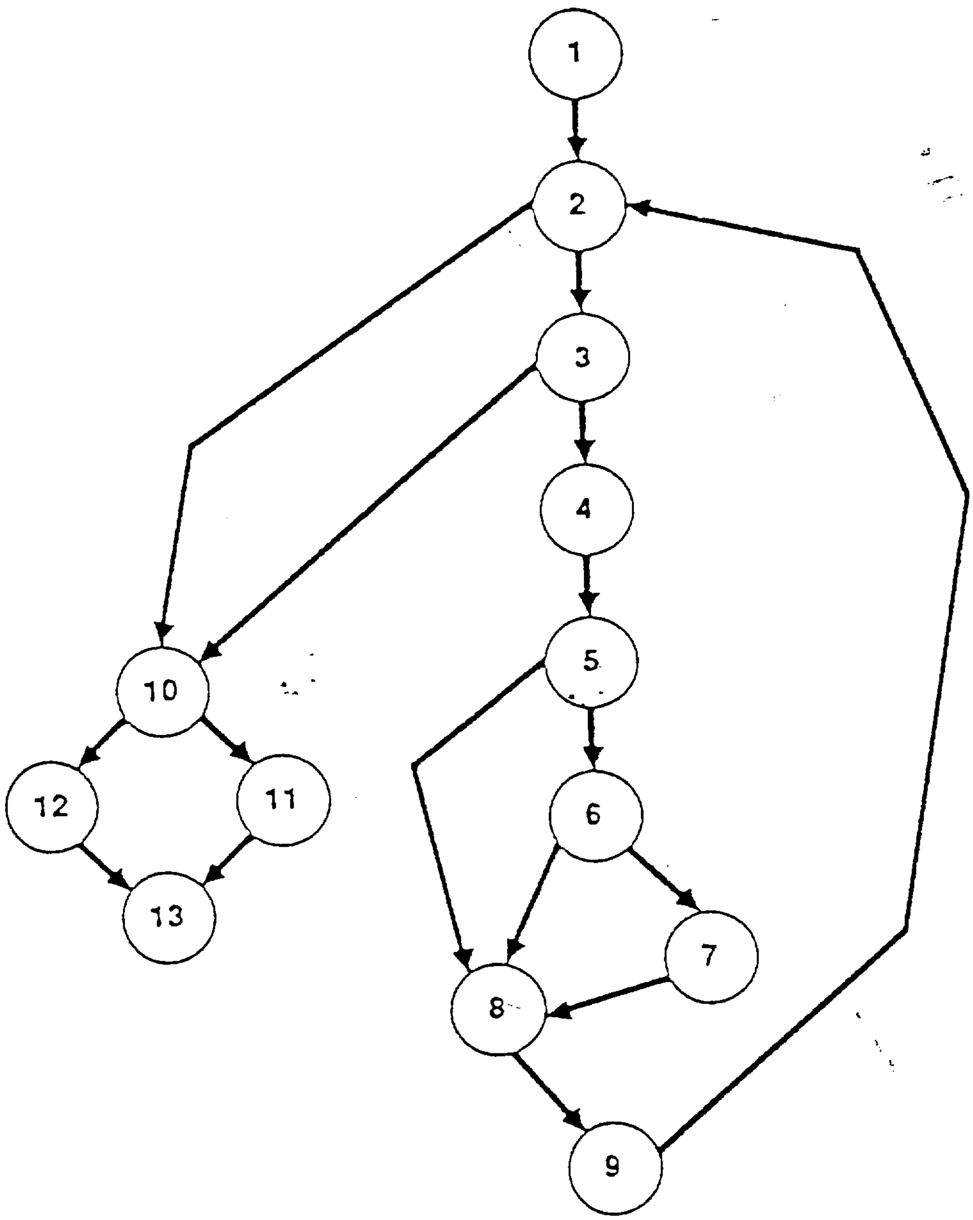
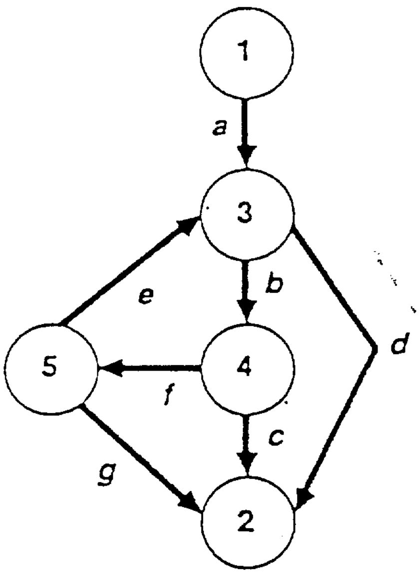


FIGURE 13.9
Flow graph of the procedure *average*.



Flow graph

Node \ Connected to node	1	2	3	4	5
1			*a*		
2					
3		*d*		*b*	
4		*c*			*f*
5		*g*	*e*		

Graph matrix

Node \ Connected to node	1	2	3	4	5
1			1		
2					
3		1		1	
4		1			1
5		1	1		

Connections

1 $-1 = 0$

2 $-1 = 1$

2 $-1 = 1$

2 $-1 = \underline{1}$

$3 + 1 = \boxed{4}$

Cyclomatic complexity

PROGRAM ERRORS

- o MISSING PATH ERRORS
- o COMPUTATIONAL ERRORS
- o DOMAIN ERRORS

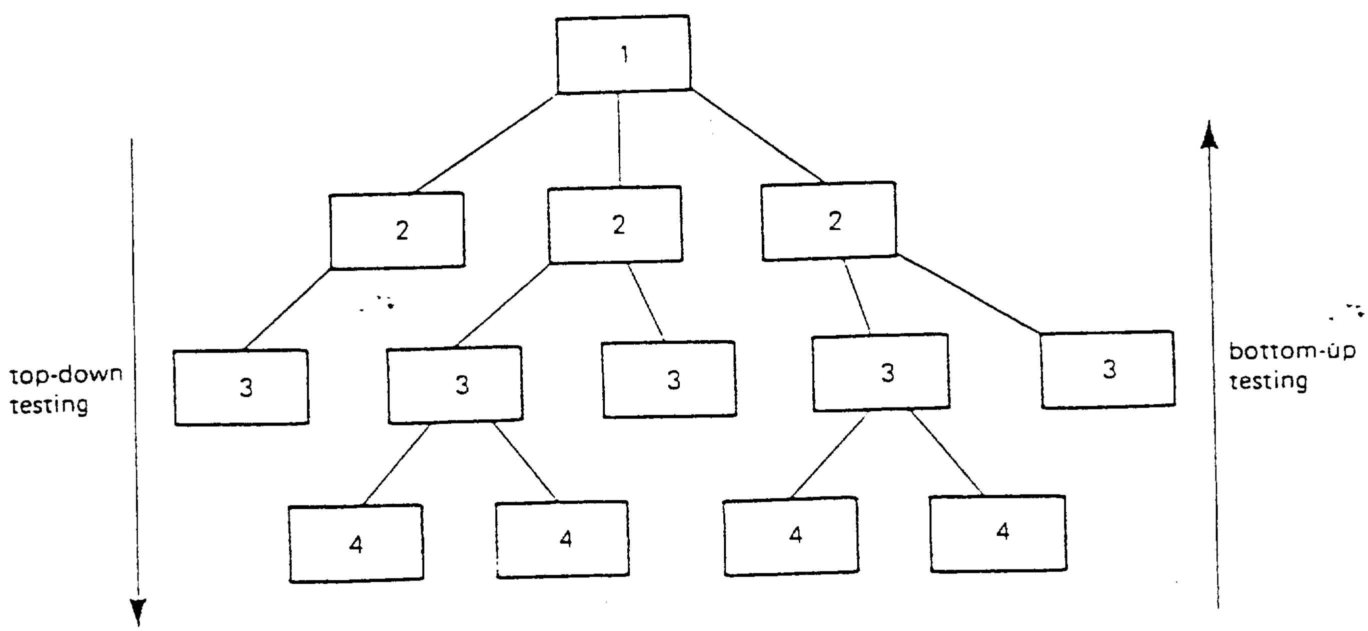
MEASURES OF TEST COVERAGE

- o STATEMENT COVERAGE
- o BRANCH COVERAGE
- o LOGICAL PATH COVERAGE

TESTING PHILOSOPHIES

- o MODULE TESTING
  - INCREMENTAL
  - NON-INCREMENTAL

- o TOP-DOWN
- o BOTTOM-UP
- o MODIFIED TOP-DOWN
- o SANDWICH
- o MODIFIED SANDWICH


1
2
2
2
3
3
3
3
3
4
4
4
4
top-down testing
bottom-up testing

TOP-DOWN VS BOTTOM-UP TESTING

- o TOP-DOWN
  - ADVANTAGES
    - EARLY DETECTION OF ERRORS
    - PRELIMINARY VERSION SOON
    - ELIMINATES DRIVERS
  - DISADVANTAGES
    - PROGRAM STUBS
    - TEST OUTPUT DIFFICULT TO OBSERVE
- o BOTTOM-UP
  - ADVANTAGES
    - EASIER TO CONSTRUCT TEST CASES
    - PROGRAM STUBS ELIMINATED
  - DISADVANTAGES
    - LATE DETECTION ERRORS-REWRITE
    - PRELIMINARY VERSION LATE
    - PROGRAM DRIVERS

THREAD TESTING APPROACH

- o EARLY DEMO OF KEY FUNCTIONS
- o EARLY COMPLIANCE WITH INTERFACE REQMT'S
- o EXCELLENT STATUS/QUALITY OF CODE

THREAD TESTING

- STRING OF PROGRAMS WHICH DEMONSTRATE A DISTINCT PROCESSING FUNCTION

SYSTEM TESTING

- o RECOVERY
- o SECURITY
- o STRESS

DEBUGGING

- o LOCATE PARTS OF CODE INCORRECT
- o MODIFY CODE TO MEET REQMT'S

DEBUGGING TECHNIQUES

- o PROGRAM STATEMENTS
- o BACKTRACKING
- o CAUSE ELIMINATION

- ERROR SEEDING
- INDEPENDENT GROUP
- HISTORICAL DATA

FAULT TOLERANCE

- o RECOVERY BLOCK
- o N-VERSION SOFTWARE

PROGRAM TESTING CAN ONLY DEMONSTRATE THE PRESENCE OF ERRORS NOT THE ABSENCE

TESTING NEVER ENDS - JUST GETS

TRANSFERRED TO THE CUSTOMER