

CSE 4621 / SWE 5620

Software Metrics

Measuring Internal Product Attributes: Structure

Chapter -9-

Khaled Slhoub, PhD

- There is clearly a link between the structure of software products and their quality
 - to understand the difficulty, we sometimes have in converting one product into another
 - implementing a design as code,
 - in testing code,
 - validating requirements,
 - to predict external software attributes from early internal product measures

How complex is the following program?

```
1: read x,y,z;  
2: type = "scalene";  
3: if (x == y or x == z or y == z) type ="isosceles";  
4: if (x == y and x == z) type ="equilateral";  
5: if (x >= y+z or y >= x+z or z >= x+y) type ="not a triangle";  
6: if (x <= 0 or y <= 0 or z <= 0) type ="bad inputs";  
7: print type;
```

Is there a way to measure it?

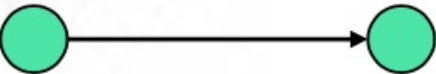
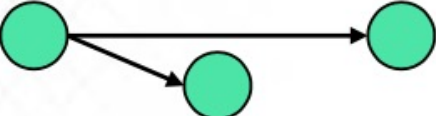
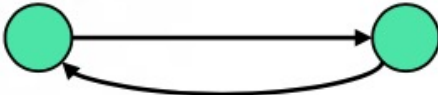
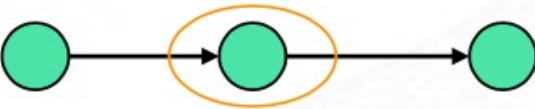
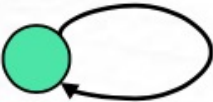
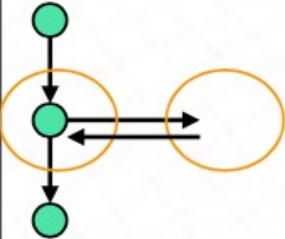
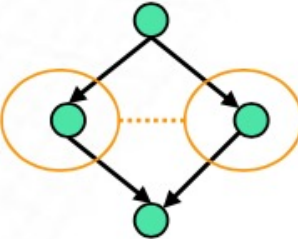
It has something to do with program structure (branches, nesting) & flow of data

Software structure is often (and misleadingly) called “complexity” metric

- We would like to assume that a large module takes longer to specify, design, code, and test than a small one.
- BUT, the structure of the product plays a part, not only in requiring development effort but also in how the product is maintained.
- We can think of structure from at least two perspectives:
 1. Control flow structure (reflects the iterative and looping nature of programs)
 2. Data flow structure (Keeping track of data as it is created or handled by the program). To depict the behavior of the data as it interacts with the program.

- Basic Control Structures (BCSs) are set of essential controlflow mechanisms used for building the logical structure of the program
- Types of Basic Control Structures
 - **Sequence**: e.g., a list of instructions with no other BCSs involved
 - **Selection**: e.g., if ... then ... else.
 - **Iteration**: e.g., while ; for, do-while.
- Other types of BCSs (advanced BCSs), such as:
 - Procedure/function/agent call
 - Recursion (self-call)
 - Interrupt
 - Concurrency

Control Flow Structure of Program Units

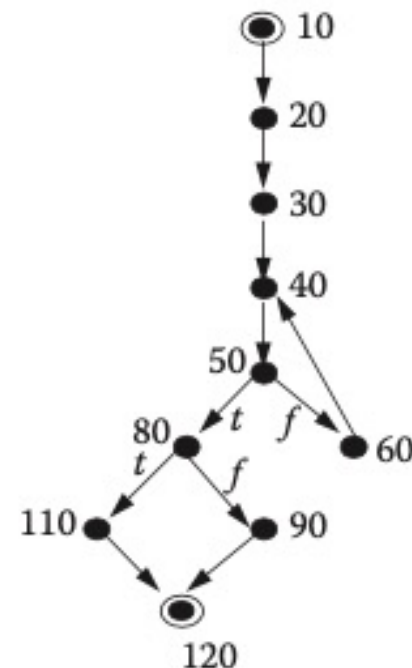
Sequence			
Selection			
Iteration			
Procedure/ function call			
Recursion			
Interrupt		Concurrency	

- Control-flow structure can be represented by **control flowgraphs**
- Flowgraph is usually modeled by a directed graph
 $CFG = \{N, A\}$
 - Nodes (each node corresponds to a program statement)
 - Start
 - Terminal
 - Procedural
 - Predicate / Decision
 - Edges (arcs) - connect two nodes (indicates flow of control from one statement of program to another)

Flowgraph - Example

- The nodes enumerate the program statements, and the arcs make visible the control patterns

```
10 INPUT P
20 Div = 2
30 Lim = INT(SQR(P))
40 Flag = P/Div - INT(P/Div)
50 IF Flag = 0 OR Div = Lim THEN 80
60 Div = Div + 1
70 GO TO 40
80 IF Flag <> 0 OR P > 4 THEN 110
90 PRINT Div; "Smallest factor of"; P; "."
100 GO TO 120
110 PRINT P; " is prime"
120 END
```



A **path** is a sequence of consecutive (directed) edges, some of which may be traversed more than once during the sequence. **Simple path** $\langle x, y \rangle$
A **simple path** is one in which there are no repeated edges.

- In-degree = # arcs arriving at the node
 - In-degree of start node is often but not necessarily 0
- Out-degree = # arcs that leave the node
 - Out-degree of terminal node = 0
- Procedure Node = out-degree = 1
- Predicate Node = out-degree > 1 (e.g., nodes 50 and 80)
- Execution starts at the start node S and ends at the terminal node T
- For each node, N,
 - There is a path from start node, S, to N
 - There is a path from N to terminal node, T
 - N could be replaced with a proper flowgraph and the resulting flowgraph will still be a proper flowgraph

Sequencing of Flowgraphs

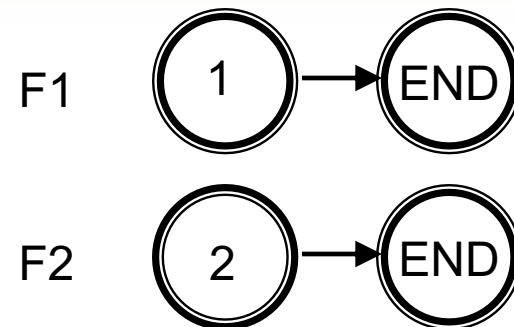
- If F1 and F2 are two flowgraphs

- We make a **sequence** of F1 and F2 by replacing the terminal node of F1 with the start node of F2.

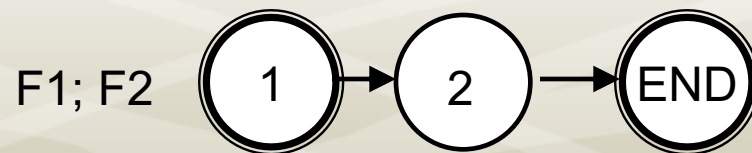
- Notation:

- F1; F2
- Seq (F1, F2)
- P2 (F1, F2)
- F1 to F2

Sequencing these:

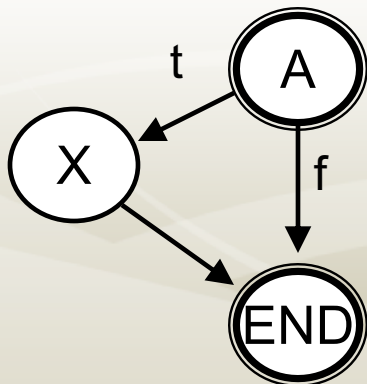


Yields this

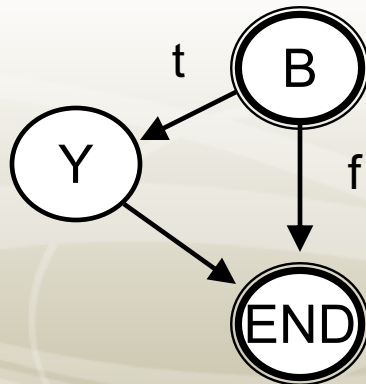


Nesting of Flowgraphs

- If F1 and F2 are two flowgraphs and X is a procedure node.
 - F2 is **nested in F1 at X** if we replace **the arc from X** with the flowgraph F2 (F2's start node is X)
 - Notation:
 - F1(F2 on X)
 - F1(F2) is OK if there is no ambiguity

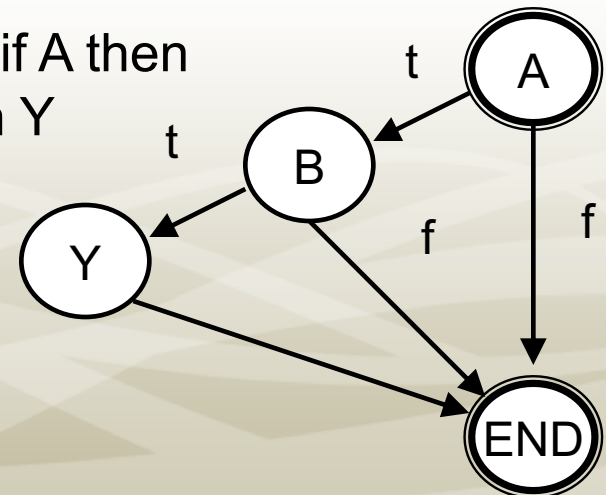


F1: if A then X



F2: if B then Y

F1(F2): if A then
if B then Y

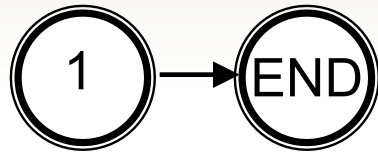


- Flowgraphs that cannot be decomposed non-trivially by sequencing and nesting.
- Examples:
 - P_n (sequence of n statements)
 - D_0 (if-condition)
 - D_1 (if-then-else-branching)
 - D_2 (while-loop)
 - D_3 (repeat-loop)
 - C_n (case)

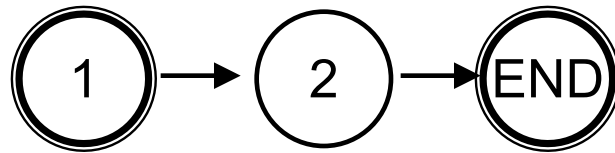
That correspond to the basic control constructs in imperative language programming.

- P_n – A simple series of statements

P_1



$P_2 (1,2)$

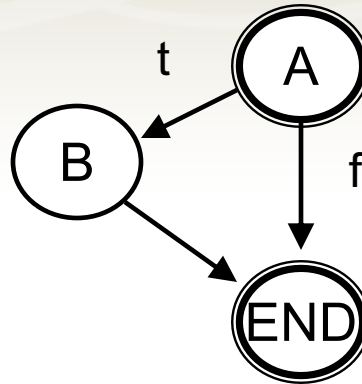


P_2 flowgraph represents the construct sequence, where a program consists of a sequence of two statements

$P_n (1,2, \dots, n)$



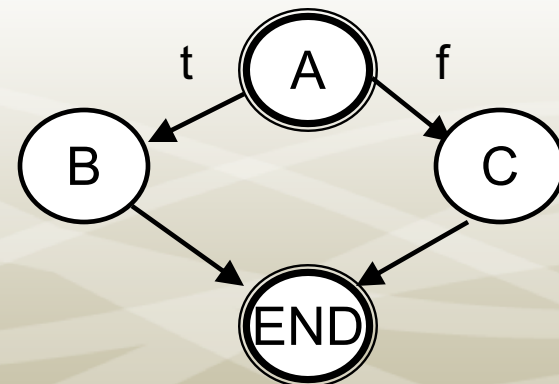
- D0 If A then B



D0 (A,B)

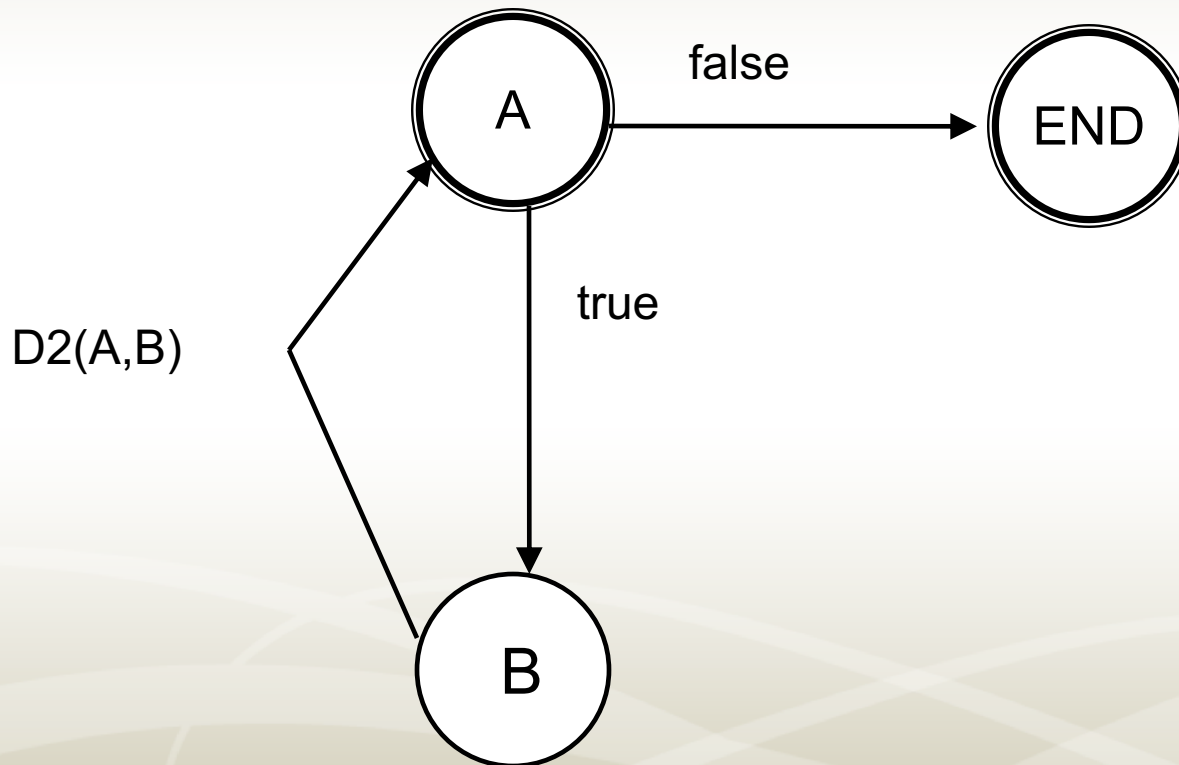
- D1 If A then B else C

D1 (A,B,C)



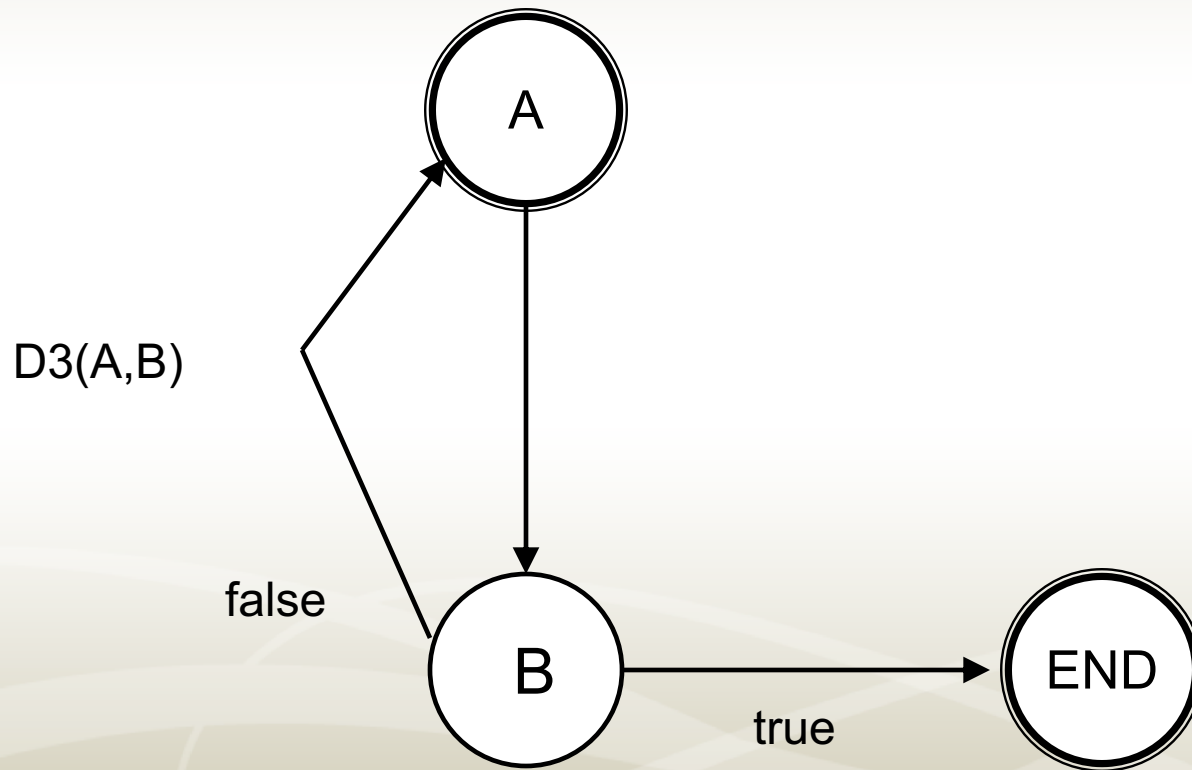
Prime Flowgraphs

- D2 while A do B

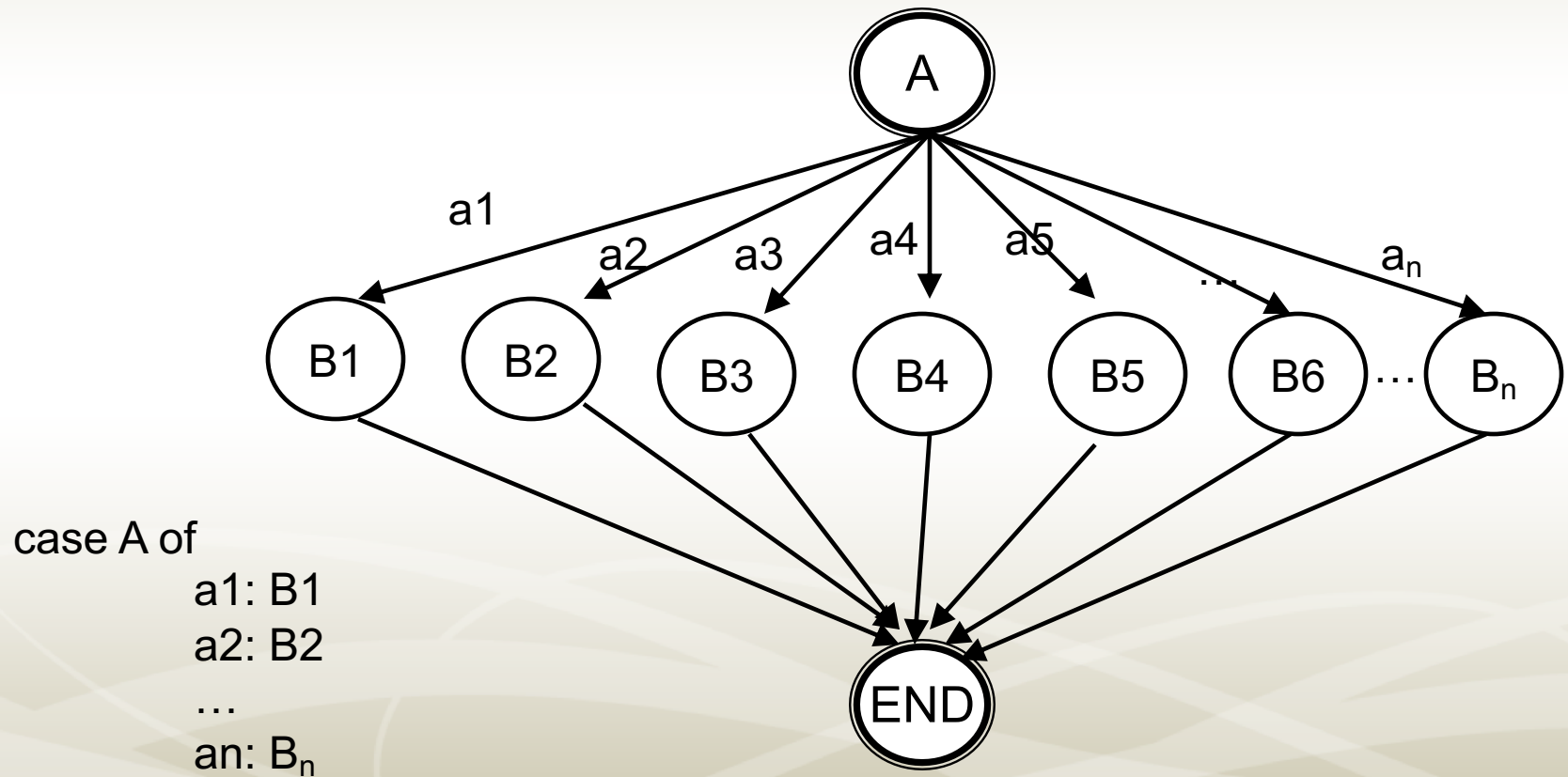


Prime Flowgraphs

- D3 repeat A until B

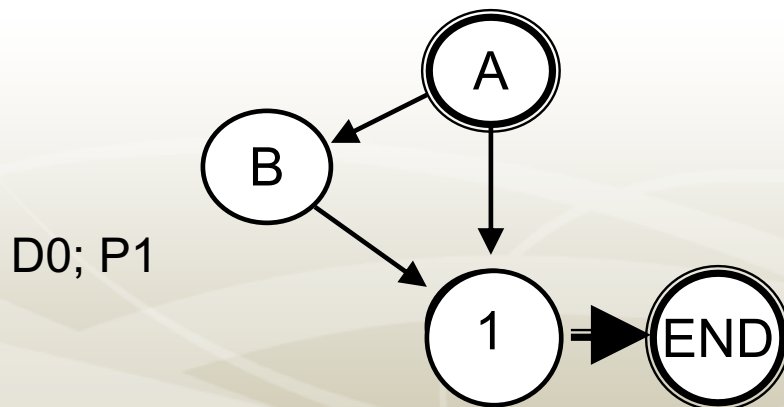
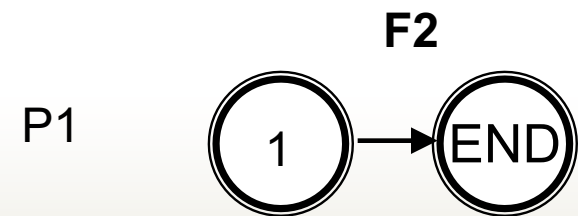
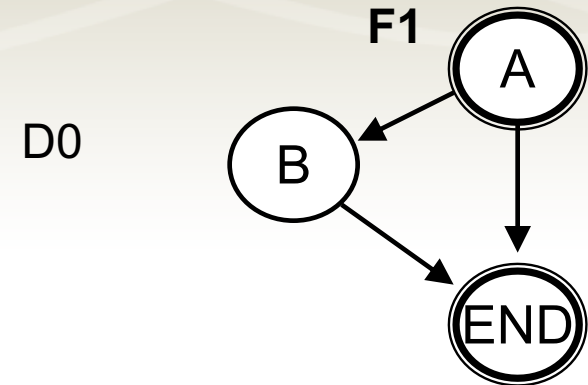


- Cn Case statement



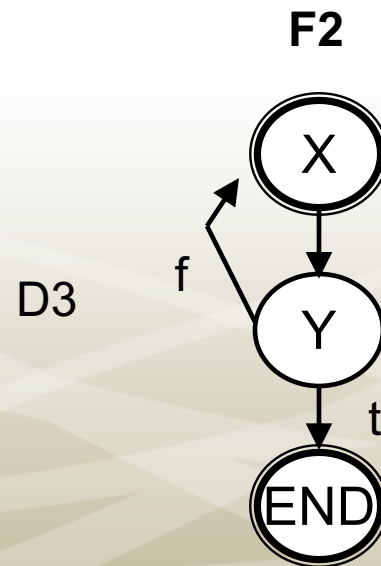
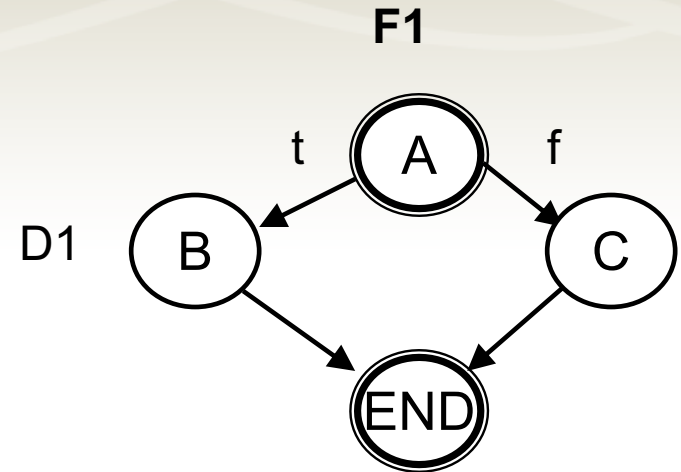
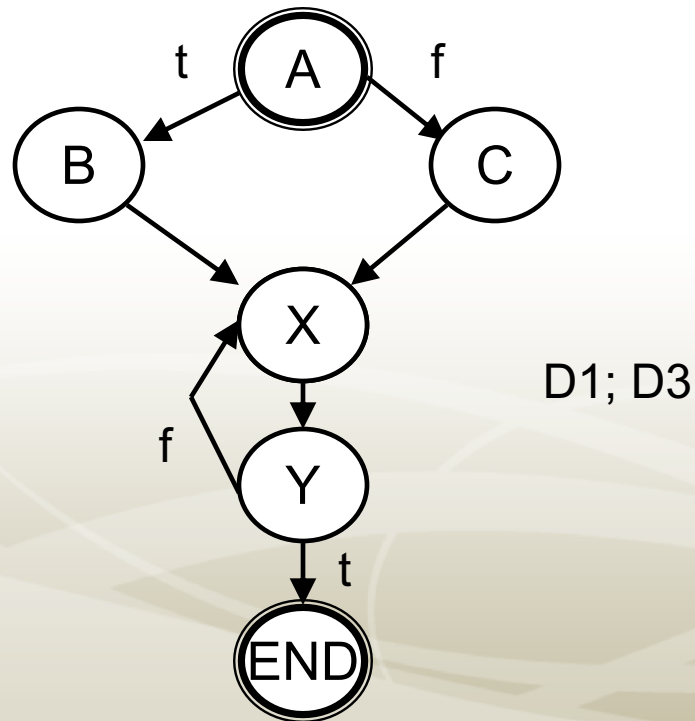
Sequence Practice

Let F_1 and F_2 be two flowgraphs. Then, the sequence of F_1 and F_2 , (shown by $F_1; F_2$) is a flowgraph formed by merging the terminal node of F_1 with the start node of F_2 .



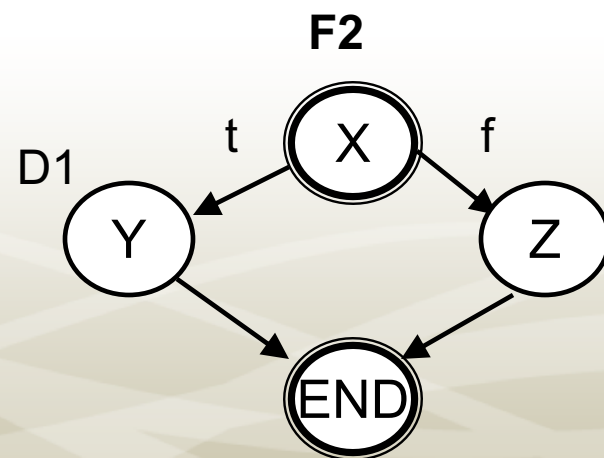
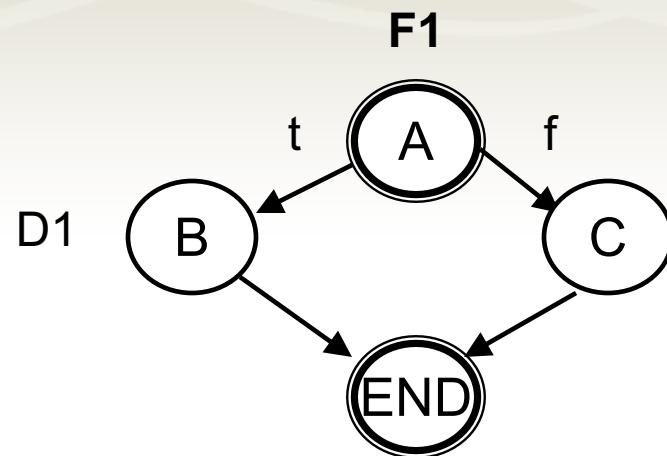
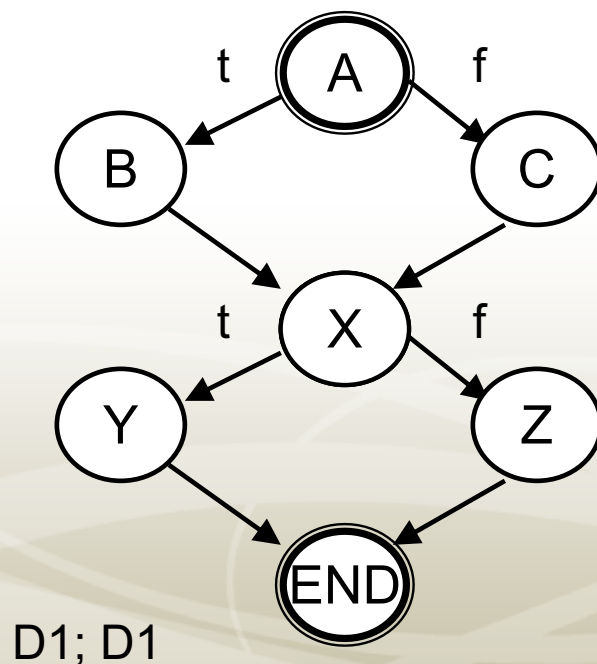
Sequence Practice

Let F_1 and F_2 be two flowgraphs. Then, the sequence of F_1 and F_2 , (shown by $F_1; F_2$) is a flowgraph formed by merging the terminal node of F_1 with the start node of F_2 .



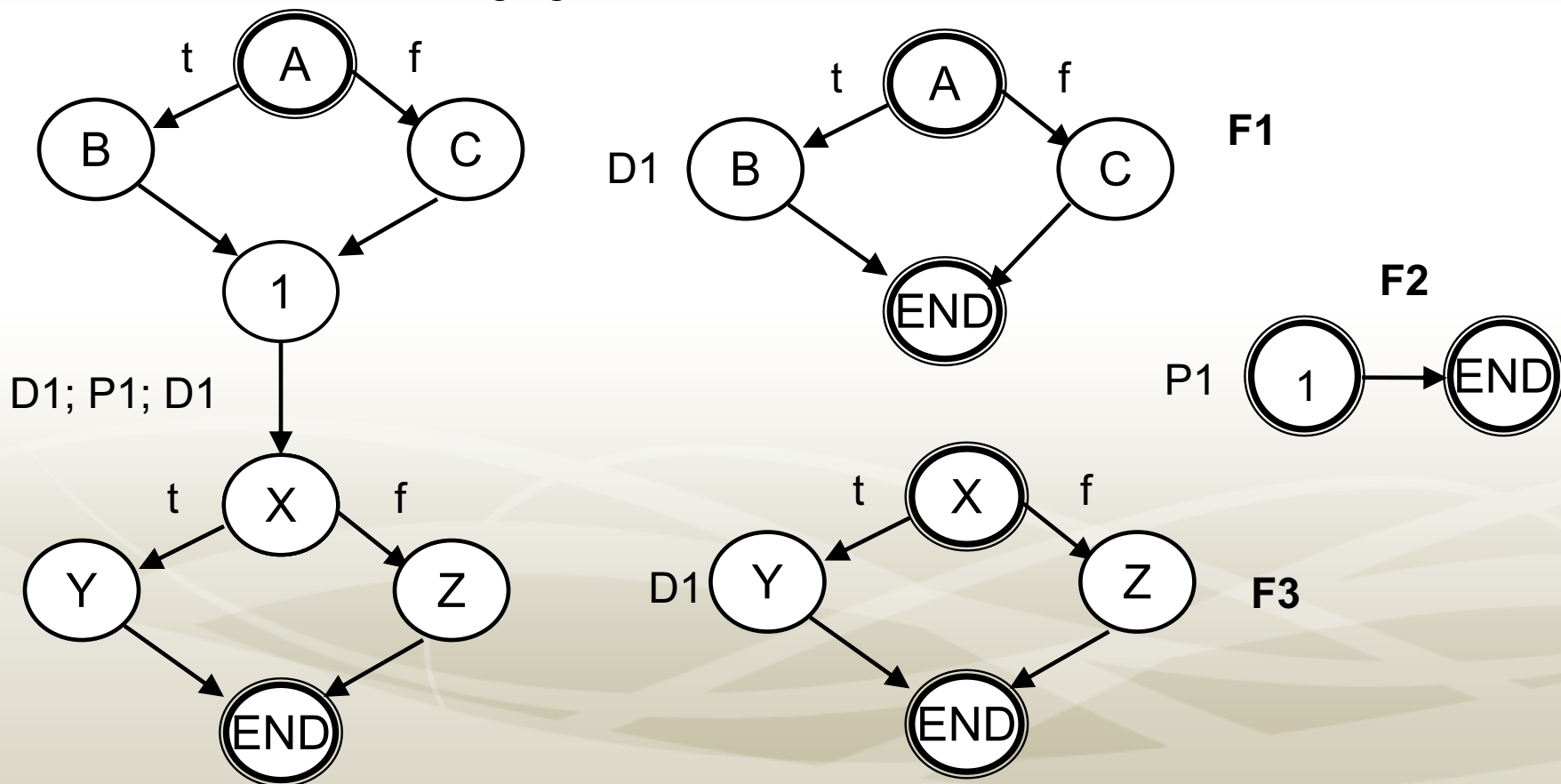
Sequence Practice

Let F_1 and F_2 be two flowgraphs. Then, the sequence of F_1 and F_2 , (shown by $F_1; F_2$) is a flowgraph formed by merging the terminal node of F_1 with the start node of F_2 .



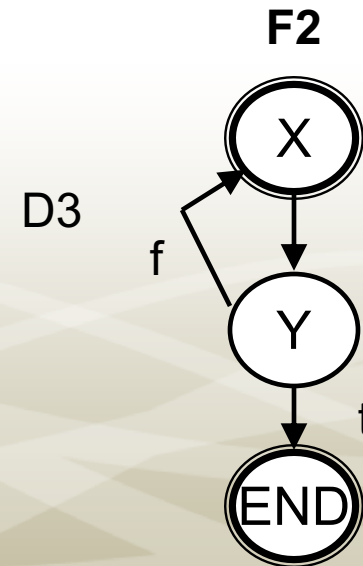
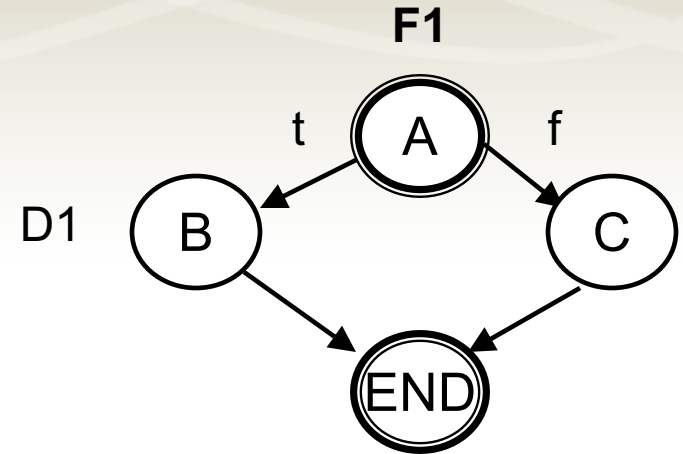
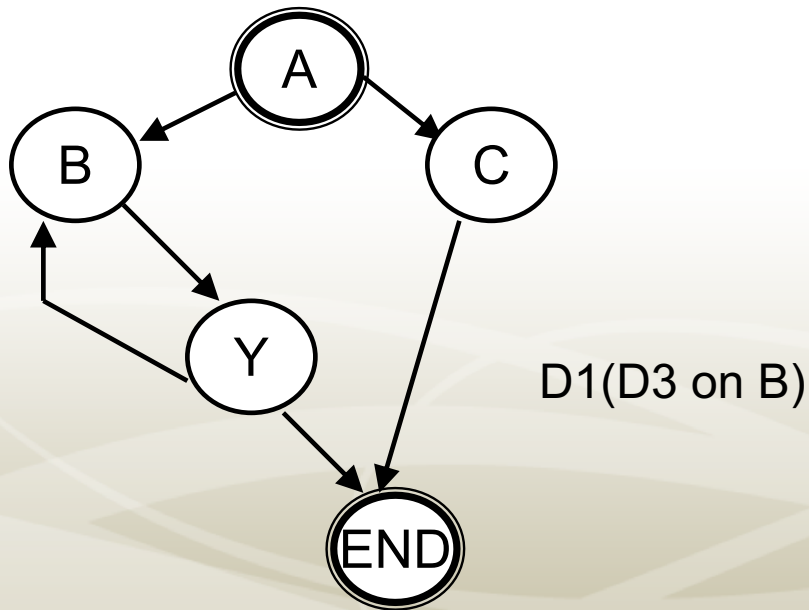
Sequence Practice

Let **F₁**, **F₂** and **F₃** be three flowgraphs. Then, the sequence of **F₁**, **F₂**, and **F₃**, (shown by **F₁; F₂; F₃**) is a flowgraph formed by merging the terminal node of **F₁** with the start node of **F₂**. The, merging the terminal node of **F₁; F₂** with the start node of **F₃**



Nesting Practice

Let F_1 and F_2 be two flowgraphs. Then, the F_2 is nested in F_1 at B, (shown by $F_1(F_2)$ on x) is a flowgraph formed by replacing the node B of F_1 with F_2 .

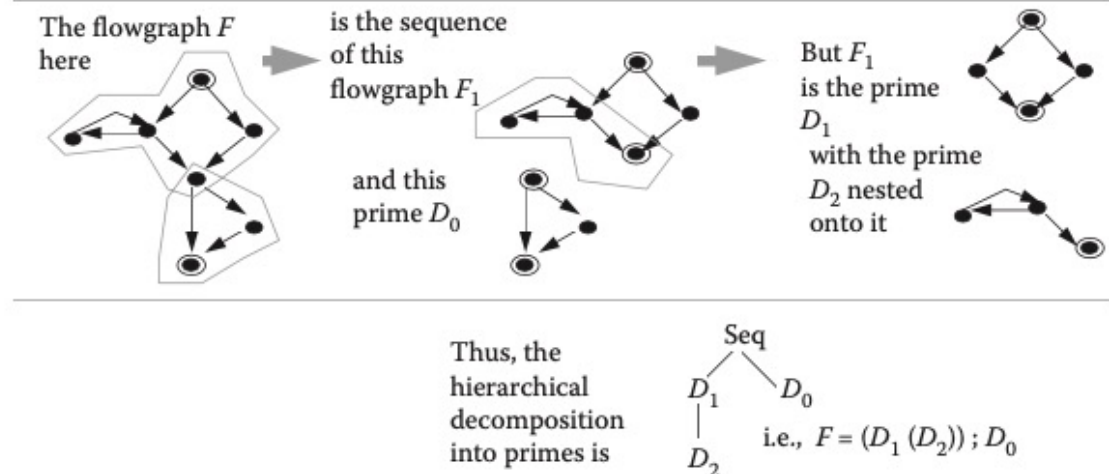


Prime Decomposition

- Functions can be modeled as directed graphs, and built up by composition of the prime/basic flowgraphs
- Any flowgraph can be *uniquely* decomposed into a hierarchy of sequencing and nesting primes, called “decomposition tree”
- Example illustrates how a decomposition tree can be determined from a given flowgraph

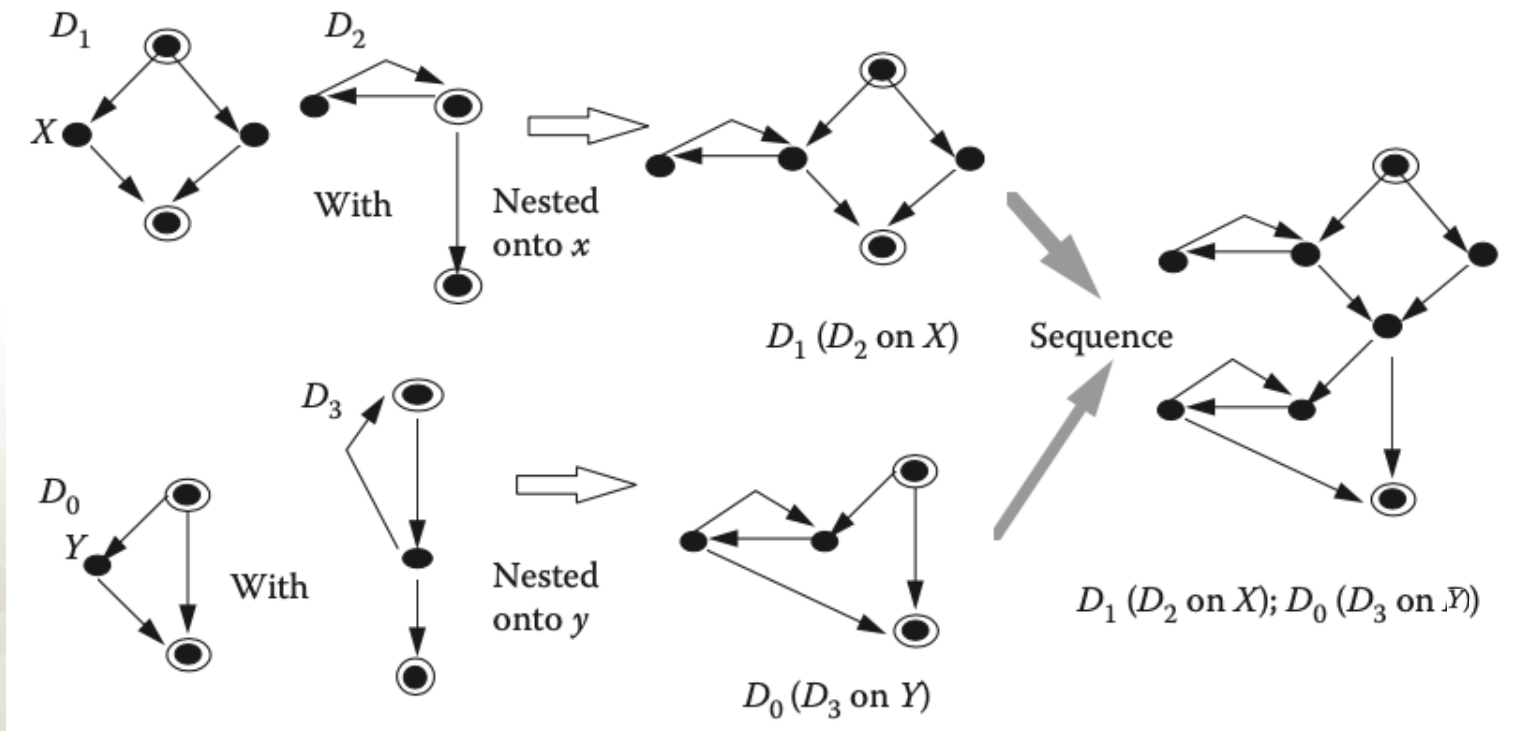
$F = (D_1(D_2)); D_0$

F1; F2
Seq (F1, F2)
P2 (F1, F2)
F1 to F2

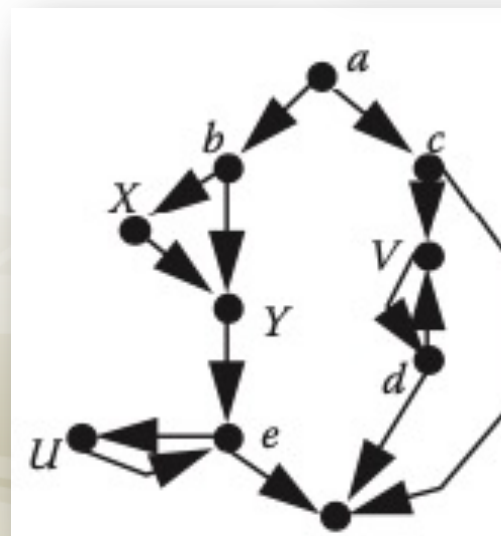
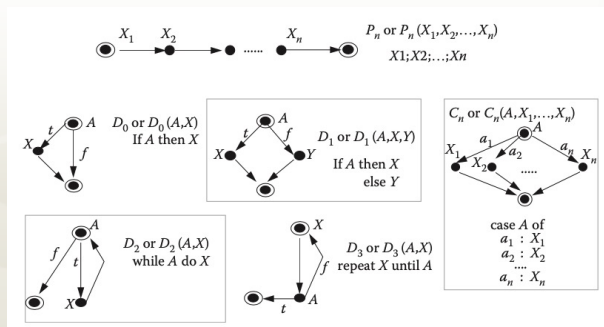
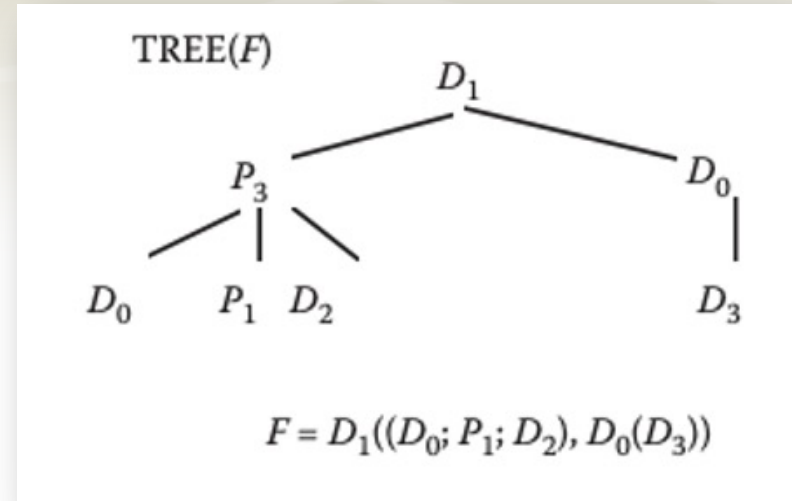
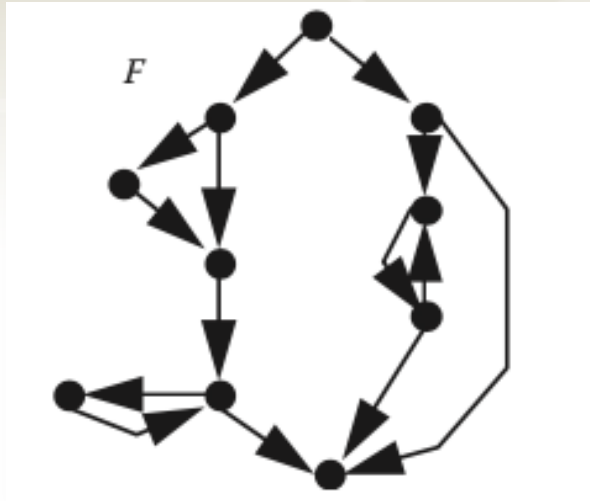


Prime Decomposition

- Construction of a flowgraph from a number of nesting and sequence operations



Prime Decomposition

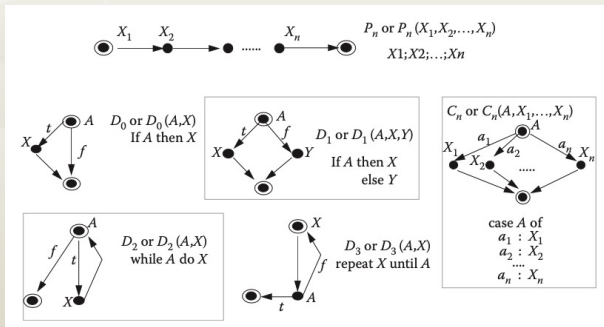
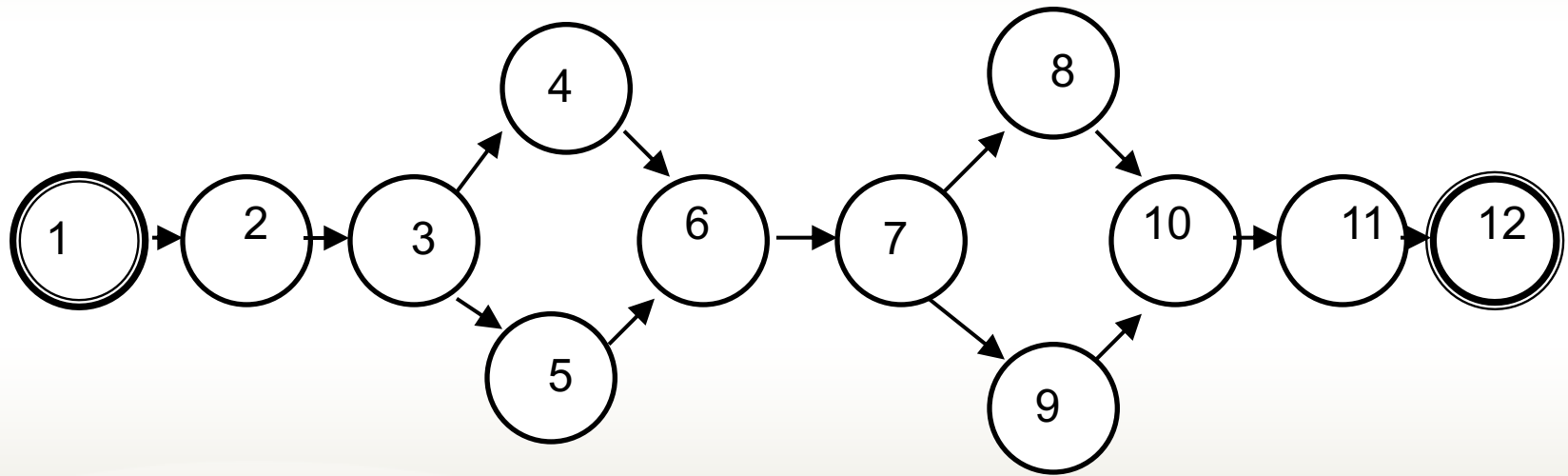


```

if a
then
  begin
    If b then do X;
    Y;
    while e do U
  end
else
  if c
  then do
    repeat V until d
  
```

Using the tree (F), we can recover the program text

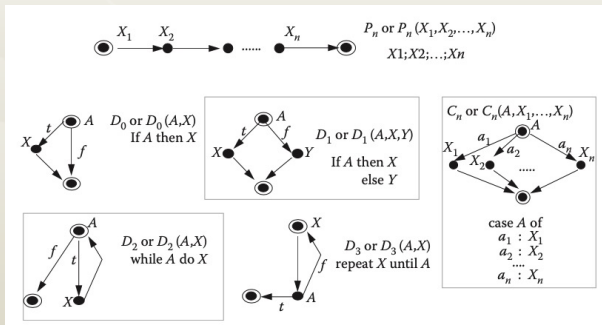
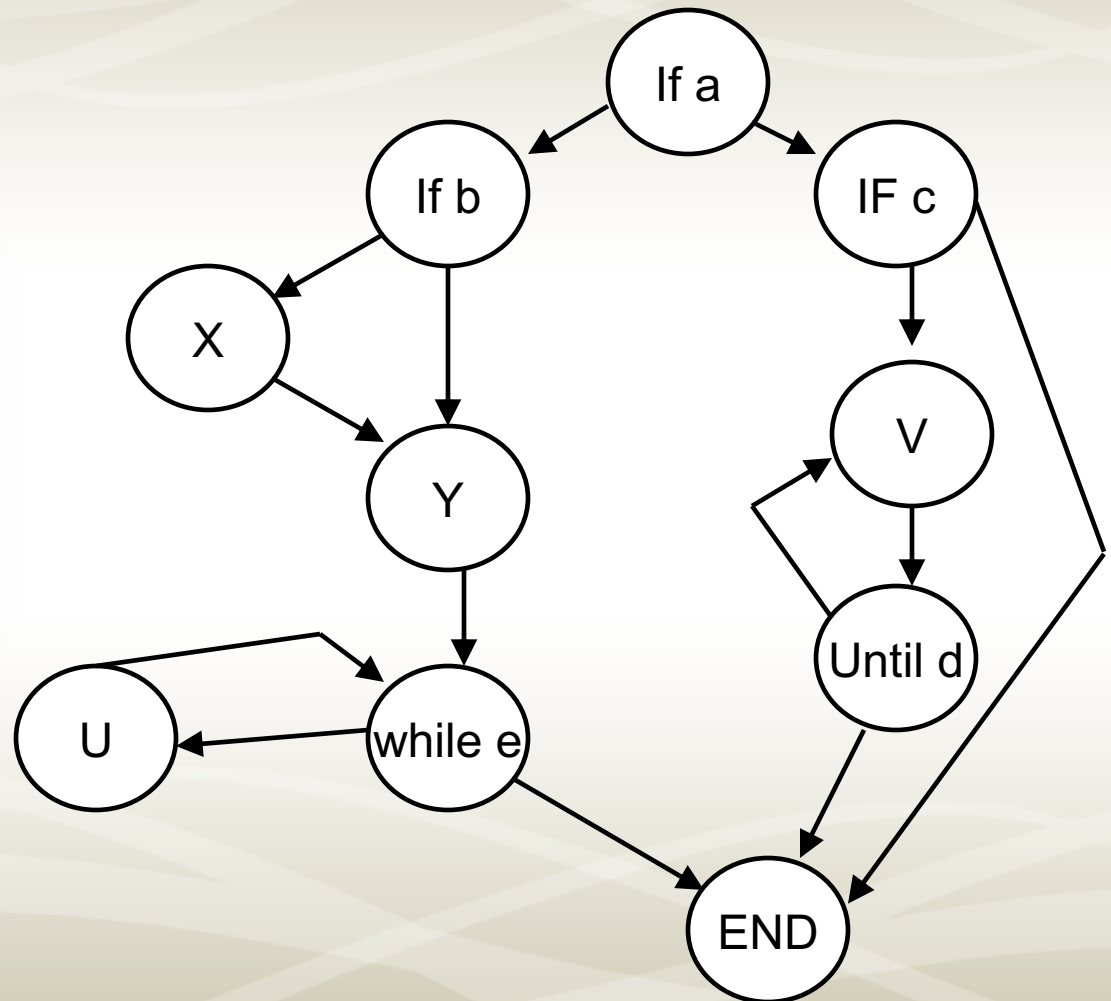
Decompose This



Graph This

```

if a
then
  begin
    If b then do X;
    Y;
    while e do U
  end
else
  if c
  then do
    repeat V until d
  
```



Now, decompose it into a hierarchy of primes

- The defined prime decomposition tree is a definitive description of the control structure of a program
- The decomposition tree is enough to measure a number of program characteristics, including:
 - Nesting factor (depth of nesting)
 - Code size
 - Structural complexity
 - etc.

```
if a
  then
    begin
      If b then do X;
      Y;
      while e do U
    end
  else
    if c
      then do
        repeat V until d
```

Measuring Depth of Nesting

- Depth of nesting $\alpha(F)$ for a flowgraph F can be expressed in terms of:

Primes:

$$\alpha(P1) = 0 ; \alpha(P2) = \alpha(P3) = \dots = \alpha(Pn) = 1$$

$$\alpha(D0) = \alpha(D1) = \alpha(D2) = \alpha(D3) = 1$$

Sequences:

$$\alpha(F1;F2; \dots ;Fn) = \max(\alpha(F1), \alpha(F2), \dots , \alpha(Fn))$$

Nesting:

$$\alpha(F(F1,F2, \dots ,Fn)) = 1 + \max(\alpha(F1), \alpha(F2), \dots , \alpha(Fn))$$

plus one because of the extra nesting level in F

$$F = D1((D0;P1;D2), D0(D3))$$

$$\alpha(F) = 1 + \max(x1, x2)$$

$$x1 = \max(1, 0, 1)$$

$$x2 = 1 + \max(1) = 2$$

$$\alpha(F) = 1 + \max(1, 2)$$

$$\alpha(F) = 3$$

$$\alpha(F) = \alpha(D1((D0;P1;D2), D0(D3)))$$

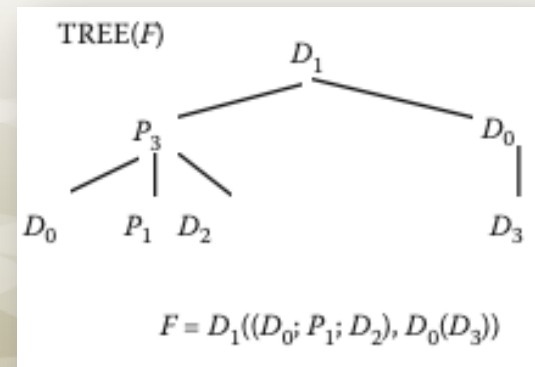
$$= 1 + \max(\alpha(D0;P1;D2), \alpha(D0(D3))) \quad \text{(Nesting rule)}$$

$$= 1 + \max(\max(\alpha(D0), \alpha(P1), \alpha(D2)), 1 + \alpha(D3)) \quad \text{(Sequence rule and nesting rule)}$$

$$= 1 + \max(\max(1, 0, 1), 2)$$

$$= 1 + \max(1, 2)$$

$$= 3$$



Measuring Code size

- Formal size measure, v , captures unambiguously the number of statements in a program

Primes:

$v(P1) = 1$, and for each prime $F \neq P1$, $v(F) = n + 1$, where n is the number of procedure nodes in F

Sequences:

$v(F1; \dots; Fn) = \sum v(Fi)$

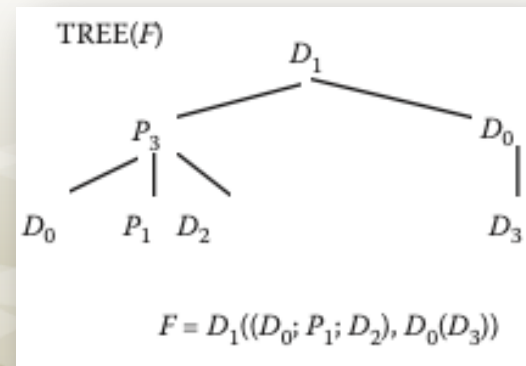
Nesting:

$v(F(F1, \dots, Fm)) = 1 + \sum v(Fi)$ for each prime $F \neq P1$

```

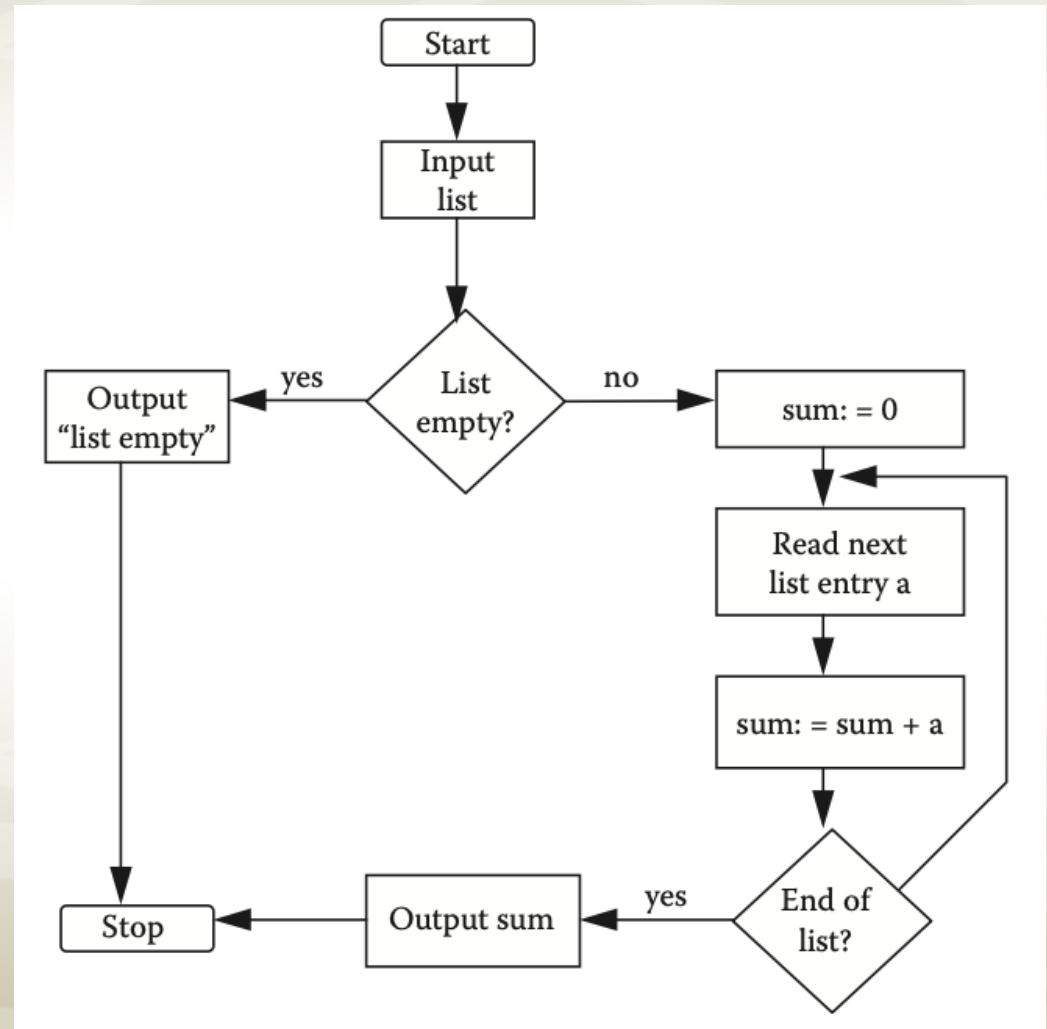
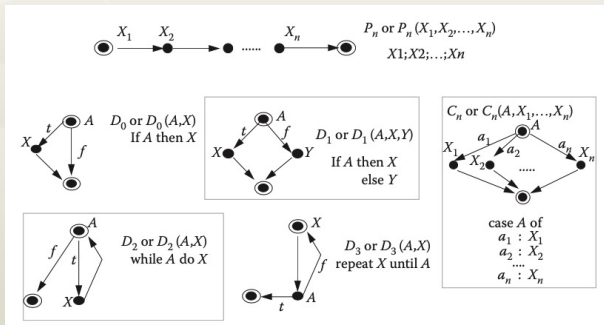
if a
then
  begin
    if b then do X;
    Y;
    while e do U
  end
else
  if c
  then do
    repeat V until d
  
```

$$\begin{aligned}
 v(F) &= v(D_1((D_0; P_1; D_2), D_0(D_3))) \\
 &= 1 + (v(D_0; P_1; D_2) + v(D_0(D_3))) && \text{(Nesting rule)} \\
 &= 1 + (v(D_0) + v(P_1) + v(D_2)) && \text{(Sequence rule)} \\
 &\quad + (1 + v(D_3)) && \text{and nesting rule)} \\
 &= 1 + (2 + 1 + 2) + (1 + 2) \\
 &= 9
 \end{aligned}$$



Exercise - Decompose & Measure

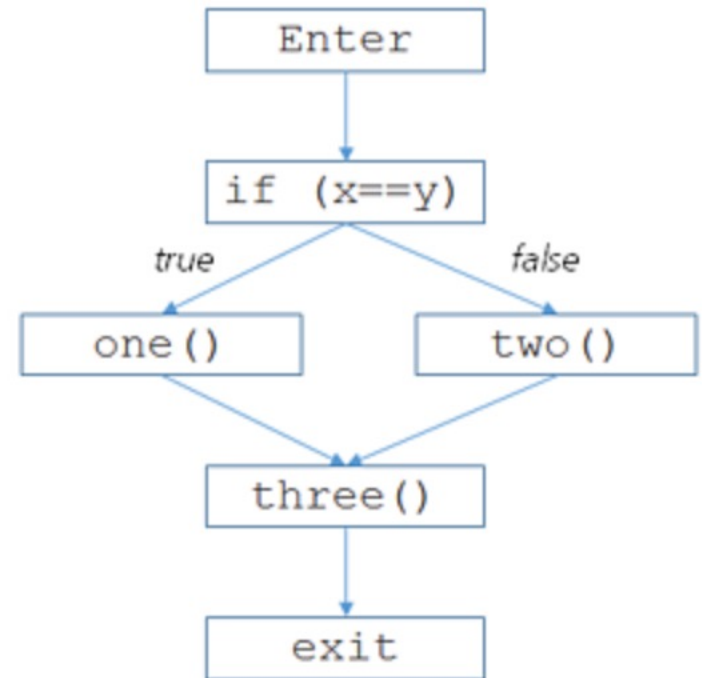
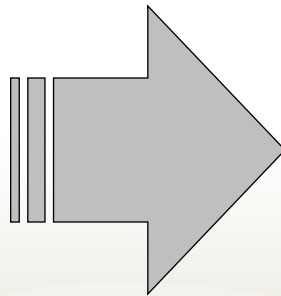
- Decomposition tree
- Decomposition expression
- Depth of Nesting
- Size of Code



Cyclomatic Complexity Measure

- McCabe proposed the cyclomatic number of a program's flowgraph as a measure of program complexity

```
void foo(void)
{
    if (x == y)
        one();
    else
        two();
    three();
}
```

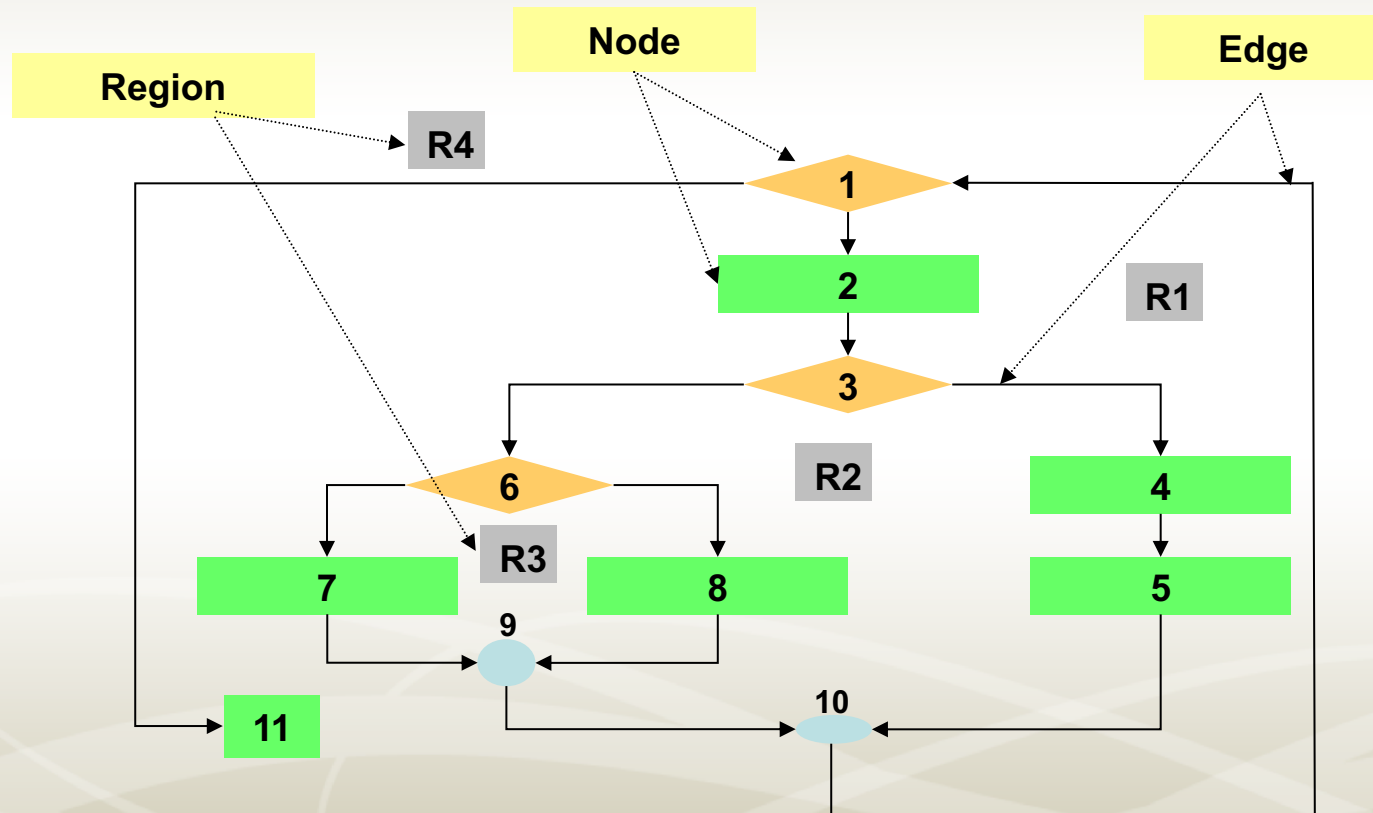


- Cyclomatic complexity is a software metric (measurement) developed by Thomas McCabe. It is used to measure the logical complexity of a program.
- It directly measures the number of linearly independent paths through a program's source code.
- An independent path is a path that has at least one edge which has not been traversed before in any other paths.
- Cyclomatic complexity can be calculated with respect to code blocks, methods or classes within a program.

Cyclomatic Complexity for a flowgraph can be computed using one of the following:

- The numbers of regions (R) of the flow graph correspond to the Cyclomatic complexity.
 $V(G) = R$
- Cyclomatic complexity, $V(G)$, for a flow graph G is defined as
 $V(G) = E - N + 2$
where E is the number of flow graph edges, and N is the number of flow graph nodes
- **$V(G) = P + 1$**
Where P is the number of predicate nodes contained in the flow graph G (nodes that contain condition).

Cyclomatic Complexity Measure



Cyclomatic Complexity Measure

Region, $R = 4$

Number of Predicate Nodes = 3

Number of Nodes (N) = 11

Number of Edges (E) = 13

Cyclomatic Complexity, $V(G)$:

1. $V(G) = R = 4$

Or

2. $V(G) = \text{Predicate Nodes} + 1$

$$= 3 + 1 = 4$$

Or

3. $V(G) = E - N + 2$

$$= 13 - 11 + 2 = 4$$

Cyclomatic Complexity Measure

the cyclomatic number
is a useful indicator of
how difficult a
program or module
will be to test and
maintain

Complexity No.	Corresponding Meaning of $V(G)$
1-10	1) Well-written code, 2) Testability is high, 3) Cost / effort to maintain is low
11-20	1) Moderately complex code, 2) Testability is medium, 3) Cost / effort to maintain is medium.
21-40	1) Very complex code, 2) Testability is low, 3) Cost / effort to maintain is high.
> 40	1) Not testable, 2) Any amount of money / effort to maintain may not be enough.

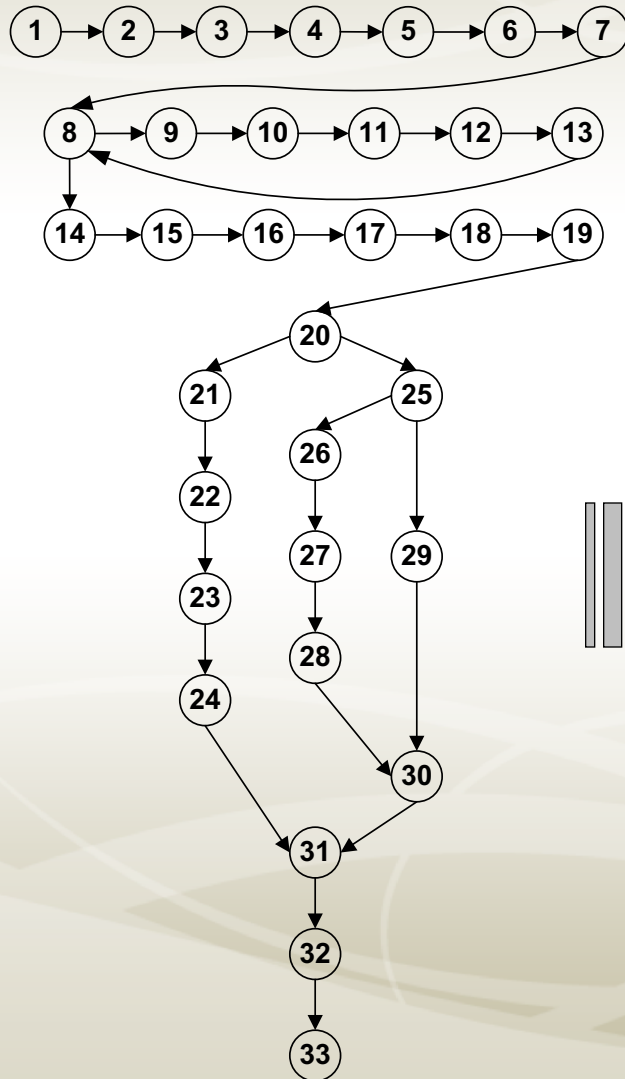
when $v(G)$ is greater than 10 in any one module, the module may be problematic

Cyclomatic Complexity Measure

```
public int getValue(int param) {  
    int value = 0; -----1  
    if (param == 0) { -----2  
        value = 4; -----3  
    }  
    else {  
        value = 0; -----4  
    } -----5  
    return value; -----6  
}
```

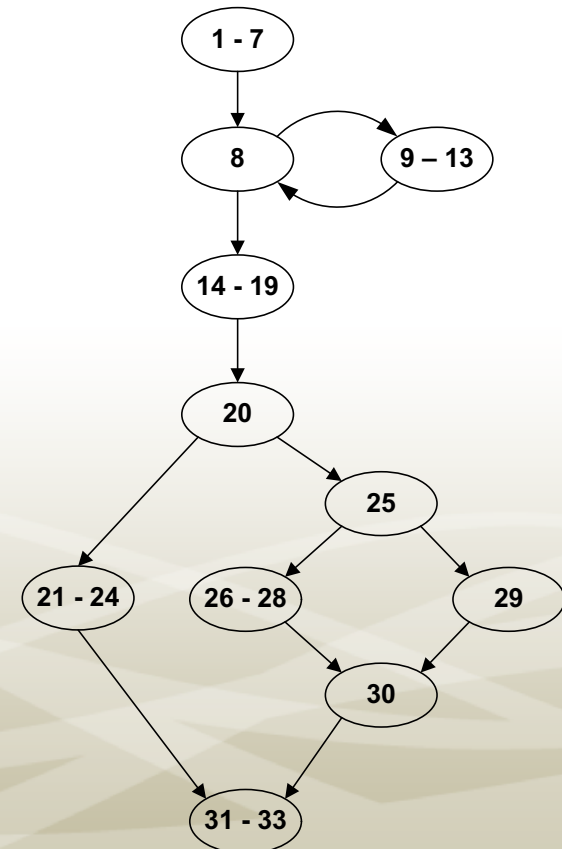
Cyclomatic Complexity Measure

DD-Path Graph

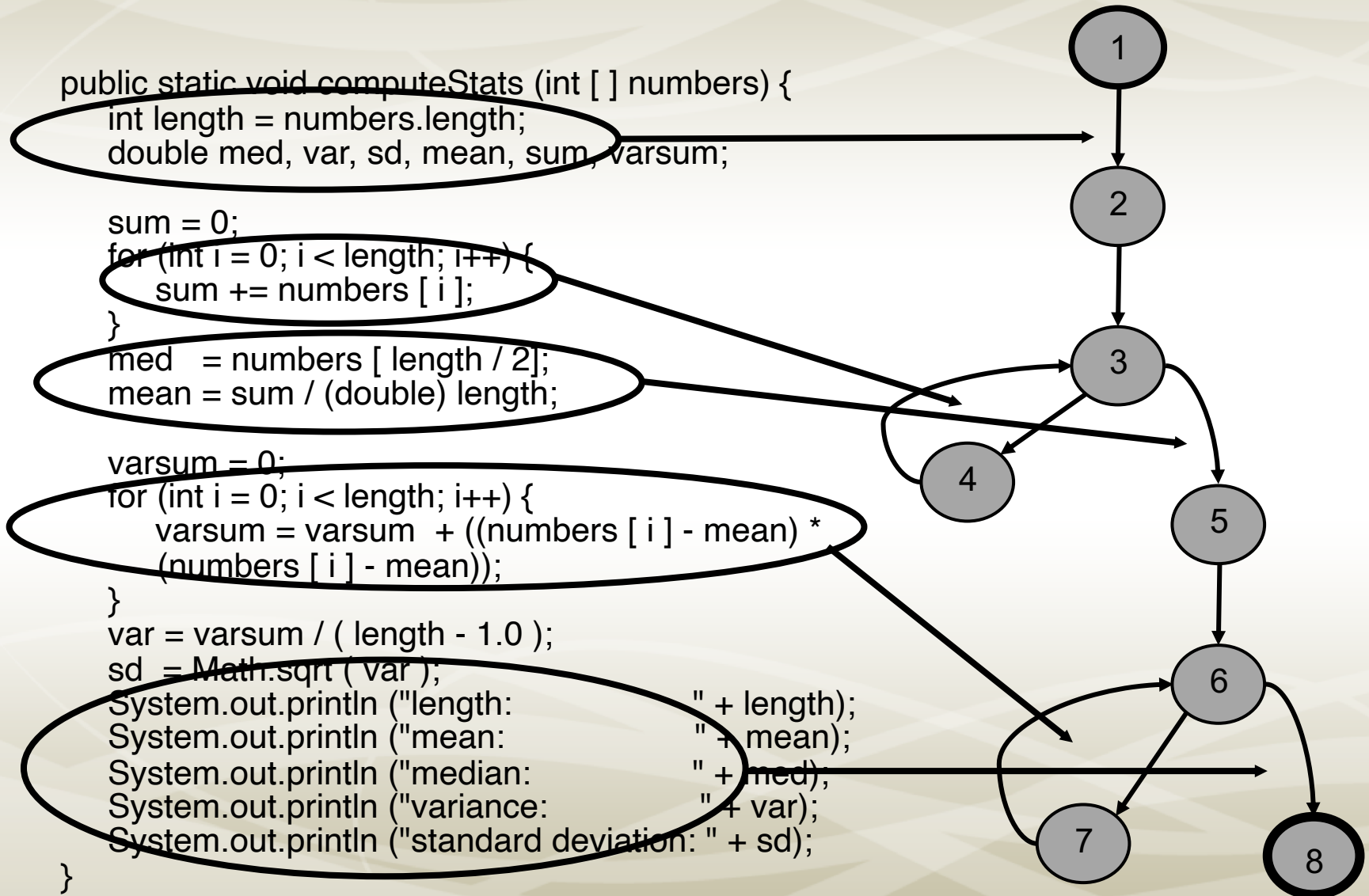


Decision-to-Decision path graph

Concentrate only one decision node - a sequence of nodes are combined in a single node



Exercise



Exercise

```
i = 0;  
n=4;    //N-Number of nodes present in the graph  
while (i < n-1) do  
    j = i + 1;  
    while (j < n) do  
        if A[i] < A[j] then  
            swap(A[i], A[j]);  
        end while;  
    i=i+1;  
end while;
```


Determine cyclomatic complexity for the following Java program:

```
#include <stdio.h>
main()
{
    int a ;
    scanf ("%d", &a);
    if ( a >= 10 )
        if ( a < 20 )        printf ("10 < a< 20 %d\n" , a);
        else                printf ("a >= 20      %d\n" , a);
    else                    printf ("a <= 10      %d\n" , a);
}
```

Exercise

```
1: WHILE NOT EOF LOOP
2:   Read Record;
3:   IF field1 equals 0 THEN
4:     Add field1 to Total
5:     Increment Counter
6:   ELSE
7:     IF field2 equals 0 THEN
8:       Print Total, Counter
9:       Reset Counter
10:    ELSE
11:      Subtract field2 from Total
12:    END IF
13:  END IF
14:  Print "End Record"
15: END LOOP
16: Print Counter
```

Determine cyclomatic complexity for the following Java program:

```
01. import java.util.*;
02. public class CalendarTest
03. {
04.     public static void main(String[] args)
05.     {
06.         // construct d as current date
07.         GregorianCalendar d = new GregorianCalendar();
08.         int today = d.get(Calendar.DAY_OF_MONTH);
09.         int month = d.get(Calendar.MONTH);
10.         // set d to start date of the month
11.         d.set(Calendar.DAY_OF_MONTH, 1);
12.         int weekday = d.get(Calendar.DAY_OF_WEEK);
13.         // print heading
14.         System.out.println("Sun Mon Tue Wed Thu Fri Sat");
15.         // indent first line of calendar
16.         for (int i = Calendar.SUNDAY; i < weekday; i++)
17.             System.out.print(" ");
18.         do
19.         {
20.             // print day
21.             int day = d.get(Calendar.DAY_OF_MONTH);
22.             if (day < 10) System.out.print(" ");
23.             System.out.print(day);
24.             // mark current day with *
25.             if (day == today)
26.                 System.out.print("* ");
27.             else
28.                 System.out.print(" ");
29.             // start a new line after every Saturday
30.             if (weekday == Calendar.SATURDAY)
31.                 System.out.println();
32.             // advance d to the next day
33.             d.add(Calendar.DAY_OF_MONTH, 1);
34.             weekday = d.get(Calendar.DAY_OF_WEEK);
35.         }
36.         while (d.get(Calendar.MONTH) == month);
37.         // the loop exits when d is day 1 of the next month
38.         // print final end of line if necessary
39.         if (weekday != Calendar.SUNDAY)
40.             System.out.println();
41.     }
42. }
```

- The structure of a module is related to the difficulty we find in testing it
- One testing strategy is to select test cases so that every program statement is executed at least once. This approach is called **statement coverage**
- Each test case causes one test path through the program to execute – **the number of test cases, is equivalent to the cyclomatic complexity of the program**
- In terms of the program flowgraph, statement coverage is achieved by finding a set of test cases that execute test paths such that every flowgraph node lies on at least one test path