Preliminary Design Measures

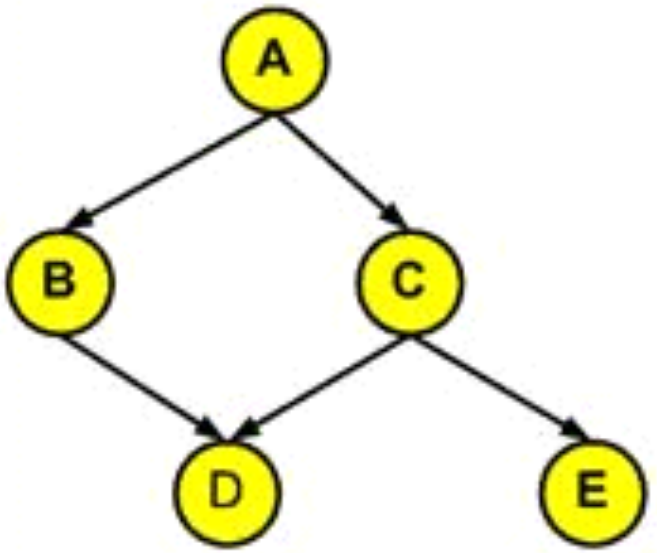
- **Inter-modular measures**, that take into account dependencies among modules, according to the system architecture developed at the design phase
  - Measures of attributes of individual modules are called **intra-modular measures** (used during *detailed design* and *implementation*)
- A **module** is a contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier (Yourdon and Constantine, 1979)
- During the preliminary design phase, decisions concerning system architecture have a major impact on many important qualities of the resultant software, such as ease of implementation, reliability and maintainability
- Preliminary design measures have the potential to provide the necessary feedback on the characteristics of the system being developed

Pre-Design and Code Relationships

- The relationship between preliminary design and code (implemented from the design) include one-to-one relationships between:
  - **modules** indicated in the *design* and **modules** in the code
  - **intermodular connections** indicated in the *design* and **intermodular references** in the *code*
  - **intermodular data interfaces** indicated in the *design* and **intermodular shared data** in the *code*

Modules architecture
(structure chart)

- The modules architecture of a software system can be represented by a graph, S = {N, R}
- Each node *n* in the set of nodes (N) corresponds to a module.
- Each edge *r* in the set of relations (R) indicates a relation (e.g., procedure call, data flow, etc.) between two subsystems.



Modularity

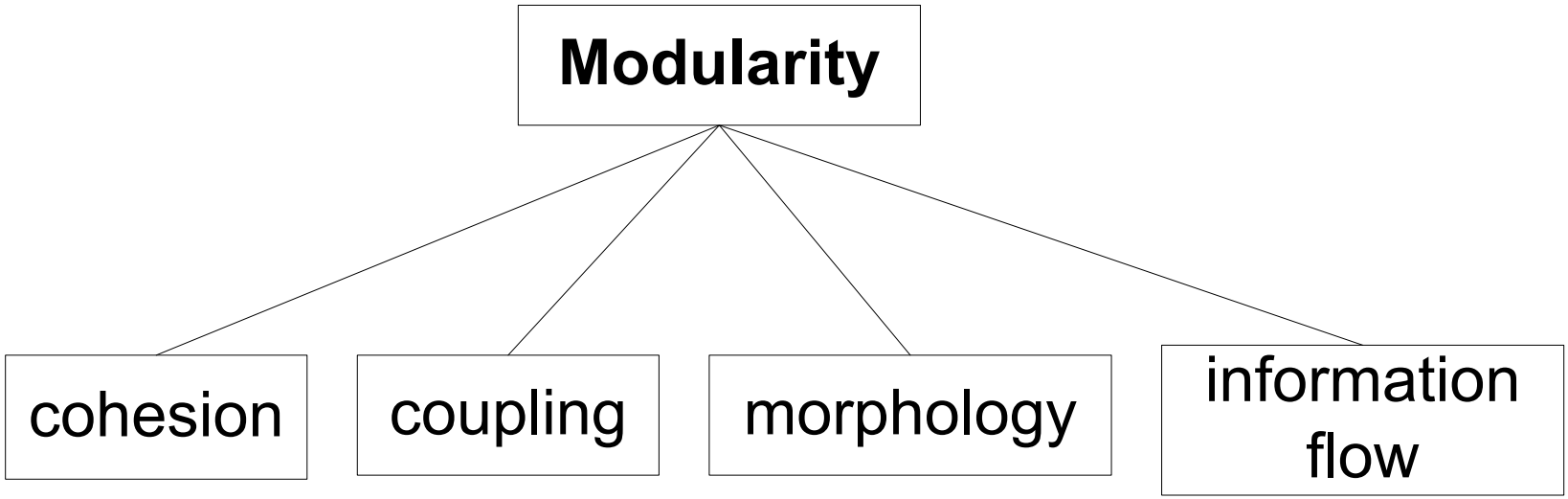
- Def. (IEEE Standard Glossary of Software Engineering Technology):

  the extent to which software is composed of discrete components such that a change to one component has minimal impact on the other components

- **High** modularity is desirable because programs with **low** modularity are believed to be more error prone, less maintainable, less reusable, etc.

Modularity attribute

- Modularity is a quality attribute that can be measured in terms of the following sub-attributes



Modularity sub-attributes (2)

- Cohesion
  - the extent to which an individual module performs a single well-defined task
- Coupling
  - the degree of interdependence between modules
- Morphology
  - the *shape* of the overall system's structure
- Information flow
  - interconnections that a module has with other modules in a system, i.e., the *fan-in* and *fan-out* of the module

Morphology

- Morphology refers to the overall shape of the software system architecture.
- It is characterized by:
  - **Size:** number of nodes and edges
  - **Depth:** longest path from the root to a leaf node
  - **Width:** maximum number of nodes at any level
  - **Edge-to-node ratio:** connectivity density measure

Morphology: Example

Size:

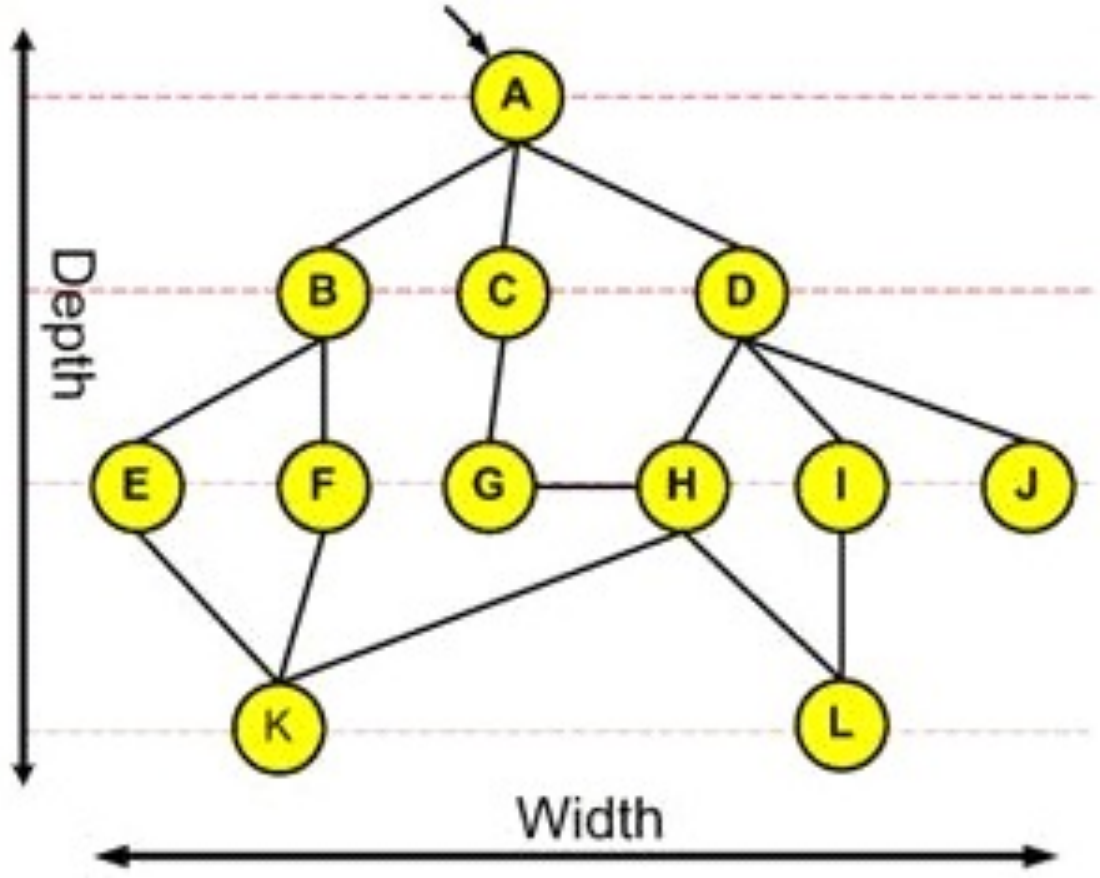
12 nodes

15 edges

Depth: *4*

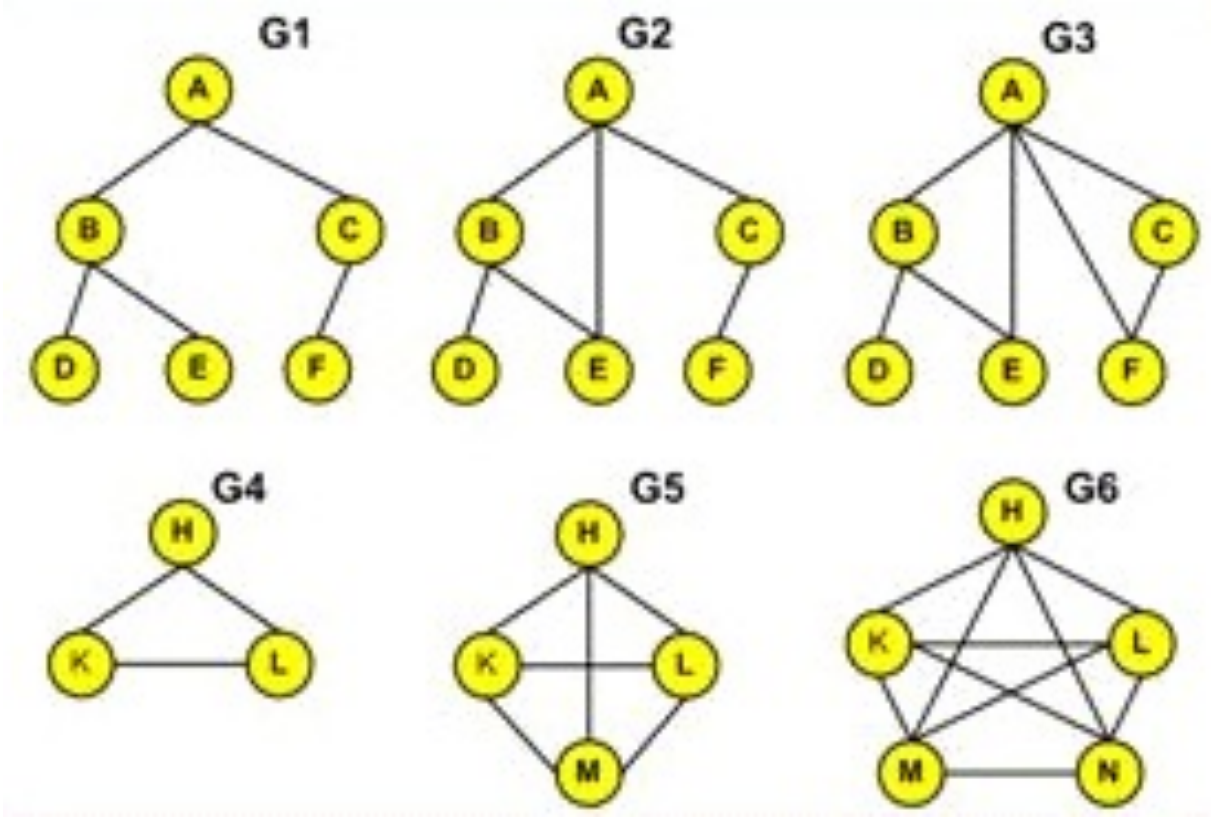
Width: *6*

e/n = 1.25



Tree Impurity

- The tree impurity *m(G)* measures how much the graph *G* is different from a tree (hypothesis: a good design should be as “tree-like” as possible)
- The smaller *m(G)* denotes the better design



Tree Impurity (2)

- Tree impurity can be defined as:

$$m(G) = \frac{\text{number of edges more than spanning tree}}{\text{maximum number of edges more than spanning tree}}$$

$$m(G) = \frac{2(e - n + 1)}{(n-1)(n-2)}$$

- Example

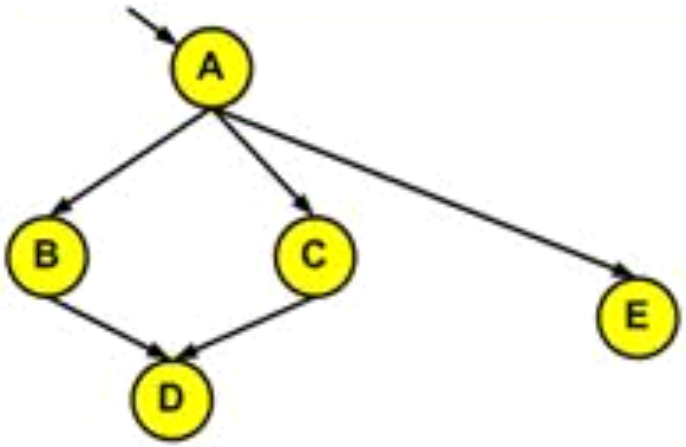
$$m(G_1) = 0 \quad m(G_2) = 0.1 \quad m(G_3) = 0.2$$

$$m(G_4) = 1 \quad m(G_5) = 1 \quad m(G_6) = 1$$

Internal Reuse
(Yin and Winchester measure)

- Internal reuse is a measure, proposed by Yin and Winchester, indicating the extent of reuse of modules *within the same product* (different from *external* reuse).

  $r(G) = e - n + 1$
- The smaller *r(G)* means less reuse
- **Critics:** cannot account for repetitive calls and cannot account for the size of reused module



Module D is reused by modules B and C

Example:

r(G1) = 0	*r(G2) = 1*
r(G3) = 2	*r(G4) = 1*
r(G5) = 3	*r(G6) = 6*

Information flow

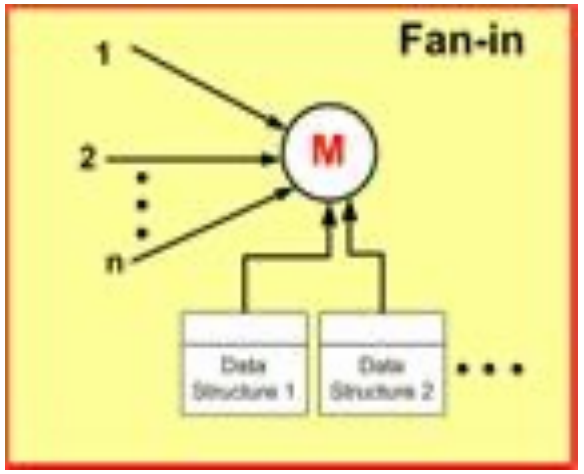
- Information flow measures assume that the complexity of a module depends on 2 factors:
  - the complexity of the module code
  - the complexity of the module's interfaces, i.e., its connections to its environment
- The total level of information flow through a system, where the modules are viewed as atomic components, is an *inter-modular attribute*
- The total level of information flow between an individual module and the rest of the system is an *intra-modular attribute*

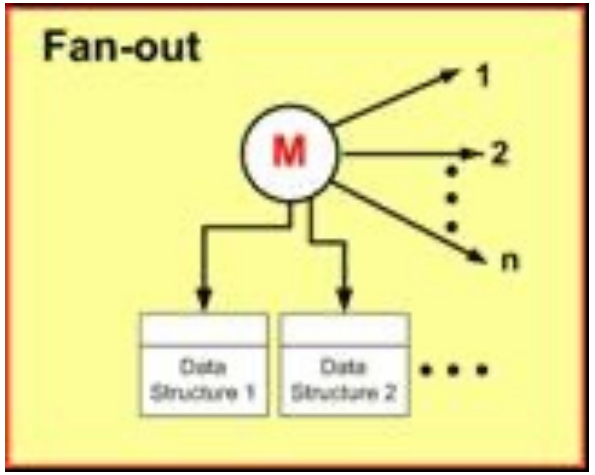
Information Flow Measures

- Information flow measures are counts based on the interconnections that a module has with other modules in a system, i.e., the *fan-in* and *fan-out* of a module
- Information flow measures are based on:
  - **local** flow of information:
    - **direct**: one module calls another module
    - **indirect**: flow arising from return values
  - **global** flow of information: information that is passed between modules via a global data structure
- Information flow measures:
  - can be used to identify the critical parts of a software system
  - are thought to identify stress points in the system
  - provide an insight into potential design problems

Fan-in and Fan-out

- **Fan-in** of a module M is the number of (direct + indirect) local flows terminating at M plus the number of global flows (data structures) from which info is retrieved by M.
- **Fan-out** of a module M is the number of (direct + indirect) local flows starting at M plus the number of global flows (data structures) updated by M.





Fan-in and Fan-out Values

- High fan-out of a module indicates it influences/controls many other modules
- High fan-in of a module indicates that it is influenced/controlled by many other modules
- A module with high fan-in and fan-out is normally at the heart of the system
- A module with low fan-in and fan-out is normally at the periphery of the system

Fan-in and Fan-out Values (2)

- High fan-in and fan-out indicates an often complex module that may be error-prone because:
  - *the modules perform more than one function*. If the design structure needs to be changed, the aspect to be changed will not be found in a single place but in several places and everything to do with it is not within a particular module but with several interconnected (and hence having high fan-in and fan-out) modules. Therefore, many modules will need altering.
  - *the modules are inadequately refined i.e., the system's structure could be missing a level of abstraction* (if the structure is too wide and less deep or the reverse the modules could be said to be inadequately refined).

Information Flow Measure
(Henry & Kafura)

- Information flow (IF) for module M_i [Henry-Kafura, 1981]:

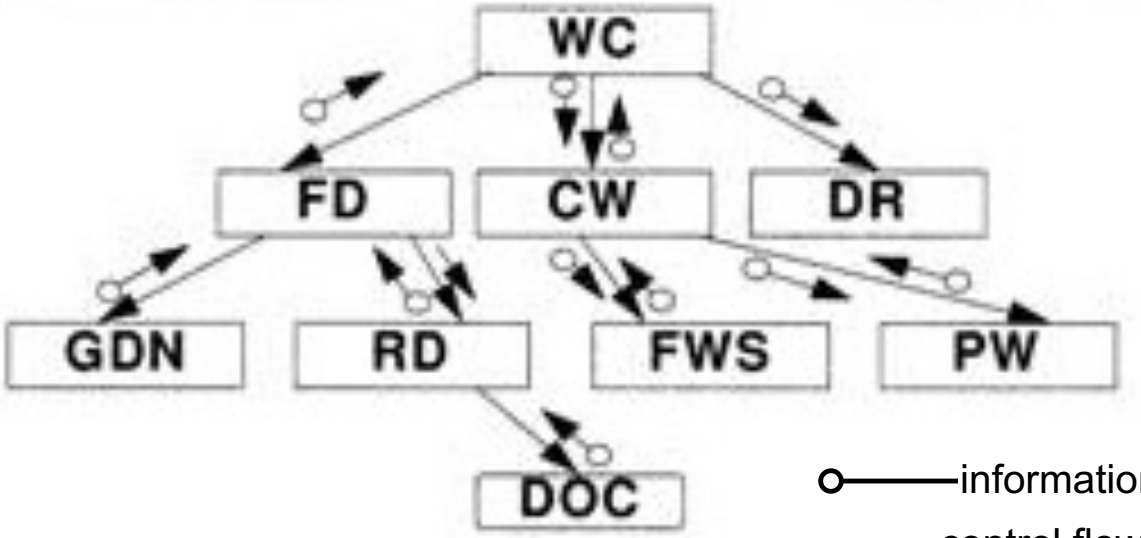
$$IF(M_i) = [\ fan\text{-}in(M_i)\ x\ fan\text{-}out(M_i)\ ]^2$$

- IF for a system with *n* modules is:

$$IF = \sum_{i=1}^{n} IF(M_i)$$

- The original Henry-Kafura IF measure includes both control flows and information flows when counting fan-in and fan-out
- The Shepperd variant includes only information flows

Information Flow Measure: example



information flow

control flow (calls)

Module	fan-in	fan-out	[(fan-in)(fan-out)]^2
WC	2	2	16
FD	2	2	16
CW	3	3	81
DR	1	0	0
GDN	0	1	0
RD	2	1	4
FWS	1	1	1
PW	1	1	1

IF = 119

(*Shepperd* variant)

Structural Measurement
(at detailed design and implementation)

- Structure can have 3 components:
  - **Control-flow structure:** Sequence of execution of instructions of the program
  - **Data flow:** Keeping track of data as it is created or handled by the program
  - **Data structure:** The organization of data itself independent of the program
- *Structural measurement* is used in software tools for:
  - reverse engineering
  - testing (path coverage)
  - code restructuring
  - data flow analysis

Goal & Questions …

- How to represent “structure” of a program?
  - control flowgraph (or simply flowgraph)
- How to define “complexity” in terms of the structure?
  - cyclomatic complexity

Basic Control Structure

- Basic Control Structures (BCSs) are set of essential control-flow mechanisms used for building the logical structure of the program.
- BCS types:
  - Sequence: e.g., a list of instructions with no other BCSs involved.
  - Selection: e.g., *if … then … else*.
  - Iteration: e.g., *do … while* ; *repeat ... until*.
- There are other types of control structures, or Advanced Control Structures (ACSs):
  - Procedure/function/agent call
  - Recursion (self-call)
  - Interrupt
  - Concurrence

Flowgraph

- Control flow structure is usually modeled by a *flowgraph FG*, i.e., a directed graph (di-graph)

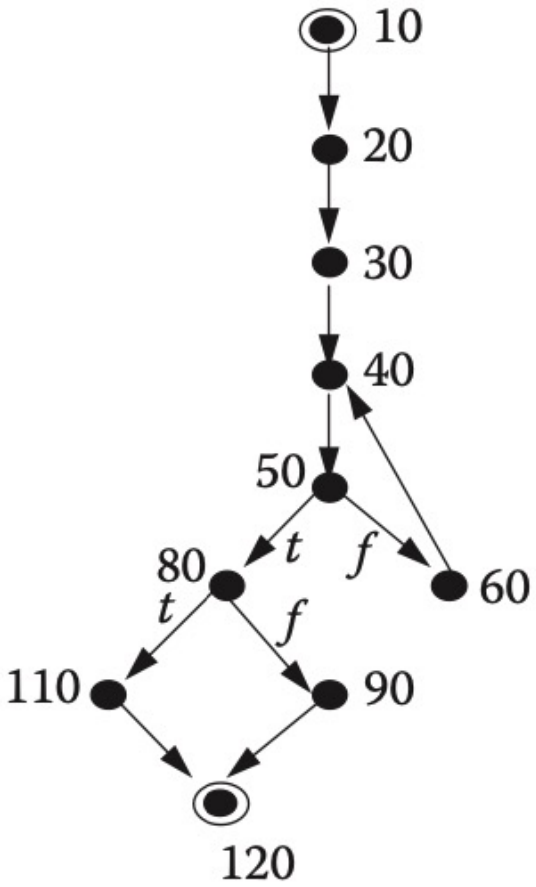
$$FG = \{N, E\}$$

- Each node *n* in the set of nodes (N) corresponds to a program statement
  - **Procedure nodes:** nodes with out-degree 1
  - **Predicate nodes:** nodes with out-degree other than 1
  - **Start node:** nodes with in-degree 0
  - **Terminal (end) nodes:** nodes with out-degree 0

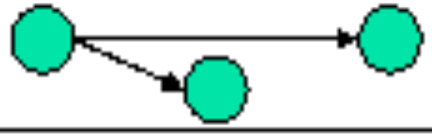
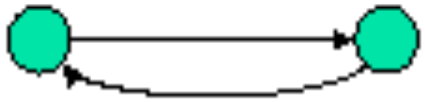
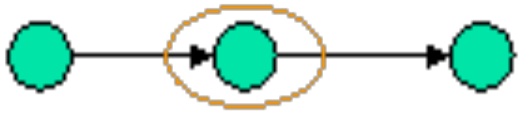
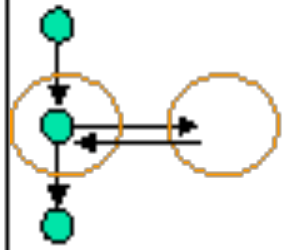
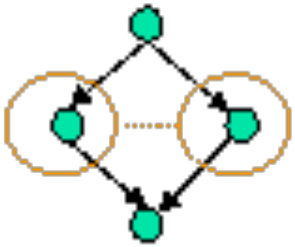
  where *out-degree* is the number of edges leaving a node and *in-degree* is the number of edges entering a node
- Each directed edge (or directed arc) *e* in the set of edges (E) indicates flow of control from one program statement to another statement

Flowgraph example

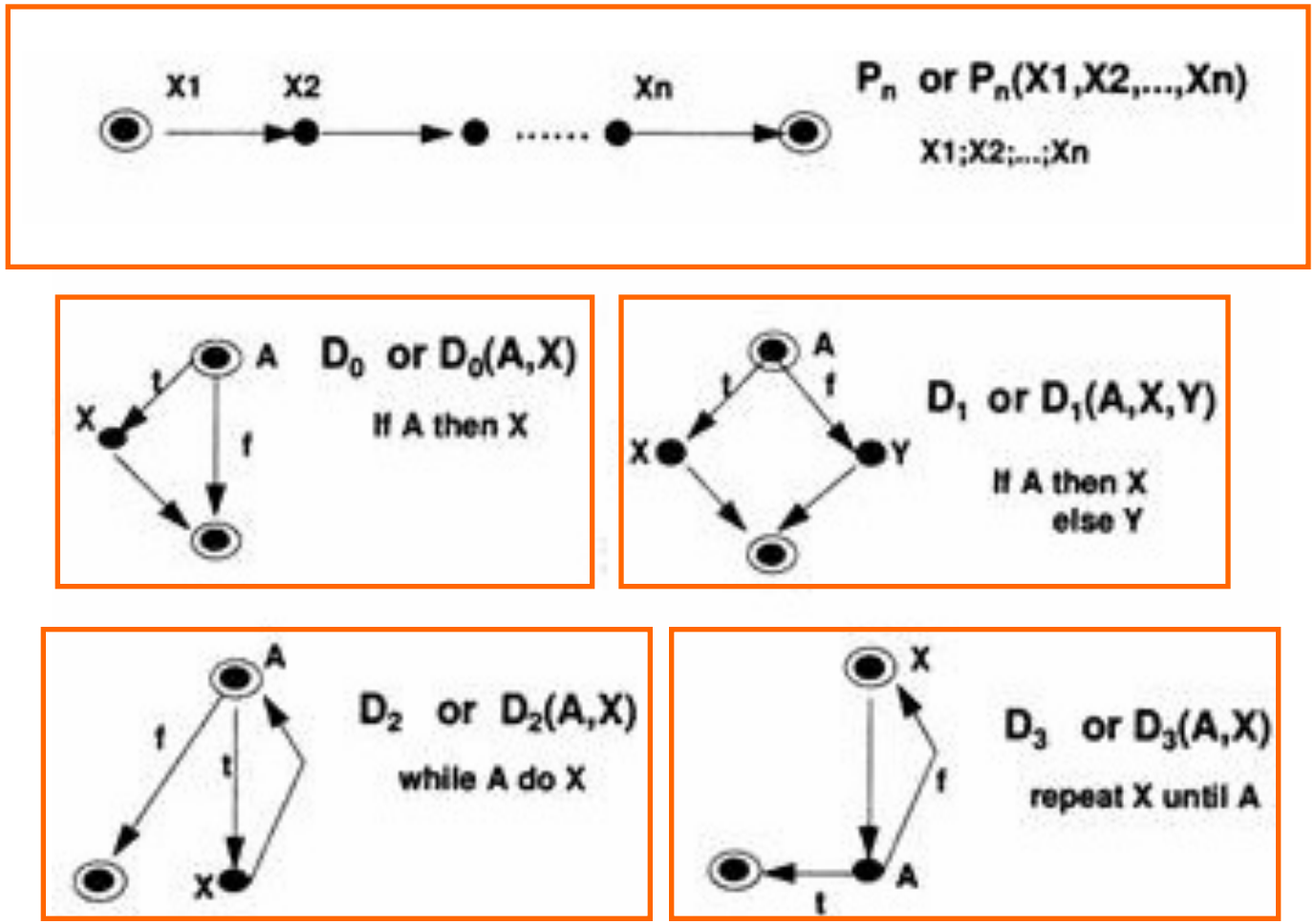
```
10  INPUT P
20  Div = 2
30  Lim = INT(SQR(P))
40  Flag = P/Div - INT(P/Div)
50  IF Flag = 0 OR Div = Lim THEN 80
60  Div = Div + 1
70  GO TO 40
80  IF Flag <>0 OR P>4 THEN 110
90  PRINT Div; "Smallest factor of"; P; "."
100 GO TO 120
110 PRINT P; " is prime"
120 END
```



Flowgraph constructs

Basic CS	Sequence	
	Selection	
	Iteration	
Advanced CS	Procedure/ function call	
	Recursion	
	Interrupt 	Concurrence 

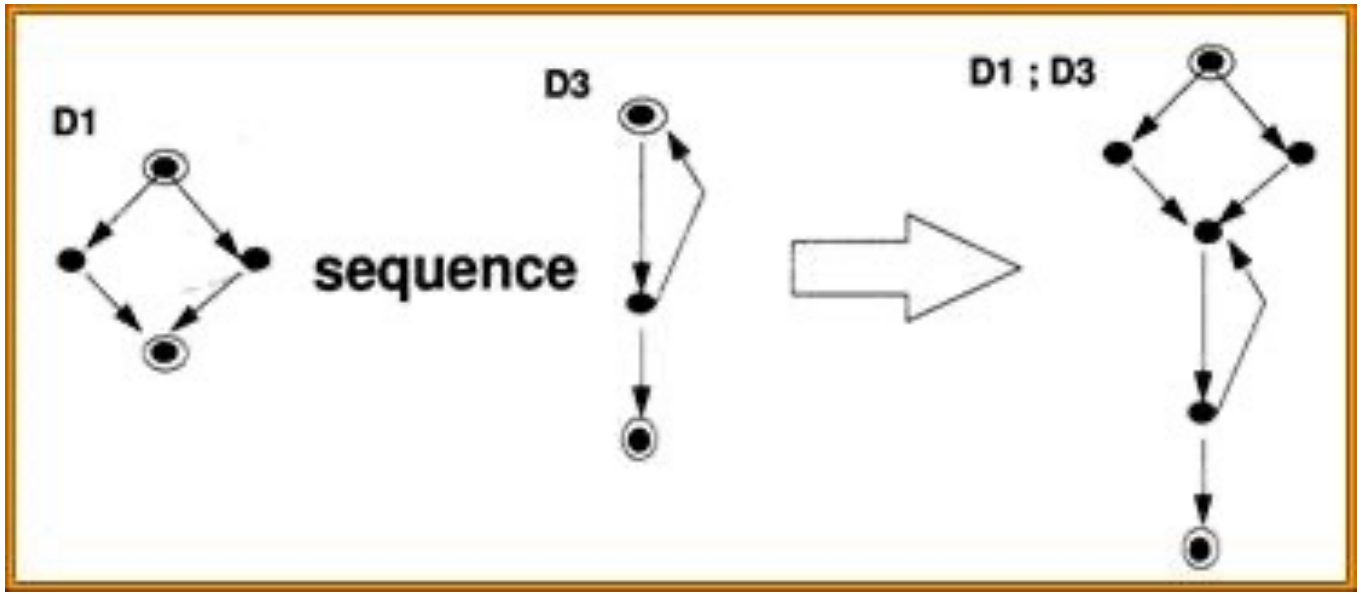
Common Flowgraph Program Models



Encircled nodes distinguish terminal nodes (*start* and *stop*)

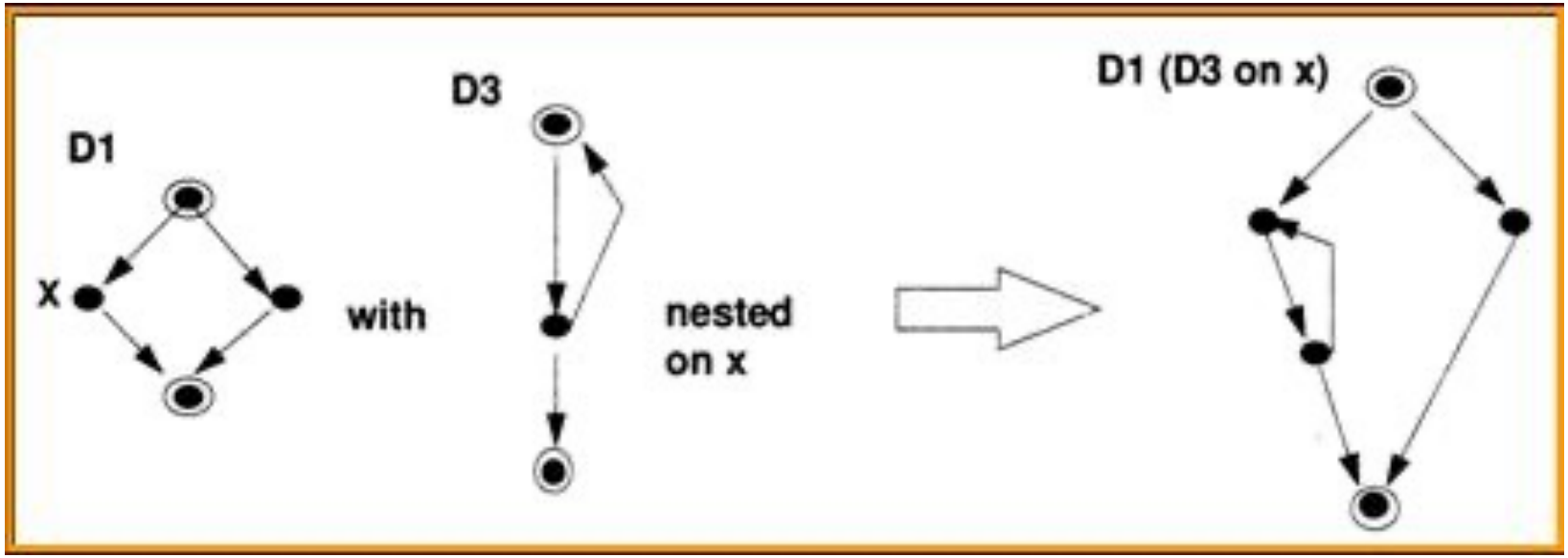
Sequencing

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



Nesting

- Let $\mathbf{F_1}$ and $\mathbf{F_2}$ be two flowgraphs. Then, the nesting of $\mathbf{F_2}$ onto $\mathbf{F_1}$ at *x*, shown by $\mathbf{F_1(F_2)}$, is a flowgraph formed from $\mathbf{F_1}$ by replacing the arc from *x* with the whole of $\mathbf{F_2}$



Prime Flowgraphs

- Prime flowgraphs are flowgraphs that cannot be decomposed non-trivially by sequencing and nesting.
- **Common primes:**
  - P_1
  - D_0
  - D_1
  - D_2
  - D_3

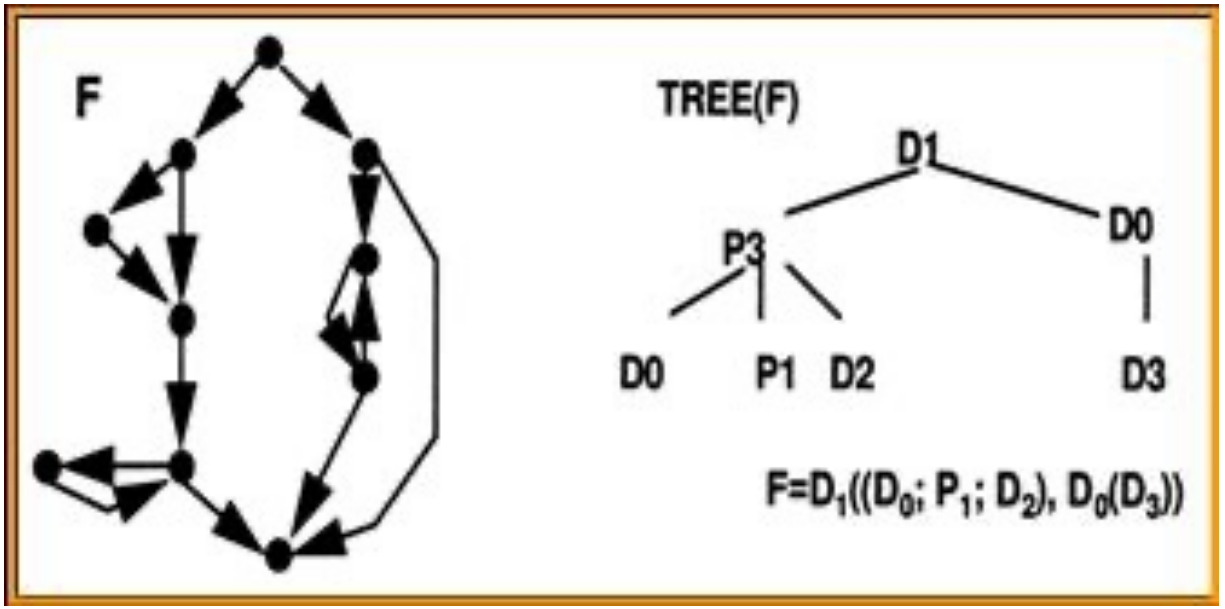
D-structures
(typical of structured programming, "D" is the initial of Edsger Dijkstra who pioneered structured programming)

Prime Decomposition

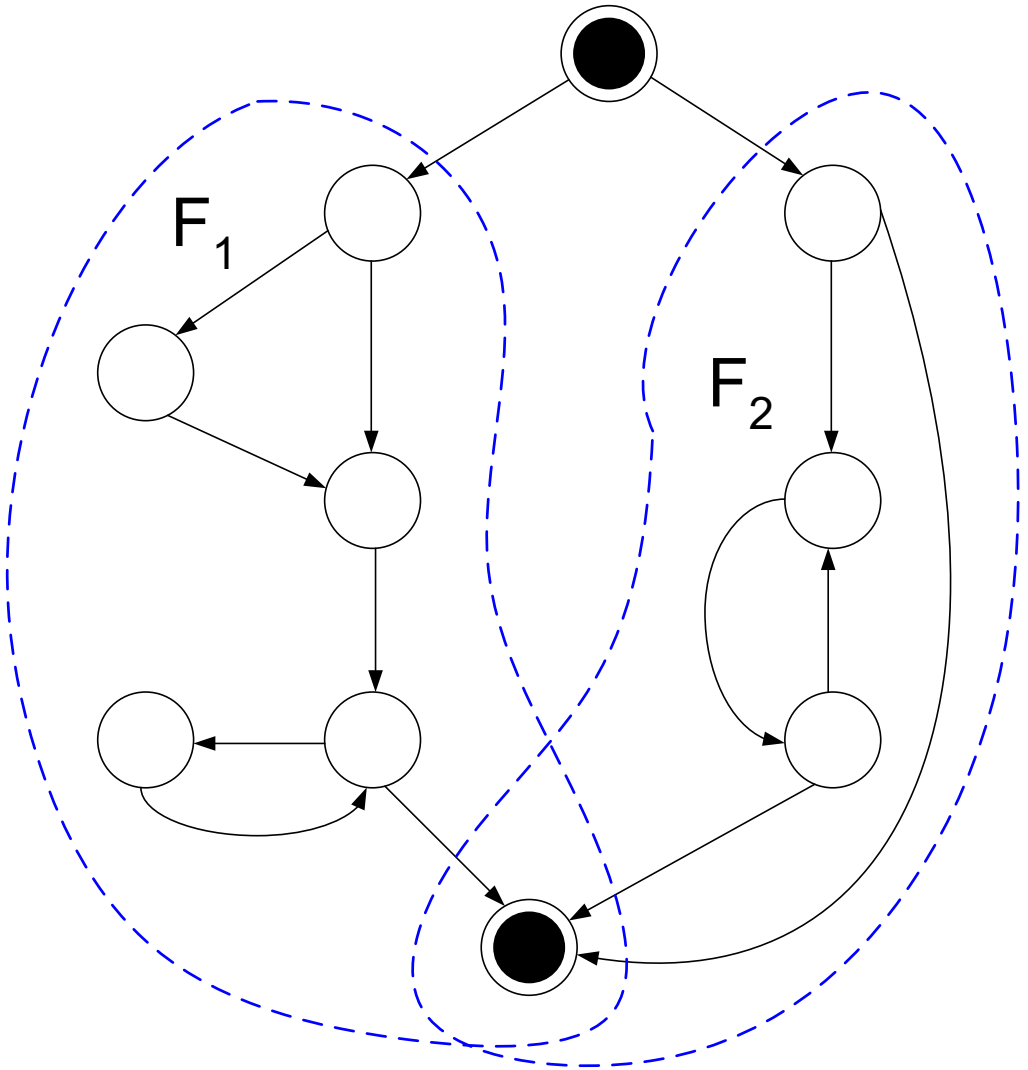
Prime Decomposition Theorem (Fenton-Whitty)

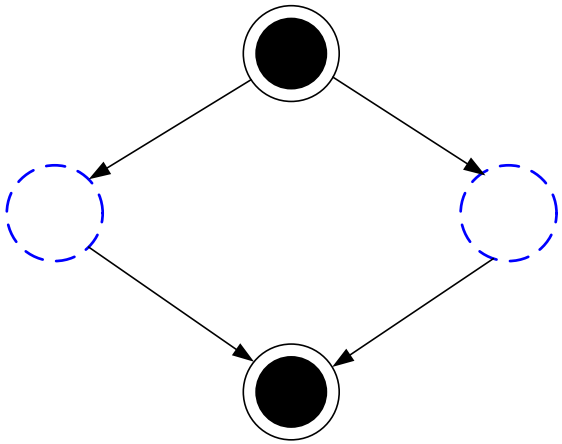
Every flowgraph has a *unique* decomposition into a hierarchy of primes, called "**decomposition tree**".

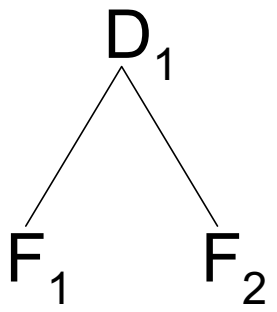
```
if a
   then
      begin
         if b then do X;
         Y;
         while e do U
      end
   else
      if c
         then do
            repeat V until d
```



Decomposition – step 1

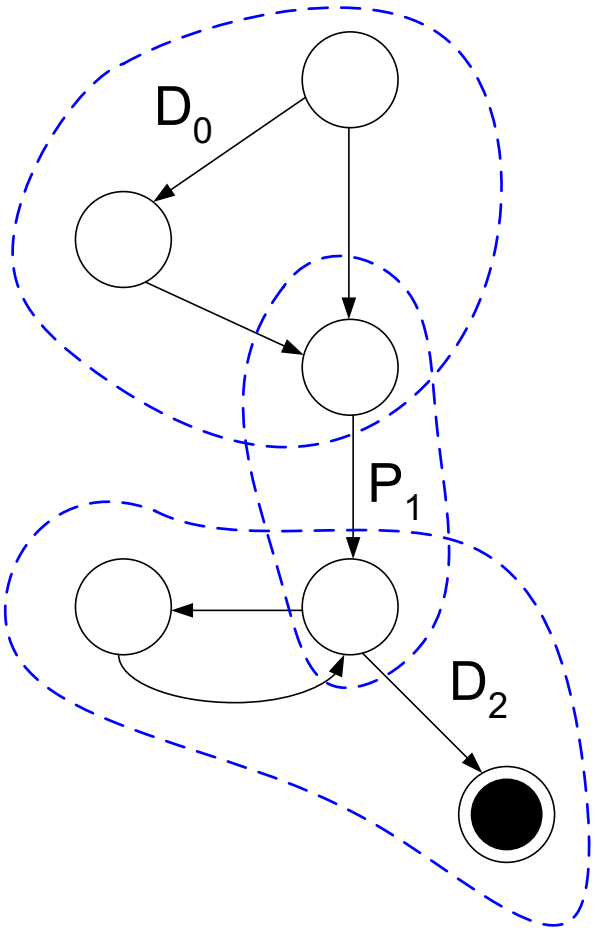


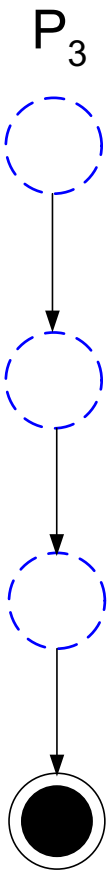


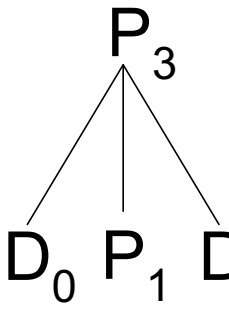


$F=D_1(F_1,F_2)$

Decomposition – step 2a

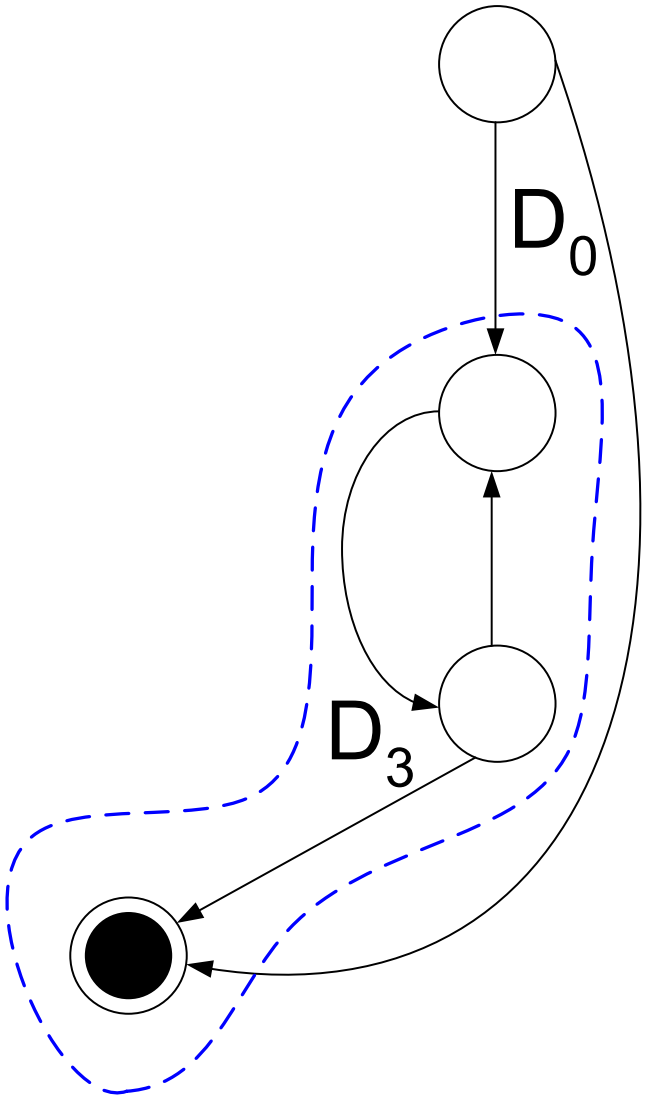


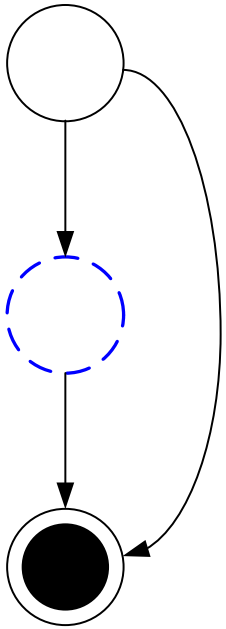




$F_1 = P_3(D_0, P_1, D_2)$

Decomposition – step 2b



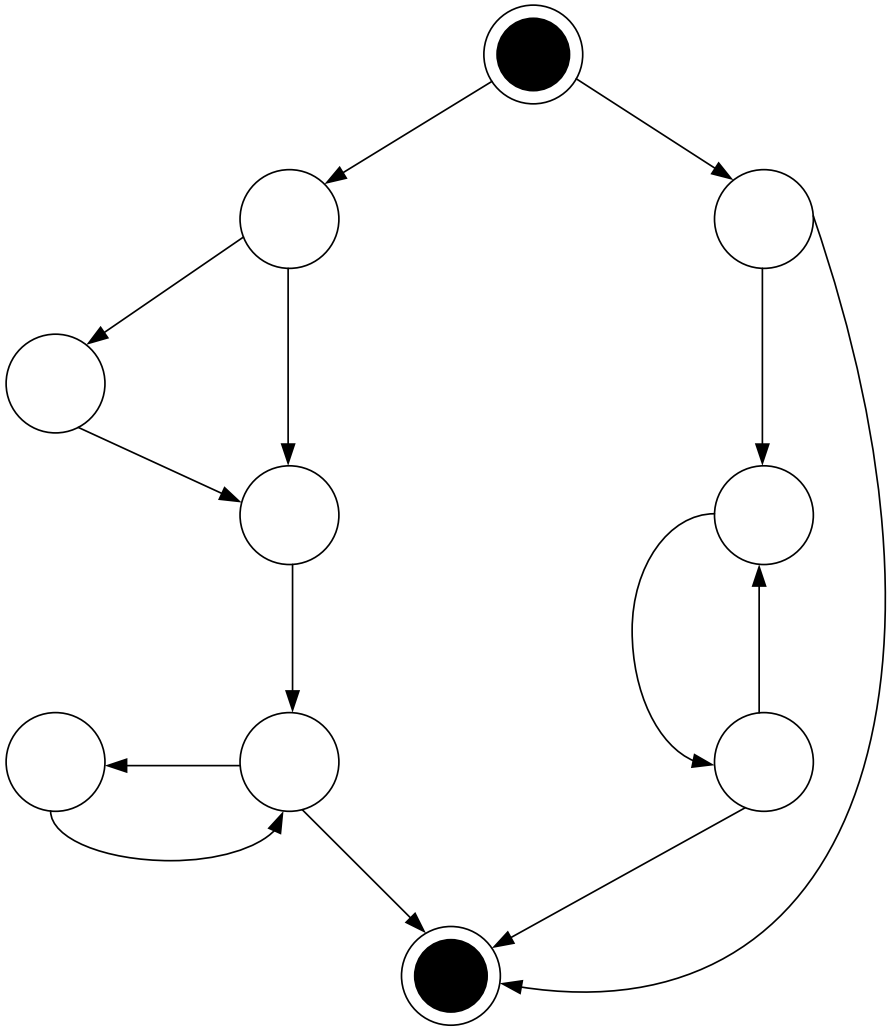


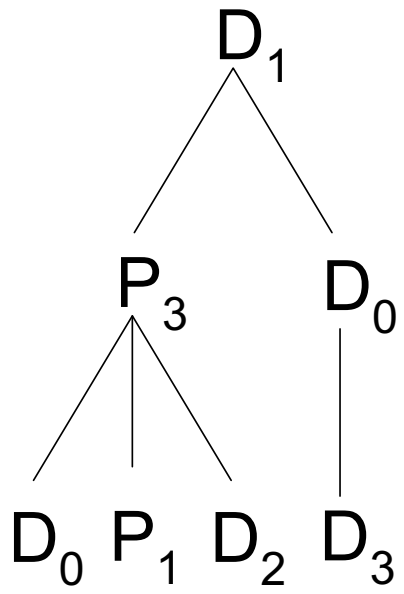
D_0

D_3

$F_2=D_0(D_3)$

Decomposition – step 3





$F=D_1(F_1,F_2)$

$F=D_1(P_3,D_0(D_3))$

$F=D_1((D_0;P_1;D_2),D_0(D_3))$

Hierarchical Measurement

- Hierarchical measurement is a way of defining flowgraph measures using the decomposition tree
- It is based on the idea that we can measure a flowgraph attribute by:
  1. defining the measure for *prime* flowgraphs
  2. describing how the *sequencing* operation affects the attribute
  3. describing how the *nesting* operation affects the attribute
- Example flowgraph measures that can give information about the complexity of code structure are:
  - **Depth of nesting**
  - **D-structuredness**

Depth of Nesting

- Depth of nesting *n(F)* for a flowgraph *F* can be measured in terms of:
- **Primes:**

  $n(P_1) = 0 ;\ n(P_2) = n(P_3) = \ldots = n(P_k) = 1$

  $n(D_0) = n(D_1) = n(D_2) = n(D_3) = 1$

- **Sequencing:**

  $n(F_1;F_2;\ldots;F_k) = max\{ n(F_1), n(F_2), \ldots, n(F_k) \}$

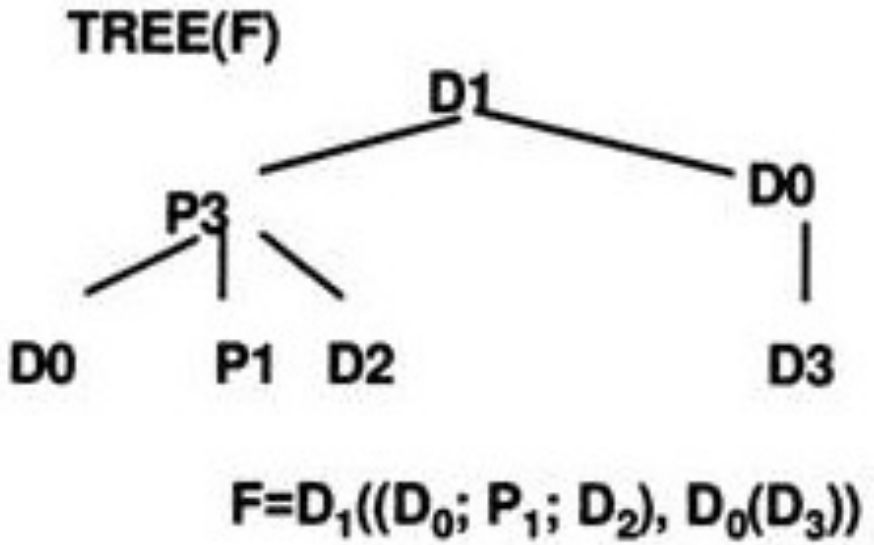
- **Nesting:**

  $n(F(F_1,F_2,\ldots,F_k)) = 1 + max\{n(F_1), n(F_2), \ldots, n(F_k) \}$

Depth of Nesting: example

- **Example:**

  $F = D_1((D_0;P_1;D_2), D_0(D_3))$



$n(F) = n(D_1((D_0;P_1;D_2), D_0(D_3))) =$

$= 1 + max\{ n(D_0;P_1;D_2), n(D_0(D_3)) \} =$

$= 1 + max\{ max\{ n(D_0), n(P_1), n(D_2) \}, 1+n(D_3) \} =$

$= 1 + max\{ max\{ 1,0,1 \}, 2 \} = 1 + max\{ 1,2 \} =$ ***3***

D-Structuredness

- Most popular definitions of stuctured programming assert that a program is structured if it can be composed using only a small number of allowable constructs (e.g., sequence, selection and iteration)
- By considering the flowgraph of a program, we can decide whether or not it is structured
- To this purpose the *D-structuredness* measure can be used
- The informal definition of structured programming can be formally expressed by asserting that a program is structured iff it is *D-structured*

D-Structuredness (2)

- D-structuredness *d(F)* for a flowgraph *F* can be measured in terms of:
- **Primes:**

  $d(P_1) = 1$; $d(D_0) = d(D_1) = d(D_2) = d(D_3) = 1$

  0 otherwise

- **Sequencing:**

  $d(F_1;F_2;\ldots;F_k) = min\{ d(F_1), d(F_2), \ldots, d(F_k) \}$

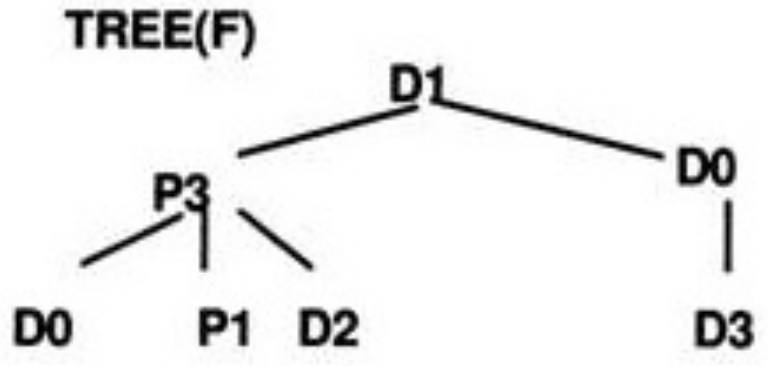
- **Nesting:**

  $d(F(F_1,F_2,\ldots,F_k)) = min\{ d(F), d(F_1), d(F_2), \ldots, d(F_k) \}$

D-Structuredness: example

- **Example:**

  $F = D_1((D_0;P_1;D_2), D_0(D_3))$



$d(F) = d(D_1((D_0;P_1;D_2), D_0(D_3))) =$ $F=D_1((D_0; P_1; D_2), D_0(D_3))$

$= min\{ d(D_1), d(D_0;P_1;D_2), d(D_0(D_3)) \} =$

$= min\{ d(D_1), min\{ d(D_0), d(P_1), d(D_2) \},$

$min\{ d(D0), d(D_3) \}\} =$

$= min\{ 1, min\{ 1,1,1 \}, min\{ 1,1 \}\} = min\{ 1,1,1 \} = \mathbf{1}$

➔ **F is D-structured** (F is built up of common primes, i.e. simple structures allowable in structured programming)

Cyclomatic Complexity

- A program's complexity can be measured by the cyclomatic number of the program flowgraph.
- The cyclomatic number can be calculated in 2 different ways:
  - Flowgraph-based
  - Code-based

Cyclomatic Complexity (2)

- For a program with the program flowgraph *F*, the cyclomatic complexity *v(F)* is measured as:

$$v(F) = e - n + 2$$

where

  - *e* is the number of edges (representing branches and cycles), and
  - *n* is the number of nodes (representing block of sequential code)

- The cyclomatic number actually measures the number of linearly independent paths through F (a set of paths is linearly independent if no path in the set is a linear combination of any other path)

Cyclomatic Complexity (3)

- For a program with the program flowgraph *F*, the cyclomatic complexity *v(F)* is measured as:

  $$v(F) = 1 + d$$

  where *d* is the number of predicate nodes (i.e., nodes with *out-degree* greater than 1), representing the number of decision points in the program
- The complexity of primes depends only on the predicates in them
- v(F) can be defined as a hierarchical measure

Cyclomatic Complexity: Questions

1. **What is the complexity of the primes?**

   The complexity of primes is the number of predicates plus one.

$$v(F) = 1 + d$$

2. **What is the complexity of sequencing?**

   Complexity of a sequence is equal to the sum of the complexities of the components minus the number of components plus one.

$$v(F_1;F_2;...;F_n) = \sum_{i=1}^{n} v(F_i) - n + 1$$

Cyclomatic Complexity: Questions (2)

3. **What is the complexity of nesting onto a given prime?**

   Complexity of nesting components on a prime F is equal to the complexity of F plus the sum of complexities of the components minus the number of components.

$$v\left(F(F_1;F_2;\ldots;F_n)\right) = v(F) + \sum_{i=1}^{n} v(F_i) - n$$

Example: Flowgraph-based

$v(F) = e - n + 2$

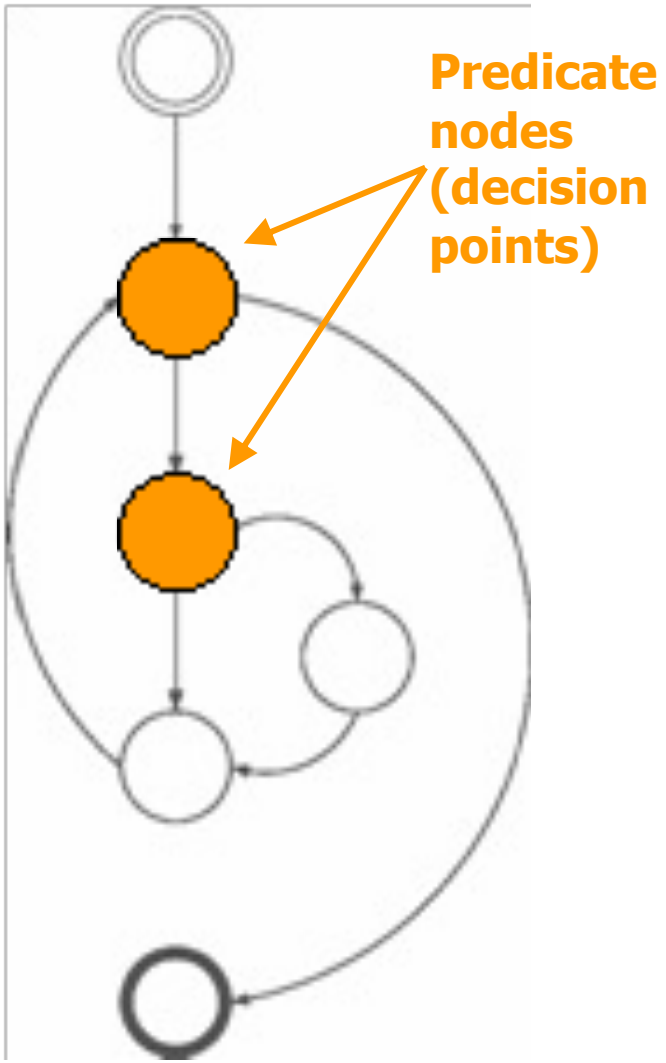
$v(F) = 7 - 6 + 2$

$v(F) = 3$

or

$v(F) = 1 + d$

$v(F) = 1 + 2 = 3$



Example: Code-based

```
#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);
}
```

$$v(F) = 1 + d = 1 + 2 = 3$$

McCabe's Essential Complexity

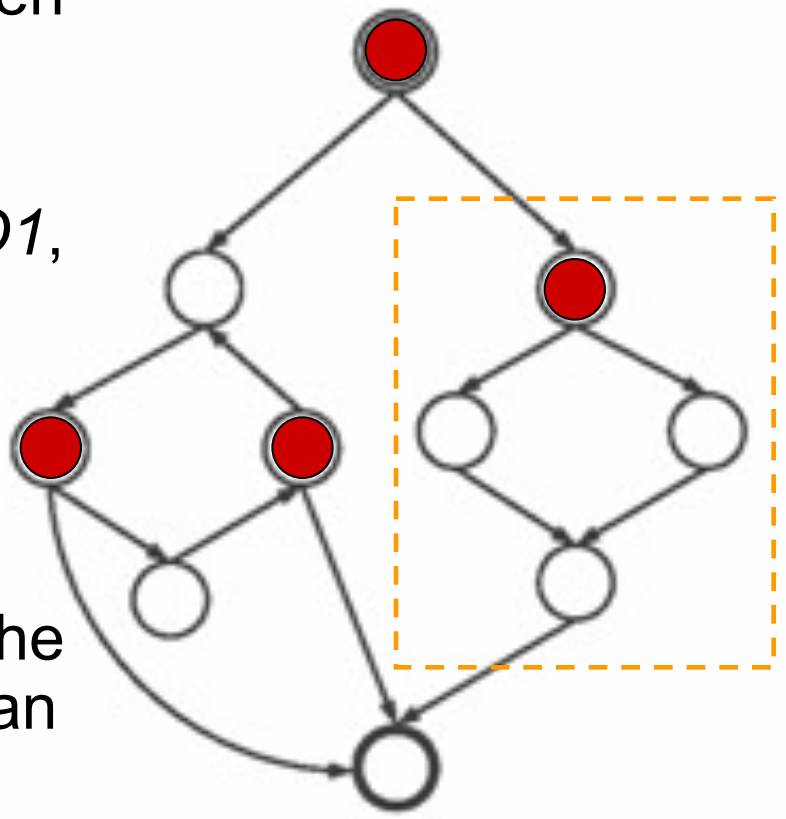
- **Essential complexity** of a program with flowgraph *F* is given by:

  $ev(F) = v(F) - m$

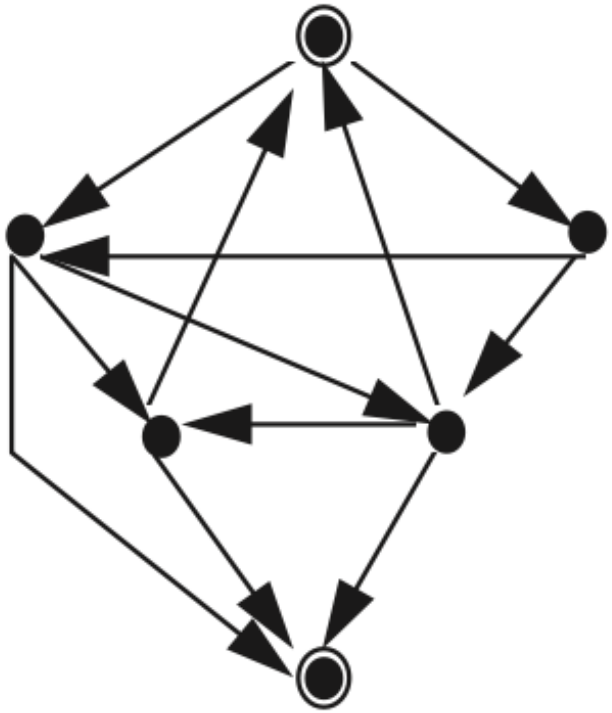
  where *m* is the number of *D0*, *D1*, *D2* and *D3* sub-flowgraphs of *F*
- Example:

  $v(F) = 5$

  $ev(F) = 5 - 4 = 1$
- Essential complexity indicates the extent to which the flowgraph can be reduced by decomposing all *D0*, *D1*, *D2* and *D3* sub-flowgraphs (*ev(F)* = 1 for a D-structured program with flowgraph *F*)



Example Unstructured Prime



Essential complexity = 6

Cyclomatic Complexity: Critics

- **Advantages:**
  - Objective measurement of complexity.
- **Disadvantages:**
  - Can only be used at the component level.
  - Two programs having the same cyclomatic complexity number may need different programming effort.
  - Requires complete design or code visibility