USING GRAPHS TO ANALYZE APL FUNCTIONS

Robert Metzger
Education Manager
I.P. Sharp Associates, Inc.
1200 First Federal Plaza
Rochester, N.Y. 14618

INTRODUCTION

Pictures and diagrams are used in a variety of professions to analyze problems and design solutions. Computing is no exception. Flow charts were developed early in the history of computing. They were the first program design tool to become widely used, and some people still use them today. Programs which generated flow charts from existing programs followed close behind. So, using pictures to help analyze programs is not a new idea.

Another diagram commonly used to design and analyze programs is the subroutine call tree. These diagrams show the hierarchical structure of a set of programs in a tree format. These, too, can be generated by a program. Just as flow charts show flow of control within a program, these charts show flow of control between programs.

Flow charts and subroutine call trees, as well as other diagrams representing programs, can be viewed as graphs in the sense of mathematical graph theory. Graphs can be represented by at least three kinds of matrices. Once we have an array representation of a graph, we can use *APL* to develop quantitative measures of the qualities of these programs.

The purpose of this paper is to show how graphs can be used to analyze *APL* functions. Several kinds of graphs are introduced. In each case, *APL* functions which generate matrix representations of these graphs derived from *APL* programs are presented. Then, *APL* functions which compute measures of the quality of a program based upon these graphs are suggested.

GRAPH THEORY

In graph theory, a graph consists of a set of points (vertices) and a set of lines (edges). In a directed graph (digraph), the lines extend in a specified direction. A graph may be represented by a square boolean matrix. In this connection (adjacency) matrix, if element $[I;J]$ is 1, then there is a line from point I to J. Unless the following two conditions are met, the graph is considered a multigraph.
1) Multiple lines between the same pair of points are not allowed.
2) Lines which begin and end at the same point are not allowed.
Therefore, the connection matrix is always boolean, and the main diagonal contains only zeros.

The number of points or nodes in the graph is $1\uparrow\rho GRAPH$. The number of lines in the graph is $+/,GRAPH$. The in-degree of a point is the number of lines ending at that point. The out-degree of a point is the number of lines starting at that point. The expression $+/GRAPH$ will give the out-degrees of the nodes and $+\not/GRAPH$ will yield the in-degrees.

If *GRAPH* identifies adjacent vertices, then the result of $GRAPH\vee.\wedge GRAPH$ identifies all paths of length one or two between the vertices of the graph. The generalization of this operation is the transitive closure. It identifies all paths of any length between two points. It can be computed with the following function.

```
CLOSURE◊ ∧/,ω=C←ω∨ω∨.∧ω ◊ CLOSURE C ◊ C
```

A graph is connected if there is a path between any two of its vertices. This can be tested wth $\wedge/\wedge/CLOSURE\ GRAPH$.

This function, like several others in this paper, is listed in the Direct Form, which was first introduced in [IVER76]. The notation has been modified, per Iverson's current usage, in two ways. The diamond (◇) is used instead of the colon (:) to separate the name of a function from its body. In addition, the order of the segments in a conditional Direct form is
NAME◇ IF ◇ ELSE(Ø) ◇ THEN(1).

A short exposition of graph theory in terms of *APL* can be found in "Notation as a Tool of Thought" [IVER8Ø]. More detailed explanations of graph theory concepts can be found in [CHRI75]. Applications of graph theory done in APL are presented in [BLAA8Ø]. An entire chapter of [BERZ75] is devoted to the uses of graphs in analyzing programs, and includes an annotated bibliography.

FLOW CHARTS

A flow chart is a picture which shows the possible paths of control flow a program can take. Texts enclosed by geometric figures are connected by lines with arrows which indicate direction.

There are a number of figures used, some of which have conflicting meanings. The five most common are Terminal, Connector, Input/Output, Process, Decision. The first two are necessary because most flow charts don't fit on a single piece of paper. The next two figures have the common characteristic that only one arrow can leave them. They represent work being done. The last figure represents those instructions which cause flow of control to go in one of several directions. They are the reason for having flow charts. The flow chart of a program which has no decision blocks is trivial.

If we remove the texts from a flow chart and make all the figures into points, it is easy to see that a flow chart is a graph. In fact, flow charts are directed graphs (digraphs), since the lines indicate connection in one direction only.

Since a flow chart is a graph, we can apply the metrics of graph theory to compute quantitative measures of the flow of control of a given program. A linear program, ie. one which contained no branches, will have 1's in the diagonal above the main diagonal. A linear program of four statements would be represented by the matrix below.

```
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
```

Conditional branch statements have out-degrees of 2 or more. A program segment and its graph are listed below.

```
[∘] CTR←first-1
[∘] LMT←last
[∘] LOOP:→(LMT<CTR←CTR+1)/END
[∘] work
[∘] →LOOP
[∘] END:
```

```
                        OUTDEGREES
0 1 0 0 0 0       1
0 0 1 0 0 0       1
0 0 0 1 0 1       2
0 0 0 0 1 0       1
0 0 1 0 0 0       1
0 0 0 0 0 0       0
```

Implementing the flowchart involves two steps. First , the control flow must be analyzed. Then, the diagram must be drawn. The function *FLOWCHART* calls two subfunctions to do this.

```
    ∇ FLOWCHART NAME;TEXT
[1]  TEXT←⎕CR NAME
[2] TEXT DRAWCHART 0 GRAPHFLOW 1 0 ↓TEXT ∇
```

(Programming Note: Names with alternating underscores are used in functions which execute ⎕*CR* in order to minimize name conflicts.)

The function *GRAPHFLOW* returns a connection matrix. It assumes that all branch targets in a function are labelled lines. As with each of the functions in this paper whose name begins with *GRAPH*, the left argument of *GRAPHFLOW* is a boolean scalar. If it is 1, the rows of the result are labelled, and the result is a character matrix. If it is Ø, no labels are provided, and the result is a numeric matrix. The right argument is the canonical representation of a function, with the header removed.

```
    ∇ GRAPH←CONTROL GRAPHFLOW TEXT
;CODE;SELECT;LABELS;LINES;BRANCHES
;TARGETS;CTR;LMT;LINE;MASK;NAMES;WIDTH
[1]  CODE←MARKCODE TEXT
[2]  SELECT←~CODE∧TEXT=':'
[3]  SELECT←, 0 1 ↓∧\(~∧/SELECT),SELECT
[4]  LABELS←(ρTEXT)ρSELECT\SELECT/,TEXT
[5]  LABELS←(LABELS∨.≠' ')⌿
     (' '∨.≠LABELS)/LABELS
[6]  LINES←(∨/(ρTEXT)ρSELECT)/⍳1↑ρTEXT
[7]  GRAPH←0, 0 ¯1 ↓(⍳1↑ρTEXT)∘.=⍳1↑ρTEXT
[8]  BRANCHES←(∨/CODE∧TEXT='→')/⍳1↑ρTEXT
[9]  GRAPH[BRANCHES;]←0
[10] CTR←0
[11] LMT←ρBRANCHES
[12]LOOP:→(LMT<CTR←CTR+1)ρEND
[13] LINE←TEXT[BRANCHES[CTR];]
[14] LINE←((LINE⍳':')×BRANCHES[CTR]∈
     LINES)↓LINE
[15] MASK←(-ρLINE)↑CODE[BRANCHES[CTR];]
[16] NAMES←MASK GETNAMES LINE
[17] WIDTH←(¯1↑ρNAMES)⌈¯1↑ρLABELS
[18] TARGETS←(∨/(((1↑ρLABELS),WIDTH)↑
     LABELS)∧.=⍉((1↑ρNAMES),WIDTH)
     ↑NAMES)/LINES
[19] TARGETS←TARGETS,(~∧/MASK∧LINE∈' →',
     ALFNUM)/1+BRANCHES[CTR]
[20] GRAPH[BRANCHES[CTR];TARGETS]←1
[21] →LOOP
[22]END:→(~1∈CONTROL)ρEXIT
[23] GRAPH←(⍒((1↑ρGRAPH),1)⍳1↑ρGRAPH),
     ' ',⍒GRAPH
[24]EXIT: ∇
```

(Programming Note: This function assumes that the targets of all branches are labels. It will not handle branches to numbers or ⎕*LC*.)

The function *GETNAMES* returns a matrix containing all the names used in a program text. The *MARKCODE* function produces a boolean vector marking those characters in a program text which are not literals or comments.

```
    ∇ NAMES←MASK GETNAMES TEXT
;SELECT;EXPAND
[1]  TEXT←TEXT,' '
[2]  MASK←,(ρTEXT)↑MASK
[3]  TEXT←,TEXT
[4]  SELECT←MASK∧TEXT∈
     ALPHABET,'⎕0123456789'
[5]  TEXT←SELECT\SELECT/TEXT
[6]  TEXT←((TEXT≠' ')∨TEXT≠¯1⌽TEXT)/TEXT
[7]  EXPAND←(TEXT=' ')/⍳ρTEXT
[8]  EXPAND←¯1+EXPAND-0,¯1↓EXPAND
[9]  EXPAND←(EXPAND∘.≥⍳⌈/EXPAND),1
[10] NAMES← 0 ¯1 ↓(ρEXPAND)ρ(,EXPAND)\TEXT
[11] NAMES←(NAMES[;1]∈ALPHABET)⌿NAMES ∇

    ∇ MASK←MARKCODE TEXT
[1]  MASK←TEXT=''''
[2]  MASK←MASK∨~≠\MASK
[3]  MASK←MASK∧~∨\(TEXT='⍝')∧MASK ∇
```

(Programming Note: The variable *ALPHABET* should contain all valid alphabetics. The variable *CR* should contain the Carriage Return, or New Line character.)

The function *DRAWCHART* has been omitted in the interest of brevity. These functions will work on any IBM-compatible APL. They would be more complicated if they had to handle such language enhancements as the diamond statement separator.

We can now see a way to compute a measure of the flow of control complexity of an *APL* program. We can compute the ratio of the number of branch edges to the number of statement vertices. The argument to this function is a boolean connection matrix.

```
FLOWCOMPLEXITY1◊ (+/0⌈¯1++/ω)÷1↑ρω
```

The result this program gives for a linear program is 0. A program which consisted of nothing but two-way conditional branches would be given a value of 1. The larger the result, the more dense and complicated the branching.

There is an alternatve way of computing flow complexity. The concept was originally explained in [MCCA78]. If a graph is connected, the number of independent paths in the graph is 1+ the number of edges + the number of nodes. Now program flow graphs are not connected. If we assume, however, that there is an edge from the last statement to the first, then they are connected. This means that the number of independent branch paths is (1+((+/+/*GRAPH*)-1↑ρ*GRAPH*)) +1 (for the imaginary path). This the rationale for the following function.

```
FLOWCOMPLEXITY2◊ 2+(+/+/ω)-1↑ρω
```

This measure of flow complexity is also very useful in generating test cases to test out a program.

FUNCTION CALL TREES

A function call tree represents the hierarchy of functions. It shows which function may call another during a particular execution. It does not show the sequence of the functions calls, nor the conditions under which they may be called. It consists of the names of the functions connected by lines which indicate subroutine calls.

We can use the same boolean connection matrix to represent the subroutine calls. The connection matrix is a square matrix in which there is a row and column for each function in the workspace. ⎕*LX* must be treated as a function for this representation to make sense. If element [*I*;*J*] is 1, then function *I* calls *J*. If we relax the requirement that the main diagonal be 0 (no loops), we can test for the presence of recursive functions with ∨/1 1⍉*GRAPH*. Strictly speaking, a graph

showing recursions is a multigraph. We can locate the main routine in the workspace with (0=+⌿*GRAPH*)⍳1.

For an *APL* workspace, we may have to represent the hierarchy with more than one tree. We will have several roots (and several trees) if there are functions which do not trace their activation back to ⎕*LX*. These functions will be activated by the user directly while in immediate execution mode. Paul Berry calls such workspaces 'open systems'. Thus the test for an open (versus closed) system is 1<+/0=+⌿*GRAPH*.

Implementing the function call tree also involves an analysis and drawing phase.

```
    ∇ FNCALLTREE;FNS
[1]  FNS←⎕NL 3
[2]  FNS←((¯1↑⍴FNS)↑'⎕LX'),[1] FNS
[3]  FNS DRAWTREE 0 GRAPHCALLS FNS ∇
```

GRAPHCALLS returns a boolean matrix representing the function call tree. The program which draws the trees has been omitted. The right argument is a character matrix containing functions names, one per row.

```
    ∇ GRAPH←CONTROL GRAPHCALLS FNS
;CTR;LMT;NAMES;WIDTH
[1]  GRAPH←(2⍴1↑⍴FNS)⍴0≠0
[2]  CTR←0
[3]  LMT←1↑⍴FNS
[4] LOOP:→(LMT<CTR←CTR+1)⍴END
[5]  NAMES←⍕⍉(,(2 3 ⍴'⎕CR⍕  ')
     [1+'⎕'=1↑FNS[CTR;]];]),' FNS[CTR;]'
[6]  NAMES←((⍳2=⍴⍴NAMES),0)↑NAMES
[7]  →(0∈⍴NAMES)⍴LOOP
[8]  NAMES←(MARKCODE NAMES) GETNAMES NAMES
[9]  WIDTH←(¯1↑⍴FNS)⌈¯1↑⍴NAMES
[10] GRAPH[CTR;]←∨/(((1↑⍴FNS),WIDTH)↑FNS)
     ∧.=⍉((1↑⍴NAMES),WIDTH)↑NAMES
[11] →LOOP
[12] END:→(~1∈CONTROL)⍴EXIT
[13] GRAPH←FNS,' ',⍤GRAPH
[14] EXIT: ∇
```

The function call graph leads us to a way of quantifying the modularity of an application. We can make the following generalizations about modular systems.
1) Modular systems normally have fewer statements per program than monolithic ones.
2) Modular systems normally have more references to a given program than monolithic ones.
Both of these phenomena occur when more than one function uses subfunctions which are common to several functions.

The degree of modularity of a set of functions can be calculated with the following function. The argument is a matrix namelist of functions to be analyzed. The larger the result of the *MODULARITY* function, the less modular the set of programs is.

```
MODULARITY◇ ((+/COUNTFNLINES ⍵)÷1↑⍴⍵)
     ÷1+(+/+⌿GRAPHCALLS ⍵)÷1↑⍴⍵

   ∇ LINES←COUNTFNLINES FNS;CTR;LMT
[1]  LINES←⍳0
[2]  CTR←0
[3]  LMT←1↑⍴FNS
[4] LOOP:→(LMT<CTR←CTR+1)/END
[5]  LINES←LINES,¯1+1↑⍴⎕CR FNS[CTR;]
[6]  →LOOP
[7] END: ∇
```

'Modularity' has been in the past a pious programmer's platitude. Everyone said it was good, but no one could or would define what it meant. Now that you have an objective definition of modularity, you can judge for yourself who just preaches it and who actually practices it.

VARIABLE DEPENDENCE

A variable dependence graph shows the relationships between variables in a program. The vertices represent the variables in the program. The edges represent references to the values of other vertices (variables) in an assignment.

If we analyze a single *APL* program, we cannot get a true variable dependence graph. This is because we cannot resolve the referent type of names which are not assigned. They can be global functions or variables. The syntax of the references in the function are not sufficient, in general, to determine the referent type of the object. We can, however, compute a name dependence graph which includes global functions and variables.

If we analyze the following function, we will get the name dependence graph listed below it. (The names have been supplied for reference).

```
     ∇Z←X JOIN Y;W
[1]  W←(¯1↑⍴X)⌈(¯1↑⍴Y)
[2]  Z←(((1↑⍴X),W)↑X),[1] ((1↑⍴Y),W)↑Y ∇

X    0 0 0 0
Y    0 0 0 0
W    1 1 0 0
Z    1 1 1 0
```

The names are listed in the order that they appear in the program. The arguments are considered to be assigned on line [0], so they appear first.

We can note several standard features of these graphs. Unless the arguments are re-assigned, their rows will be all zeros. The rows related to subfunctions will also

be zeros, as will any global variables which are not assigned. This graph may or may not be a multigraph, because the previous values of variables may be used to re-assign them.

The functions listed below implement the variable dependence graph. The right argument to *GRAPHASSIGN* is a character vector which contains the name of the function to be analyzed.

```
    ∇ GRAPH←CONTROL GRAPHASSIGN NAME
;TEXT;ALFNUM;NUMBERS;SELECT;ORDER;NAMES
[1]  TEXT←0 MARKHEADER ⎕CR NAME
[2]  TEXT←(,(MARKCODE TEXT),1)/,TEXT,CR
[3]  ALFNUM←TEXT∈ALPHABET,'0123456789⎕⍙'
[4]  TEXT←ALFNUM MARKINDEXASSIGN TEXT
[5]  SELECT←ALFNUM COMPRESSTEXT TEXT
[6]  TEXT←SELECT/TEXT
[7]  ALFNUM←SELECT/ALFNUM
[8]  NUMBERS←ALFNUM REFNUMBERS TEXT=CR
[9]  SELECT←ALFNUM∨¯1↓0,ALFNUM
[10] NAMES←(~SELECT/ALFNUM) MATFROMVEC
     SELECT/TEXT
[11] ORDER←(' ',ALPHABET,'0123456789')
     CHARMATGRADE 0 ¯1 ↓NAMES
[12] GRAPH←NUMBERS[ORDER] BUILDASSIGNGRAPH
     NAMES[ORDER;] ∇
```

The function *MARKHEADER* is used here is extract the names of the arguments. It also marks them as assigned on line [0].

```
    ∇ FUNCTION←CONTROL MARKHEADER TEXT
;HEADER;RESULT;PARMS;TEMP;LOCALS;⎕IO
[1]  ⎕IO←1
[2]  HEADER←TEXT[1;]
[3]  HEADER←(φ∨\φ' '≠HEADER)/HEADER
[4]  LOCALS←(¯1+HEADER⍳';')↓HEADER
[5]  HEADER←(-⍴LOCALS)↓HEADER
[6]  RESULT←(('←'∈HEADER)×HEADER⍳'←')↑
     HEADER
[7]  HEADER←(⍴RESULT)↓HEADER
[8]  RESULT←(¯1↓RESULT),(0≠⍴RESULT)
     /¯1↑¯1↓⎕AV
[9]  PARMS←''
[10] →(2 1 0 =+/HEADER=' ')/LEFT,RIGHT,END
[11] LEFT:TEMP←HEADER⍳' '
[12] PARMS←((¯1+TEMP)↑HEADER),¯1↑¯2↓⎕AV
[13] HEADER←TEMP↓HEADER
[14] RIGHT:PARMS←PARMS,((HEADER⍳' ')
     ↓HEADER),¯1↑¯2↓⎕AV
[15] END:HEADER←(CONTROL/RESULT),PARMS
     ,(CONTROL/LOCALS)
[16] FUNCTION←((¯1↑⍴TEXT)↑HEADER)
     ,[⎕IO] 1 0 ↓TEXT ∇
```

The function *MARKINDEXASSIGN* flags those index references (*A*[...]) which are actually indexed assignments.

```
    ∇ FUNCTION←ALFNUM MARKINDEXASSIGN TEXT
;LOCATIONS;BRACKETS;POSITIONS;SELECT
[1]  LOCATIONS←TEXT∈'[]'
[2]  →(~∨/LOCATIONS)⍴END
[3]  BRACKETS←LOCATIONS/TEXT
[4]  POSITIONS←(BRACKETS=']')++\
     1 ¯1['[]'⍳BRACKETS]
[5]  POSITIONS←(LOCATIONS/⍳⍴LOCATIONS)
     [⍋POSITIONS]
[6]  POSITIONS←((0.5×⍴POSITIONS),2)
     ⍴POSITIONS
[7]  SELECT←(¯1⌽ALFNUM)∧~ALFNUM
[8]  POSITIONS←(POSITIONS[;1]∈SELECT/
     ⍳⍴SELECT)⌿POSITIONS
[9]  TEXT[(TEXT[POSITIONS[;2]+1]='←')
     /POSITIONS[;1]]←¯1↑⎕AV
[10] END:FUNCTION←TEXT ∇
```

The next four functions do most of the work to prepare a traditional cross reference listing. *COMPRESSTEXT* selects the names in the text and the symbols immediately to their right. *REFNUMBERS* computes the line numbers that the names are located on. *MATFROMVEC* creates a matrix of the names in the text. *CHARMATGRADE* grades the matrix namelist. On SHARP APL, it can be replaced with a call to dyadic grade up.

```
    ∇ SELECT←ALFNUM COMPRESSTEXT TEXT
;PARTN;LINES;TEMP
[1]  LINES←TEXT=CR
[2]  SELECT←LINES∨ALFNUM∨¯1↓0,ALFNUM
[3]  TEXT←SELECT/TEXT
[4]  LINES←SELECT/LINES
[5]  ALFNUM←SELECT/ALFNUM
[6]  PARTN←LINES∨ALFNUM∧~¯1↓0,ALFNUM
[7]  TEMP←PARTN/TEXT∈'0123456789⍙⎕'
[8]  SELECT←SELECT\~≠\PARTN\TEMP≠¯1↓0,TEMP
∇

    ∇ NUMBERS←ALFNUM REFNUMBERS LINES
;PARTN;POSITIONS
[1]  PARTN←ALFNUM∧~¯1↓0,ALFNUM
[2]  POSITIONS←(PARTN∨LINES)/LINES≤PARTN
[3]  POSITIONS←POSITIONS/⍳⍴POSITIONS
[4]  NUMBERS←(POSITIONS-⍳⍴POSITIONS)
     ++\PARTN/LINES ∇

    ∇ MATRIX←DELIMITERS MATFROMVEC VECTOR
;EXPAND;LENGTH;ENDS
[1]  ENDS←DELIMITERS/⍳⍴DELIMITERS
[2]  LENGTH←¯1↓ENDS-0,¯1↓ENDS
[3]  EXPAND←(LENGTH∘.≥⍳⌈/LENGTH),1
[4]  MATRIX←(⍴EXPAND)⍴(,EXPAND)\VECTOR ∇

    ∇ ROWS←KEY CHARMATGRADE MATRIX
;BASE;COLUMNS
[1]  ROWS←⍳1↑⍴MATRIX
[2]  BASE←1+⍴KEY
[3]  COLUMNS←1↓⍴MATRIX
[4]  COLUMNS←(-COLUMNS⌊⌊BASE⊛2147483647)
     ↑⍳COLUMNS
[5]  LOOP:→(0=⍴COLUMNS)⍴END
[6]  ROWS←ROWS[⍋BASE⊥⍉KEY⍳MATRIX
     [ROWS;COLUMNS]]
[7]  COLUMNS←COLUMNS-⍴COLUMNS
[8]  COLUMNS←(COLUMNS≥1)/COLUMNS
[9]  →LOOP
[10] END: ∇
```

BUILDASSIGNGRAPH takes the cross reference information and turns it into a boolean matrix representing the graph. It uses partitioning techniques to handle the non-rectangular data.

```
    ∇ GRAPH←NUMBERS BUILDASSIGNGRAPH NAMES
;USAGE;PARTN;MAX;SELECT;ORDER;CTR;LMT;SET;
SEGMENT
[1]  USAGE←('←:[↓↑',(¯1↑⎕AV),' ')
     [('←:[',¯3↑⎕AV)⍳,NAMES[;¯1↑⍴NAMES]]
[2]  NAMES← 0 ¯1 ↓NAMES
[3]  PARTN←1,1↓∨/NAMES≠¯1⊖NAMES
[4]  NAMES←PARTN⌿NAMES
[5]  SELECT←~PARTN PARTNORREDUCE ':'=USAGE
[6]  NAMES←SELECT⌿NAMES
[7]  SELECT←≠\PARTN\SELECT≠¯1↓0,SELECT
[8]  NUMBERS←SELECT/NUMBERS
[9]  PARTN←SELECT/PARTN
[10] USAGE←SELECT/USAGE
[11] ORDER←⍋PARTN/NUMBERS
[12] NAMES←NAMES[ORDER;]
[13] ORDER←⍋(⍋ORDER)[+\PARTN]
[14] PARTN←PARTN[ORDER]
[15] NUMBERS←NUMBERS[ORDER]
[16] USAGE←USAGE[ORDER]
[17] SET←(NUMBERS≠0)∧~USAGE∈'[ '
[18] CTR←0
[19] LMT←+/PARTN
[20] GRAPH←(0,LMT)⍴0
[21] SEGMENT←LMT↑1
[22] LOOP:→(LMT<CTR←CTR+1)⍴END
[23] SELECT←SET∧≠\PARTN\SEGMENT≠
     ¯1↓0,SEGMENT
[24] GRAPH←GRAPH,[1] PARTN PARTNORREDUCE
     (NUMBERS×(~SELECT)+¯1×SELECT)
     ∈SELECT/NUMBERS
[25] SEGMENT←¯1⌽SEGMENT
[26] →LOOP
[27] END:→(~1∈CONTROL)⍴EXIT
[28] GRAPH←NAMES,' ',▼GRAPH
[29] EXIT: ∇
```

```
PARTNORREDUCE◇ (C/1⌽C←(α∨ω)⌿α)≤α/ω
```

(Programming Note: This function assumes that the flow of control in the program it is analyzing is top-down. This means that the line number of every branch destination is greater than or equal to the line number of the origin, except for branches returning to the head of a loop.)

We have an alternative function which uses the SHARP APL implementation of enclosed arrays. The ratio of statements between the two is almost 3 to 1. This would seem to substantiate the claims that have been made about productivity improvements from enclosed arrays. When a data structure is non-rectangular, the difference between an implementation which has enclosed arrays and one which doesn't may be as significant as the difference between *APL* and FORTRAN.

The reduction in coding is due to replacing partitioning operations with uses of the new operators. Lines [5 6 7 8 9 10] and a subfunction call are replaced by line [5]. Lines [11 12 13 14 15 16] are replaced by line [6]. Lines [17 18 19 20 21 22 23 24 25 26], which include a loop and a subfunction call, are replaced by lines [7 8].

```
  ∇ GRAPH←NUMBERS BUILDASSIGNGRAPH2 NAMES
;USAGE;PARTN;MAX;TABLE;SELECT;⎕PS
[1]  USAGE←,('←:[↓↑',(¯1↑⎕AV),' ')
     ⊥('←:[',¯3↑⎕AV)⍳,NAMES[;¯1↑⍴NAMES]]
[2]  NAMES← 0 ¯1 ↓NAMES
[3]  PARTN←1,1↓∨/NAMES≠¯1⊖NAMES
[4]  TABLE←(<⍤1 PARTN⌿NAMES),
   (PARTN 2⍤< NUMBERS),[1.5]
     PARTN 2⍤< USAGE
[5]  TABLE←(~':'∈⍤>TABLE[;3])⌿TABLE
[6]  TABLE←TABLE[⍋1↑⍤>TABLE[;2];]
[7]  SELECT←(~⍨⍤>TABLE[;3]∈⍨⍤><
     '[ ')∧⍨⍤>0≠⍨⍤>TABLE[;2]
[8]  GRAPH←⍉∨/⍤>((~⍨⍤>SELECT)/⍨⍤>TABLE[;2])
     ∘.∈⍨⍤>SELECT/⍨⍤>TABLE[;2]
[9]  →(~1∈CONTROL)⍴EXIT
[10] ⎕PS← ¯1 ¯1 0 1 ◇ GRAPH←(▼TABLE[;,1]),
     ' ',▼GRAPH
[11] EXIT: ∇
```

Space limitations prevent me from discussing the details of how this function works. Those interested in this implementation may read more about it in [BERN8Ø]. Topics demonstrated here include:

1) ON (⍤) as an axis operator (<⍤1),
2) ON (⍤) as a partitioning operator (2⍤<),
3) ON (⍤) as a composition operator (∈⍤> and ↑⍤>),
4) WITH (⍨) as the dual operator (≠⍨> ∧⍨> ∈⍨> ~⍨>),
5) Derived functions as arguments to operators (*S*/⍨> ∘.∈⍨> ∨/⍨>),
6) ⎕*PS* for controlling the formatting of enclosed arrays.

What is the significance of the name dependence graph? To learn this, we must recall a concept from graph theory. The transitive closure of a graph identifies all paths of any length between two points. If (*CLOSURE GRAPH*)[*I*;*J*] is 1, there is a path of some length from *I* to *J*.

In the case of the name dependence graph, its closure will indicate any dependence of one name on another. This dependence may be direct or indirect. If *I* depends on *J*, *J* depends on *K*, and *K* depends on *L*, then (*CLOSURE GRAPH*)[*I*;*L*] will be 1.

This result may be useful in program maintenance. You can quickly check all the variables which will be effected if you change the definition of some object. If you just look down the column corresponding to that variable, it will mark all variables that depend on that variable.

The closure of this graph may also be useful in debugging. If you just look

across the row corresponding to that variable, it will mark all the names that it depends on. If the value of a variable is wrong, you can easily identify all the objects which might be the source of the problem.

Perhaps the most interesting use of this graph is in the area of software metrics. It can be used to measure an important software quality- module cohesion. This concept is an essential part of the discipline of Structured Design. It is defined by Yourdon and Constantine [YOUR79] as "the degree of functional relatedness of processing elements within a single module." Myers [MYER78] defines his equivalent term ('module strength') as "a measurement of the relationships among the elements within a single module."

Both of these books propose a hierarchy of the degrees of cohesion.

Yourdon	Myers
coincidental	coincidental
logical	logical
temporal	classical
procedural	procedural
communicational	communicational
sequential	
functional	functional
	informational

They describe the distinctives of each these degrees. Unfortunately, while they speak of 'degree' and 'measurement' their descriptions are very qualitative.

The most desirable degree of cohesion is functional. The definitions are a module in which "every element of processing is an integral part of, and essential to, the performance of a single function" [YOUR79] and "a functional strength module is defined as a module that performs a single specific function." [MYER78]

A broad class of functionally cohesive programs are those which compute an explicit result based entirely upon one or two arguments. Such functions can be easily identified with the closure of the name reference graph. If the only rows which are all zeros are those corresponding to the arguments, and if the row corresponding result is all 1's, except for the column corresponding to itself, then the program is functionally cohesive. This means that the result depends on every variable which is calculated, including the arguments. Every calculation is done for the ultimate goal of calculating the result.

This definition can be broadened. If only arguments and results are used for inter-function communication, we can relax the restriction on all-zero rows. They may also correspond to subfunctions. Either of these definitions is sufficient, but not necessary. In other words, there may be programs which don't fit these criteria which are functionally cohesive. Even this partial definition, however, is much more satisfactory than the 'warm fuzzies' which Structured Design advocates have been content with in the past.

FUNCTION CONNECTION

A function connection graph shows the data relationships between functions. The vertices represent the functions being analyzed. The edges represent references to variables belonging to another function.

If we analyze the following functions, we will get the function connection matrix listed below them. (The names have been supplied for reference).

```
     ∇R←X FOO1 Y
[1]  R←X+FOO2 Y ∇
     ∇R←Y FOO2 Z
[1]  R←Z-FOO3 Y ∇
     ∇R←A FOO3 W
[1]  R←X+A+W×Y ∇

FOO1   0 0 0
FOO2   0 0 0
FOO3   1 1 0
```

In this case, *FOO3* uses a variable belonging to *FOO1*, and also one belonging to *FOO2*.

The functions listed below implement the function connection graph. The right argument to *GRAPHCONNECT* is a character matrix. It should contain the names of the functions to be analyzed, one name per row.

```
     ∇ GRAPH←CONTROL GRAPHCONNECT FNS
;CTR;LMT;XREFTAB;TEMP;WIDTH
[1]  XREFTAB← 0 3 ⍴''
[2]  CTR←0
[3]  LMT←1↑⍴FNS
[4] LOOP:→(LMT<CTR←CTR+1)⍴END
[5]  TEMP←⎕AV[CTR],XREFNL ⎕CR FNS[CTR;]
[6]  WIDTH←(¯1↑⍴XREFTAB)⌈¯1↑⍴TEMP
[7]  XREFTAB←(((1↑⍴XREFTAB),WIDTH)↑
     XREFTAB),[1]((1↑⍴TEMP),WIDTH)↑TEMP
[8]  →LOOP
[9] END:GRAPH←BUILDCONNECTGRAPH XREFTAB
[10] →(~1∈CONTROL)⍴EXIT
[11] GRAPH←FNS,' ',⍕GRAPH
[12]EXIT: ∇
```

The function *XREFNL* produces a cross reference name list for a single function. Its argument is a canonical representation of an *APL* function. It presumes the availability of the subfunctions used by *GRAPHASSIGN*.

```
    ∇ TABLE←XREFNL TEXT;ALFNUM
;SELECT;NAMES;ORDER;NUMBERS;PARTN;USAGE
[1]  TEXT←1 MARKHEADER TEXT
[2]  TEXT←(,(MARKCODE TEXT),1)/,TEXT,CR
[3]  ALFNUM←TEXT∈ALPHABET,'0123456789⎕⍞'
[4]  SELECT←ALFNUM COMPRESSTEXT TEXT
[5]  TEXT←SELECT/TEXT
[6]  ALFNUM←SELECT/ALFNUM
[7]  NUMBERS←ALFNUM REFNUMBERS TEXT=CR
[8]  SELECT←ALFNUM∨¯1↓0,ALFNUM
[9]  NAMES←(~SELECT/ALFNUM) MATFROMVEC
     SELECT/TEXT
[10] ORDER←(' ',ALPHABET,'0123456789')
     CHARMATGRADE 0 ¯1 ↓NAMES
[11] USAGE←,NAMES[ORDER;¯1↑⍴NAMES]
[12] NAMES← 0 ¯1 ↓NAMES[ORDER;]
[13] NUMBERS←NUMBERS[ORDER]
[14] PARTN←1,1↓∨/NAMES≠¯1⊖NAMES
[15] TABLE←⎕AV[1+PARTN PARTNORREDUCE
     0=NUMBERS],PARTN⌿NAMES
[16] TABLE←(~PARTN PARTNORREDUCE
     ':'=USAGE)⌿TABLE ∇
```

The function *BUILDCONNECTGRAPH* turns the accumulated cross reference lists from all the functions into the boolean matrix representing the connection graph.

```
    ∇ GRAPH←BUILDCONNECTGRAPH XREF
;FUNCTIONS;PARTN;LOCAL;VARS;USAGE
[1]  FUNCTIONS←⎕AV⍳XREF[;1]
[2]  PARTN←1,1↓FUNCTIONS≠¯1⊖FUNCTIONS
[3]  LOCAL←2=⎕AV⍳XREF[;2]
[4]  VARS← 0 2 ↓XREF
[5]  USAGE←VARS MATRIXINDEXOF VARS
[6]  CTR←0
[7]  LMT←+/PARTN
[8]  GRAPH←(0,LMT)⍴0
[9]  SEGMENT←LMT↑1
[10]LOOP:→(LMT<CTR←CTR+1)⍴END
[11] SELECT←(~LOCAL)∧≠\PARTN\
     SEGMENT≠¯1↓0,SEGMENT
[12] GRAPH←GRAPH,[1] PARTN PARTNORREDUCE
     USAGE∈SELECT/USAGE
[13] SEGMENT←¯1⌽SEGMENT
[14] →LOOP
[15]END:GRAPH←GRAPH∧(⍳1↑⍴GRAPH)∘.≠
     ⍳1↑⍴GRAPH ∇

MATRIXINDEXOF◇ ⎕IO←+/∧\⍵∨.≠⍳⍺
```

Once again, we have an alternative version which uses the SHARP APL enclosed array implementation to handle the ragged data. No new enclosed array concepts are introduced here.

```
    ∇ GRAPH←BUILDCONNECTGRAPH2 XREF
;FUNCTIONS;PARTN;LOCAL;VARS;USAGE
[1]  FUNCTIONS←⎕AV⍳XREF[;1]
[2]  PARTN←1,1↓FUNCTIONS≠¯1⊖FUNCTIONS
[3]  LOCAL←2=⎕AV⍳XREF[;2]
[4]  VARS← 0 2 ↓XREF
[5]  USAGE←PARTN 2⍤< VARS MATRIXINDEXOF
     VARS
[6]  GRAPH←⍳∨/⍤>USAGE∘.∈
     ⍤>(PARTN 2⍤< ~LOCAL)/⍤>USAGE
[7]  GRAPH←GRAPH∧(⍳1↑⍴GRAPH)∘.≠⍳1↑⍴GRAPH ∇
```

What is the significance of the function connection graph? Like the previous graph, it can be used to measure an important software quality-- module coupling. Yourdon and Constantine [YOUR79] define this as "the measure of the strength of interconnection between one module and another." One of the chief goals of Structured Design is to produce modular software. Thus, it is desirable to maximize module cohesion and minimize module coupling.

Once again, Yourdon/Constantine and Myers [MYER78] present similar, but not identical, definitions of the degrees of coupling.

Yourdon	Myers
	no direct coupling
hybrid coupling	
content coupling	content coupling
common data coupling	common coupling
	external coupling
control coupling	control coupling
	stamp coupling
data coupling	data coupling

Some of these categories are not relevant in the *APL* environment. Besides the obvious 'no coupling', the other two that definitely are relevant in an *APL* environment are 'common coupling' and 'data coupling'. The first occurs when data is transmitted between functions through the use of variables which are global to one or both of them. The second occurs when data is transmitted between functions exclusively through arguments and results.

Using the connection graph to determine coupling is rather easy. Let

```
G←0 GRAPHCALLS NL
H←0 GRAPHCONNECT NL
```

We find the following results. `G∧~H` identifies the connections which are data coupled. `K←H∧G∨⍉G` identifies the connections between functions which are common coupled, and where one function is called by another. `K∧⍉K` identifies the connections between functions which are common coupled, and where neither function calls the other. If `0=+/,K` then the programs have no direct coupling.

CONCLUSION

This paper has presented 4 graphs which may be used to analyze *APL* programs. These are the flow of control graph, the function call graph, the variable dependence graph, and the function connection graph.

In each case, the power of *APL* made it relatively easy to write functions to produce matrix representations of these

graphs directly from *APL* programs. When the data being processed was essentially non-rectangular, the SHARP APL implementation of enclosed arrays proved to be a particularly concise and powerful notation. The power of *APL* was used again to define quantitative measures of important software qualities, based upon these graphs. In these examples, the uniqueness of *APL* can be seen in the fact that *APL* programs were both the objects and means of analysis.

REFERENCES

BERN80 BERNECKY, R. and IVERSON, K. E. "Operators and Enclosed Arrays," Proceedings of the 1980 APL Users Meeting, I.P. Sharp Associates, Ltd., Toronto, 1980.

BLAA80 BLAAUW, G.A. and DUIJVESTIN, A.J.W. "The Use of the Inner Product Operator of APL in Graph Applications, APL 80, North Holland Publishing Co., Amsterdam, 1980.

BERZ75 BERZTISS, A. T. Data Structures-- Theory and Practice, Academic Press, New York, 1975.

CHRI78 CHRISTOFIDES, N. Graph Theory-- An Algorithmic Approach, Academic Press, New York, 1975.

IVER80 IVERSON, K. E. "Notation as a Tool of Thought," Communications of the ACM 23, (1980), 444-465.

MCCA78 MCCABE, T.J. "A Complexity Measure, "IEEE Transactions on Software Engineering, Vol SE-2, Dec. 1976, pp. 308-320.

MYER78 MYERS, G. J. Composite/Structured Design, Van Nostrand Reinhold Co., New York, 1978.

YOUR79 YOURDON, E. and CONSTANTINE, L. Structured Design, Prentice-Hall, Englewood Cliffs, NJ, 1979.