INTERPROCEDURAL ANALYSIS AND THE INFORMATION DERIVED BY IT

F. E. Allen

Computer Sciences Department
IBM T. J. Watson Research Center
Yorktown Heights, New York 10598
USA

ABSTRACT

Well structured programs are usually expressed as a system of functionally oriented procedures. By analyzing and transforming an entire system of procedures, linkages can be modified or eliminated and interprocedural data dependencies documented to the user. This paper presents some of the methods being developed to effect such interprocedural analysis and transformations.

1. INTRODUCTION

As part of the effort to improve programmer productivity and system reliability, a number of excellent guidelines have emerged for the programmer: "write in a high level language", "avoid goto's and external variables" ("use the parameter passing mechanism instead"), "write small, functionally oriented routines", "annotate .and document the programs carefully", etc. Furthermore a number of languages and language constructs have been developed to support (and enforce) some of these techniques. While these and other developments in programming methodology have greatly increased the potential for improved programmer productivity and system reliability, there are some major problem areas requiring attention. In this paper we consider one aspect of the problem of developing, managing and maintaining the entire collection of procedures which will typically exist in a large system, particularly one which has been developed in a top-down style using many small, functionally oriented routines.

The context in which we will be considering this problem is that of a compiling system. We will be concerned with collections of procedures (and functions) written in a high level language. Both nested and external procedures are considered. Since compilers traditionally compile only one external procedure at a time, a quite radical departure from the traditional design is required; indeed the compiler should be viewed as one component of an entire system which interfaces with the user and manages his programs. The design of such a compiling system will not be further discussed in this paper. However, most of the ideas presented here have been, or are being implemented in an Experimental Compiling System (ECS) currently under development. Since this system is PL/I oriented, the methodology being developed is designed to accommodate the many features supported by that language and hence should be applicable to a number of other languages.

In this paper a method (actually a composite of methods) is presented which analyzes the collection of procedures which constitute all or part of a program. The analysis determines the possible control flow and data flow within each procedure and between the procedures. The next section, Section 2, lists some of this information and references the Appendix, which contains a program analyzed by ECS. Section 3 gives the algorithm used to develop the information. A brief discusssion of possible uses of the information in developing, managing and transforming programs is given in Section 4. We conclude with a summary, acknowledgements and a bibliography.

2. INFORMATION DERIVED

As a result of performing the analysis to be outlined in the next section, a great deal of information is obtained about the possible data and control relationships in the program. Some of the information which is produced is:

a. the call graph showing the possible invocation relationships in the collection
b. a control flow graph for each procedure
c. the data flow within each procedure
d. the control flow between procedures
e. the data flow between procedures.

The example given in the appendix shows some of the information currently being produced by the Experimental Compiling System. The example has been chosen to illustrate the type of information available rather than the PL/I features supported by ECS. Using the example, we now give a more detailed discussion of the information collected by interprocedural analysis.

2.1 The Call Graph

Given a collection of procedures, $(p_1, p_2, \ldots p_n)$, the referencing relationships between the procedures can be expressed by a directed graph $C = (N,E)$ of nodes $n_i \in N$ and edges $\ell_i \in E$ in which

a. each node, n_i, represents a procedure, p_i, and

b. each edge $(n_j, n_k) = \ell_i \in E$, represents one or more references in procedure p_i to procedure p_j.

Such a graph C is termed a call graph.

Although methods [7,8] are currently being developed for analyzing programs which contain recursive procedures, in this paper we will restrict our attention to non-recursive procedures. The call graph in Figure 1 depicts the collection of procedures A, B, C, D, E. The main procedure, A, contains references to procedures B and C; procedure, B, references C, D, and E; and procedure, D, references C and E.

It should be noted that the call graph is not a control flow graph since returns are not shown. Figure A2 in the appendix shows the call graph produced by ECS for the partial program given there.

2.2 The Control Flow Graph

For each procedure the flow relationships are depicted by a "control flow graph". A control flow graph is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths. A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed).

Figure A5 shows the control flow graph for the outer procedure EXAMPLE, in the appendix. The blocks are arbitrarily numbered and each block in the printout shows its number and the serial numbers of the source statements in the block. Block 1 is a dummy block and block 2 contains everything up through the IF test. Block 3 contains the first call to SUB and block 4 contains the branch around the ELSE clause which will be executed on return from the first call. Block 5 (for statement 8) has the second call and block 6 has the return statement.

2.3 The Data Flow Within Each Procedure

For each procedure two types of data flow information are obtained: "definition-use" relationships and "live" information.

Using the notation X^d_i to denote the definition of data item X in block b_i and X^u_j to denote the use of X in block b_j, then a definition-use relationship (or simply, a def-use relation) exists between them if the value created by the definition at b_i can be the one used at b_j. With this notation, (introduced in [3]), a def-use relation is expressed by the pair(X^d_i, X^u_j). Such a relationship can exist only if there is a path from b_i to b_j which does not contain a redefinition of the data item. Consider the example in Figure 2. The def-use pairs are (A^d_1, A^u_2) and (A^d_3, A^u_2).
It should be noted that the term "data item" rather than variable was used in defining the relationship. The same data item can have several aliases which must all be reconciled if the information is to be useful. These aliases can result from parameter-argument associations, the use of pointers or simply by overlaying storage.

In Figure A6 in the appendix the def-use relationships for the outer procedure, EXAMPLE, are shown. Here we see some of the effects of interprocedural analysis on local def-use information. Variables A, B, and C are all passed to SUB; A and B are used in SUB and C (via parameter Z) is modified. The def-use information in Figure A6 reflects this. A, for example, is shown as being defined at statement 3 in basic block 2 and used in the two basic blocks, 3 and 5, which contain the calls to SUB. On the other hand C is shown as being defined in statements 5, 7, and 8 but not used. (C is, in fact, "dead" and interprocedural optimization might eliminate it.)

The second form of data flow information is the live informatiom. Given a def-use relation (X_i^d, X_j^u) then X_i^d is <u>live</u> on all edges of any path from b_i to b_j which does not contain a redefinition of X. X_3^d is live on edges 3 to 4, and 4 to 2 in Figure 2.

2.4 The Control Flow Between Procedures

Not only are the usual calling relationships in a system of procedures exposed but non-nested control transfers (abnormal returns) are also found.

2.5 The Data Flow Between Procedures

When one procedure references another, certain data items are mutually accessible. These are data items which are passed as arguments, are defined as global variables, have the same scope, or are indirectly accessible through pointers, overlays, etc. At each call point the data items which are referenced and/or modified as a result of the call are identified.

The Experimental Compiling System has a listing annotator which automatically inserts comments into the source listing at certain points. These comments contain some of the interprocedural flow information. Figure A8 shows the partial result of such an annotation at the call point.

3. ANALYSIS METHOD

Given a collection of procedures, $p_1, p_2, \ldots p_n$, which constitute all or part of a program P, the problem which we want to consider in this section is how to derive the information listed in Section 2. We will draw heavily on material in the literature, particularly on the paper, Interprocedural Data Flow Analysis [1]. In order not to complicate the presentation, we will initially assume that the collection is complete, i.e., all of the procedures referenced are in the collection. It will later be evident that this requirement can be relaxed but will result in less accurate (but not incorrect) information being produced.

Before giving the analysis approach, a basic question needs to be resolved: in what order should the procedures be analyzed? The dilemma posed by this question can be illustrated by the procedures in Figure 3.

If S is analyzed first we cannot determine what is defined and used by the CALL statement: G, A, and B may each be defined and/or used. We cannot, therefore, accurately deduce the data flow of S.

If T is analyzed first we don't know whether or not X, Y and G are aliased in any way: X and Y might refer to the same actual argument which also might or might not be G. Hence the definition of X may also be defining Y and/or G. Again our data flow information might be inaccurate.

T could be analyzed in its reference context in S. However, if there are many references to T this could be very costly.

In [1] this dilemma is resolved by choosing the "inverse invocation order". In that paper it was assumed that a "worst case" estimate was always made regarding certain interferences such as between X, Y, and G in Figure 3. In this paper the notion of an initial estimate [9] is introduced which, if the estimate is based on an actual examination of the program, is more accurate than a "worst case" estimate. This approach is the one actually used in the Experimental Compiling System. The basic algorithm is now given.

Algorithm for Interprocedural Analysis

Step 1. Establish an initial estimate (actually an overestimate) on the control and data relationships in P.

Step 2. Establish an order for processing the procedures based upon the invocation relationships deduced as part of step 1 or, if the process is iterated, the more refined invocation relationships which can be determined from the information collected in Step 3.

Step 3. Establish the control and data relationships in P by processing the procedures in the order determined in Step 2 by using either the estimate on the control and data flow relationships or the relationships already deduced for procedures appearing earlier in the processing order.

Step 4. If desired, update the estimates with the information collected in step 3 and repeat steps 2, 3, and 4.

A reason for the iterative refinement of the information may be illustrated by considering the following example. Suppose a procedure, S, contains a CALL EV where EV is an entry variable. By the initial estimate we may determine that EV can take on a number of procedure values say P1, P2, and P3. However, having performed steps 2 and 3 on that assumption we may be able to deduce that EV can, in fact, only have the value P2, say, at that point in S. Redoing steps 2 and 3 with this new information leads to much more accurate information.

The steps in the process will now be elaborated.

3.1 (Step 1) Establish an initial estimate on the relationships in P. Three types of information are determined by the analysis performed in this step:

a. the possible values of all pointer, label and entry variables in P

b. the aliasing relationships including parameter-argument associations. In this way we determine, for example, that the parameters and the global variable in T in Figure 3 are all distinct.

c. the call graph.

The analysis method used in ECS for performing this step is described in [10]. It essentially scans each procedure, collecting up the information of interest into a binary matrix showing immediate relationships. It is in this form that the information is expanded to expose transitive relationships (e.g., the effects of calling a procedure which calls other procedures).

3.2 (Step 2) Establish a processing order on the collection of procedures, P_1, P_2, ... P_n. From the intial estimate (Step 1) or the revised estimate (Step 4) we know the possible invocation relationships in the collection and have a call graph. If the call graph is cycle free, we can readily determine an inverse invocation order [1] for a call graph C with nodes n_1, n_2, ... Consider the precedence relation, $\prec$: $n_i \prec n_j$ if and only if node n_i is an immediate predecessor of node n_j. The nodes of C can be given a linear order (n_1, n_2, ... n_ℓ) which satisfies the constraint that if $n_i \prec n_j$ then $i < j$ in the linear order. The inverse invocation order $\overline{K}$ = (n_ℓ ... n_2, n_1) is the inverse of the linear order. By processing the procedures in this order the control and data flow within a procedure will be determined before the procedures which call it are analyzed. One inverse invocation order for the example in Figure 1 is (C, E, D, B, A).

3.3 (Step 3) Establish the control and data relationships in P by processing the procedures in the order determined in step 2. A number of methods [3,5,6,7] exist for performing this analysis within each procedure. All of these methods presume that information about what data items are used and defined in each block is known and that the control flow can be determined. In establishing this information in the context of the interprocedural analysis algorithm, the effects of a procedure call are known since it will have been previously analyzed. Thus a call can be treated like any other statement when establishing the information about local uses, defines and control flow changes.

Reference [1] discusses a means of establishing the possible flow of external data items within a procedure. The approach is to treat each such item as if it were defined at the entry point and to determine from that what uses can be affected by it and whether or not such a definition could be preserved by the procedure. In this way we can deduce the effects of the procedure on data flow from outside.

In the Experimental Compiling System, the Interval analysis method [4] is used to establish the control flow relationships. Since this method iteratively combines subgraphs into blocks to form new graphs in which the nodes represent increasingly larger areas of the program, the data flow within a procedure can be hierarchially structured. Thus the data relationships between large areas of the procedure are given, then between the areas within each area, etc.

3.4 (Step 4) Update the estimates and interate. It is not at all clear at this time how valuable such an iteration would be. (ECS does not incorporate this feature yet.) It seems probable that one iteration would make substantial improvements but it is unlikely that more iterations could.

Before completing this section it is important to consider the obvious questions of algorithm cost and what to do if procedures are missing from the collection. First of all it should be observed that existing compilers which analyze one procedure at a time make "worst case" estimates for certain data item aliasing, particularly that associated with parameters and external variables, and for the effects of calls. When a procedure is missing we revert to that strategy.

As stated earlier we view the compiler as part of a larger system which manages programs. We would not envision reanalyzing an entire system of procedures each time one was changed. An intelligent procedure library management system should be able to deduce which procedures need to be reanalyzed when one is changed. Furthermore the programmer should probably be given some control over which procedures are to be included in the collection of procedures for analysis.

4. APPLICATIONS OF INTERPROCEDURAL ANALYSIS INFORMATION

The information collected by an interprocedural analyzer can be useful in a number of applications. In this section we will briefly sketch possible uses in three major areas:

a. documentation to the programmer

b. program management

c. program optimization

4.1 Documentation to the Programmer

As a result of performing the interprocedural analysis, a great deal of information is collected which is often not obvious or not available to the programmer. For example if the programmer is using procedures not created by him, he may not be fully aware of the effects of referencing these procedures. Furthermore the analysis will frequently expose relationships which he had not intended.

Three forms of documentation seem desirable:

a. error messages which draw the user's attention to definite or possible errors in the program. For example the analysis will find variables which are used before being defined, mismatches between arguments and parameters, unreferenced arguments, etc. These should probably be presented to the programmer even when not solicited.

b. annotated listings. The Experimental Compiling System has a listing annotator which automatically inserts comments into listings at certain points such as at procedure definition and reference points. These comments sum up the effects of the procedure or of making the reference. The annotation inserted after the CALL T(A,B) in procedure S of Figure 3 would contain the following:

   whether A was used and/or modified

   whether B was used and/or modified

   whether G and any other external variable was used and/or modified other invocations resulting from this invocation and the effects of such invocations.

In other words any effect an invocation can have on the invoking procedure will be included in the comments inserted at that point. Figure A8 contains a partial example.

c. documentation, preferably via an interrogation system which permits selective probes on the information. The amount of information produced is so voluminous that an unselective presentation would be overwelming. For example, all the uses of each definition in the program are known and all the definitions affecting each use. If a programmer is tracking such data flow it is easy to postulate a system which gives him the information as he requests it and which is, therefore, more meaningful to him than a vast dump of the information in which he has to track the flow.

4.2 Program Management

Whenever a change is being made to a procedure in a collection of procedures we often need to change other procedures in the collection and frequently would like to know what the effects of such a change are. For example if the programmer decides to eliminate the use of external variables from his system and uses the argument-parameter mechanism itself. From the interprocedural analysis information he could determine what procedures reference the externals, how the procedures are linked and then adjust the appropriate parameter lists on the basis of this information. It is not hard to visualize a program management system which, in fact, makes many of these changes automatically or with only a few assists from the user.

4.3 Program Optimization

A number of the well known program optimizations such as eliminating unused code, moving code out of loops, eliminating redundant expressions, can be applied in the context of procedure references as a result of the information collected. Consider the procedure fragment in Figure 4. If A and B are not changed in SUB then the expression A + B can be removed from the loop. In fact if SUB itself does not depend on anything in the loop and is reducible, i.e., the number of references to it can be changed, then it might be possible to remove the entire reference from the loop and effect a large program improvement.

In addition to increasing the utility of the traditional program optimizations, another form of optimization can be made on the basis of the information collected: procedure integration. In this multiple procedures can be combined into one procedure and the usual formal linkages eliminated.

5. SUMMARY

An algorithm has been given which collects control and data flow information from the collection of procedures which constitute all or part of a program. The information produced as a result of applying the algorithm was discussed with reference to a specific example in the appendix which illustrates the form of the information. This example shows some of the output of the Experimental Compiling System which currently contains a partial implementation of the algorithm. The algorithm presented here does not handle recursive procedures but current indications are that this method can be extended to accommodate them.

The information produced includes:

a. a call graph showing procedure relationships
b. the control flow of a procedure in the form of a graph
c. all the uses of each definition in a procedure and inversely all the definitions affecting each use
d. the external effects of a procedure in terms of what arguments and external variables it uses and/or modifies, what kind of return it makes, what procedures it invokes, etc.

All of the information is collected by a compile time and therefore, represents potential rather than actual relationships.

A brief discussion of possible applications for this information for documenting, maintaining and optimizing programs was given.

6. ACKNOWLEDGEMENTS

Ken Davies and Bob Tapscott have contributed more than the author to the material presented in this paper. The author wishes to thank them and the many others who have contributed to this work.

REFERENCES

[1] F. E. Allen, "Interprocedural Data Flow Analysis", Proceedings IFIP Conference 1974, North Holland Publishing Company, Amsterdam, 1974 (also as IBM Research Report RC4633, T. J. Watson Research Center, Yorktown Heights, N.Y., November, 1973).

[2] F. E. Allen, "A Basis for Program Optimization", Proceedings IFIP Conference 1971, North Holland Publishing Company, Amsterdam, 1971.

[3] F. E. Allen, "A Method for Determining Program Data Relationships", International Symposium on Theoretical Programming, Edited by Andrei Ershov and Valery A. Nepomniaschy, Lecture Notes in Computer Science, Vol. 5, Springer-Verlag, pp. 299-308, 1974.

[4] F. E. Allen, "Control Flow Analysis", Proceedings of a Symposium on Compiler Optimization, SIGPLAN Notices, July, 1970.

[5] Matthew S. Hecht and Jeffrey D. Ullman, "Analysis of a Simple Algorithm for Global Flow Problems", Conference Record of ACM Symposium on Principles of Programming Languages, Boston, Mass., October, 1973

[6] K. Kennedy, "A Global Flow Analysis Algorithm", International Journal of Computer Math., Vol. 3, pp. 5-15, December, 1971.

[7] Gary A. Kildall, "A Unified Approach to Global Program Optimization, Conference Record of ACM Symposium on Principles of Programming Languages, Boston, Mass., pp. 194-206, October, 1973.

[8] Barry K. Rosen, "Data Flow Analysis for Recursive PL/I Programs" (In preparation).

[9] J. Schwartz, "Inter-Procedural Optimization", SETL Newsletter #134, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, N.Y., N.Y., July 1, 1974.

[10] Thomas C. Spillman, "Exposing Side-Effects in a PL/I Optimizing Compiler", Proceedings of IFIP Congress 1971, North Holland Publishing Company, Amsterdam, 1971.

APPENDIX: An Example

The example given here illustrates some of the results of interprocedural analysis as currently available in the Experimental Compiling System. It will be noted that the example includes only nested procedures and does not have multiple external procedures: ECS has been designed to handle multiple procedures, both external and nested, but not all of the required components are currently available. Furthermore the example does not show a recursive procedure -- another feature which is currently unsupported.

```
PL/I CHECKOUT COMPILER    EXAMPLE: PROCEDURE(PARM);

                          SOURCE LISTING
   STMT LEV NT

     1       0 |  EXAMPLE: PROCEDURE(PARM);                                            |EXA00010
               |  /* A MEANINGLESS PROGRAM TO ILLUSTRATE ANALYSIS RESULTS. */          |EXA00020
               |                                                                       |EXA00030
     2   1   0 |     DECLARE  PARM  FIXED BINARY,                                      |EXA00040
               |              A     FIXED BINARY,                                      |EXA00050
               |              B     FIXED BINARY,                                      |EXA00060
               |              C     FIXED BINARY,                                      |EXA00070
               |              D     FIXED BINARY;                                      |EXA00080
     3   1   0 |     A = 1;                                                            |EXA00090
     4   1   0 |     B = 2;                                                            |EXA00100
     5   1   0 |     C = 3;                                                            |EXA00110
     6   1   0 |     D = 4;                                                            |EXA00120
     7   1   0 |     IF PARM > 0                                                       |EXA00130
               |        THEN CALL SUB(PARM,A,B,C);                                     |EXA00140
     8   1   0 |        ELSE CALL SUB(PARM,B,A,C);                                     |EXA00150
     9   1   0 |     RETURN;                                                           |EXA00160
               |                                                                       |EXA00170
    10   1   0 |  SUB: PROCEDURE (N,X,Y,Z);                                            |EXA00180
    11   2   0 |        DECLARE  N  FIXED BINARY,                                      |EXA00190
               |                 I  FIXED BINARY,                                      |EXA00200
               |                 X  FIXED BINARY,                                      |EXA00210
               |                 Y  FIXED BINARY,                                      |EXA00220
               |                 Z  FIXED BINARY,                                      |EXA00230
               |                 G  (10) FIXED BINARY EXTERNAL;                        |EXA00240
    12   2   0 |        DO I = 1 TO 10 BY 1;                                           |EXA00250
    13   2   1 |           G(I) = G(I) + D;                                            |EXA00260
    14   2   1 |        END;                                                           |EXA00270
    15   2   0 |        Z = X + Y;                                                     |EXA00280
    16   2   0 |        RETURN;                                                        |EXA00290
    17   2   0 |     END SUB;                                                          |EXA00300
    18   1   0 |  END EXAMPLE;                                                         |EXA00310
```

Figure A1

IDENTIFIER	ALIASES
Z	C,C
X	A,B
Y	B,A

CALL GRAPH

```
 -------------
|      1      |
|             |
|  (SYSTEM)   |
 -------------
       |
       |
       V
 -------------
|      2      |
|             |
| EXAMPLE     |
 -------------
       |
       |
       V
 -------------
|      3      |
|             |
| SUB         |
 -------------
```

```
PROCESSING ORDER WILL BE :
            SUB   EXAMPLE
```

Figure A2

*** ANALYSIS FOR SUB ON AUGUST 27, 1974 AT 10:32 AM 08.000 SECS. ***

FLOW GRAPH FOR SUB

```
            -------------
            |     1     |
            |           |
            |   0 -    0|
            -------------
                  |
                  |
                  V
            -------------
            |     2     |
            |           |
            |  10 -   12|
            -------------
                  |
                  |
                  V
            -------------
        --->|     3     |
        |   |           |
        |   |  12 -   12|----
        |   -------------   |
        |         |         |
        |         |         |
        |         V         |
        |   -------------   |
        |   |     4     |   |
        |   |           |   |
        *---|  13 -   14|   |
            -------------   |
                            |
                            |
                            |
            -------------   |
            |     5     |<---
            |           |
            |  15 -   16|
            -------------
```

Figure A3

DATA FLOW INFORMATION FOR PROCEDURE - SUB

DEFINITION - USE RELATIONSHIPS

IDENTIFIER	DEFINED AT STMT	DEFINED AT BLOCK	USED IN BLOCKS
D	0	1	4
Z	15	5	IS NOT USED OUTSIDE THE BLOCK
G	13	4	4
I	12	2	3 4
	14	4	3 4
X	0	1	5
Y	0	1	5

LIVE INFORMATION

IDENTIFIER	DEFINED AT STMT	DEFINED AT BLOCK	LIVE ON EDGES FROM	TO
D	0	1	1 2 3 4	2 3 4 3
Z	15	5	IS NOT LIVE	
G	13	4	3	4
I	12	2	2 3	3 4
	14	4	3 4	4 3
X	0	1	1 2 3 3 4	2 3 5 4 3
Y	0	1	1 2 3 3 4	2 3 5 4 3

Figure A4

*** ANALYSIS FOR EXAMPLE ON AUGUST 27, 1974 AT 10:32 AM
25.000 SECS. ***

FLOW GRAPH FOR EXAMPLE

```
 ------------
|     1      |
|            |
|  0 -     0 |
 ------------
      |
      |
      V
 ------------
|     2      |
|            |
|  1 -     7 |----
 ------------     |
      |           |
      |           |
      V           |
 ------------     |
|     3      |    |
|            |    |
|  7 -     7 |    |
 ------------     |
      |           |
      |           |
      V           |
 ------------     |
|     4      |    |
|            |    |
|  7 -     7 |----|----
 ------------     |    |
                  |    |
                  |    |
                  |    |
 ------------     |    |
|     5      |<---     |
|            |         |
|  8 -     8 |         |
 ------------          |
      |                |
      |                |
      V                |
 ------------          |
|     6      |<--------
|            |
|  9 -     9 |
 ------------
```

Figure A5

DATA FLOW INFORMATION FOR PROCEDURE - EXAMPLE

DEFINITION - USE RELATIONSHIPS

IDENTIFIER	DEFINED AT STMT	DEFINED AT BLOCK	USED IN BLOCKS
A	3	2	3 5
B	4	2	3 5
C	5	2	IS NOT USED OUTSIDE THE BLOCK
	7	3	IS NOT USED OUTSIDE THE BLOCK
	8	5	IS NOT USED OUTSIDE THE BLOCK
D	6	2	3 5
PARM	0	1	2

LIVE INFORMATION

IDENTIFIER	DEFINED AT STMT	DEFINED AT BLOCK	LIVE ON EDGES FROM	LIVE ON EDGES TO
A	3	2	2 2	5 3
B	4	2	2 2	5 3
C	5	2	IS NOT LIVE	
	7	3	IS NOT LIVE	
	8	5	IS NOT LIVE	
D	6	2	2 2	5 3
PARM	0	1	1	2

Figure A6

```
  7   1   0 |      IF PARM > 0                                                              |EXA00130
            |         THEN CALL SUB(PARM,A,B,C);                                            |EXA00140
+-----------------------------------------------------------ANNOTATION----------------------------------
|
|                  ARGUMENT  1 : PARM ,VARIABLE,NOT USED,NOT MODIFIED
|                  ARGUMENT  2 : A  ,VARIABLE,USED,NOT MODIFIED
|                  ARGUMENT  3 : B  ,VARIABLE,USED,NOT MODIFIED
|                  ARGUMENT  4 : C  ,VARIABLE,NOT USED,MODIFIED
|
|  NO SECONDARY INVOCATIONS MAY OCCUR AS AN INDIRECT RESULT OF THIS INVOCATION.
|
|                        THE VARIABLES OF THIS PROCEDURE INDIRECTLY MODIFIED BY THE INVOKED PROCED
|
|    C
|
|                         THE VARIABLES OF THIS PROCEDURE INDIRECTLY USED BY THE INVOKED PROCEDUR
|
|    A                                       B
|
|                            THE EXTERNAL VARIABLES DIRECTLY MODIFIED BY THE INVOKED PROCEDURE
|
|               G                                       IN SUB
|
|                              THE EXTERNAL VARIABLES DIRECTLY USED BY THE INVOKED PROCEDURE
|
|               G                                       IN SUB
|
|                          THE VARIABLES OF THIS PROCEDURE DIRECTLY USED BY THE INVOKED PROCEDURE
|
|    D
|
+---------------------------------------------------------END ANNOTATION-------------------------------
```

Figure A7

```
    9    1   0 |       RETURN;                                                              |EXA00160
               |                                                                            |EXA00170
   10    1   0 |    SUB: PROCEDURE (N,X,Y,Z);                                               |EXA00180
+-------------------------------------------------------------ANNOTATION-----------------------------
|
|  THIS PROCEDURE IS INVOKED AT:
|
|   STATEMENT-     7 IN EXAMPLE
|   STATEMENT-     8 IN EXAMPLE
|
|                                                        EXTERNAL VARIABLES USED
|
|       G
|
|                                                      EXTERNAL VARIABLES MODIFIED
|
|       G
|
|                                                  VARIABLES IN CONTAINING BLOCKS USED
|
|             A                                               IN EXAMPLE
|             B                                               IN EXAMPLE
|             D                                               IN EXAMPLE
|
|                                                VARIABLES IN CONTAINING BLOCKS MODIFIED
|
|             C                                               IN EXAMPLE
|
|                                                   PARAMETERS OF THIS PROCEDURE USED
|
|       X                                     Y
|
|                                                          LOCAL VARIABLES USED
|
|       G                                     I
|
|                                                 PARAMETERS OF THIS PROCEDURE MODIFIED
| 1|        Z
|
|                                                        LOCAL VARIABLES MODIFIED
|
|       G                                     I
|
+-----------------------------------------------------------END ANNOTATION----------------------------
```

Figure A8

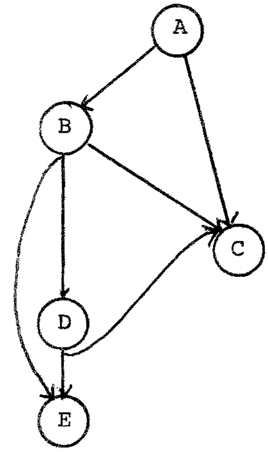

Figure 1

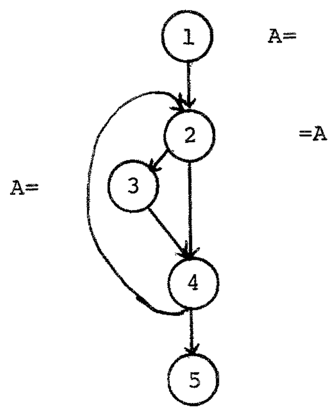

Figure 2

```
S: PROCEDURE;                    T: PROCEDURE (X,Y);
   DECLARE G EXTERNAL;              DECLARE G EXTERNAL;
         .
         .
         .
   CALL T (A,B);                    X =
         .
         .
         .
                                    END T;
   END S;
```

Figure 3

```
      .
      .
      .
DO I = 1 TO 100;
  CALL SUB (A,B);
  X(I) = A + B;
END;
```

Figure 4