Programming Techniques
G. Manacher, S. Graham Editors

A Program Data Flow Analysis Procedure

F.E. Allen and J. Cocke
IBM Thomas J. Watson Research Center

The global data relationships in a program can be exposed and codified by the static analysis methods described in this paper. A procedure is given which determines all the definitions which can possibly "reach" each node of the control flow graph of the program and all the definitions that are "live" on each edge of the graph. The procedure uses an "interval" ordered edge listing data structure and handles reducible and irreducible graphs indistinguishably.

Key Words and Phrases: program optimization, data flow analysis, flow graphs, algorithms, compilers
CR Categories: 4.12, 5.24

Authors' address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, N.Y. 10598.

1. Introduction

Given a definition of a data item in a program, it is frequently desirable to know what uses might be affected by the particular definition. The inverse is also true: for a given use the definitions of data items which can potentially supply values to it are of interest. Such data flow relationships, or "def-use" relationships, as they are often called, can be deduced by a static, compile time analysis of a program. A procedure for determining the information required for these def-use relationships is given in this paper.

Another data flow relationship which is also of interest is the following: given a program point (instruction) what data definitions are "live" at that point, that is, what data definitions given before this point are used after this point. This information is of interest, for example, when assigning index registers: data which is live at a point in a program might profitably be held in a index register at that point. The procedure given here includes the analysis required to expose live information.

In order to more precisely define the relationships derived by the procedure and to motivate the method used, certain basic concepts and constructs must be defined. This is done in Section 2. Section 3 gives the basic data flow analysis method and concludes with a brief survey of various methods. Section 4 discusses "intervals," the control flow basis used by the procedure given in this paper. In Sections 5 and 6 the procedure is described, with the latter section giving a PL/I implementation of the procedure. The paper concludes with a summary and an extended bibliography.

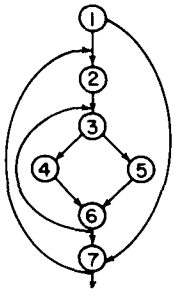
The data flow analysis procedure given here has been implemented and used in a PL/I oriented, Experimental Compiling System. Dr. Kenneth Kennedy of Rice University is responsible for the "live" analysis algorithm. Mr. Richard Stasko of IBM contributed to some of the data structure ideas used in the implementation. Dr. Jeffrey Ullman of Princeton, Dr. Matthew Hecht of the University of Maryland, Dr. Gary Kildall of the Naval Post Graduate School, Dr. Marvin Schaefer of SDC, Dr. Jacob Schwartz of New York University, and others have also made substantial contributions to this area.

2. Context and Problem Statement

Our approach is to derive and express the data flow relationships in terms of the control flow graph [3, 7]

of the program. For the purposes of this paper the control flow graph G of a program is a connected, directed graph having a single entry node n_0. G consists of a set of nodes, $N = \{n_0, n_1, n_2, \ldots, n_l\}$, representing sequences of program instructions and a set of edges, E, of ordered pairs of nodes representing the flow of control.

Fig. 1.



The graph depicted in Figure 1 has

$n_0 = 1$;
$N = \{1, 2, 3, \ldots, 7\}$ (the numbering is arbitrary); and
$E = \{(1, 2), (1, 7), (2, 3), \text{etc.}\}$

The *immediate successors* of a node n_i are all of the nodes n_j for which (n_i, n_j) is an edge in E. The *immediate predecessors* of node n_j are all of the nodes n_i for which (n_i, n_j) is an edge in E. A *path* is an ordered sequence of nodes $(n_1, n_2, \ldots, n_k)$ and their connecting edges in which each n_i is an immediate predecessor of n_{i+1}. A *closed path* or *cycle* is a path in which the first and last nodes are the same. The *successors* of a node n_i are all of the nodes n_j for which there exists a path $(n_i, \ldots, n_j)$. The *predecessors* of a node n_j are all of the nodes n_i for which there exists a path from n_i to n_j.

We can now more precisely define the data flow relationships we are interested in. A *data definition* is an expression or that part of an expression which modifies a data item. A *data use* is an expression or that part of an expression which references a data item without modifying it. A data definition potentially *affects* a use if the data items are the same and the result of the definition is available to the use. In order to determine which definitions affect which uses, two types of expression relationships can be distinguished: those relationships which exist between expressions within straight line sequences of code and those which exist in the context of control flow. Methods for finding and codifying the relationships in straight line sequences are relatively easy and will not be given in this paper. (Cocke and Schwartz [8] contains a discussion of one excellent method—value numbering.)

In this paper we are concerned with determining the data flow relationships that exist between collections of instructions. Two particularly interesting collections are now defined.

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). The nodes of the control flow graph can represent the basic blocks of a program. A node can also represent an *extended basic block*: a sequence of program instructions each of which, with the exception of the first instruction, has one and only one immediate predecessor and that predecessor precedes it, (though not necessarily immediately) in the extended basic block. An extended basic block can be formed from the tree of basic blocks such as those derived for IF ... THEN ... ELSE clauses. Again the data relationships internal to such blocks can be easily derived—a simple, stack oriented extension of the basic value numbering method can be used. The data methods in this paper will be given in terms of nodes which can represent basic blocks, extended basic blocks or, as will be seen, more complex collections of instructions and even entire procedures.

In order to introduce the basic data method more easily, several interesting sets of data flow information will now be defined in terms of basic blocks; some of these definitions will, however, later be revised to refer to the edges and paths on the control flow graph.

A *locally available definition* for a basic block is the last definition of the data item in the basic block.

Any definition of a data item X in the basic block is said to *kill* all definitions of the same data item reaching the basic block. Another way of expressing this is that all definitions of a data item which reach a basic block are *preserved* by the basic block if the data item is not redefined in the basic block.

A definition X in basic block n_i is said to *reach* basic block n_k if

1. X is a locally available definition from n_i,
2. n_k is a successor of n_i, and
3. There is at least one path from n_i to n_k which does not contain a basic block having a redefinition of the same data item; that is, X is preserved on some path from n_i to n_k.

The data flow analysis procedure given in this paper determines, among other things, the set of definitions, R_i, that reach each node in the control flow graph.

A *locally exposed use* in a basic block is a use of a data item which is not preceded in the basic block by a definition of the data item.

A use of a data item is *upwards exposed* in basic block n_i if either it is locally upwards exposed from n_i or there exists a path $(n_i \ldots n_k)$ such that the used data item is locally upwards exposed from n_k and there does not exist a j, $i \leq j < k$, which contains a definition of the data item. The upwards exposed uses at a basic block n_i can be expressed as a set U_i.

A definition, d, is *active* or *live* at basic block n_i if d reaches n_i (i.e. $d \in R_i$) and there is an upwards exposed use at n_i of the data item defined by d. The set L_i

of definitions active or live at basic block n_i is:

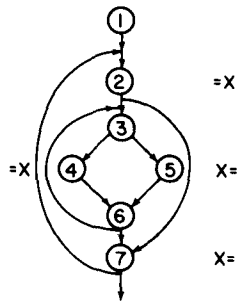
$$L_i = R_i \cap U_i .$$

(It is assumed that the elements of the R_i and U_i sets are encoded so that uses and definitions can be meaningfully intersected.) It should be noted that this definition differs slightly from some other definitions which have appeared in the literature [14, 19] in that it is the definition that is considered live rather than an exposed use of a data item.

Our procedure determines the upwards exposed uses U_i for each node n_i and the definitions which are live on each edge in the graph—this latter is a slight variant on the form of definition for L_i just given but is in fact a more useful formulation.

Consider the example in Figure 2.

Fig. 2.



Subscripting the definitions of X in nodes 5 and 7 in order to distinguish them we get the values shown in Table I for R_i, U_i, and L_i for each node n_i. It is evident from this example that one of the interesting side benefits of deriving this information is that possible uninitialized uses of data items are found.

In Section 3 the basic methods for deriving the reach information are given.

3. Basic Data Flow Analysis Method

A fundamental item of information which must be derived is what definitions of data items reach each node from other nodes in the graph. It should be readily apparent that the set of definitions which reach a node n_i is the union of the definitions available from the nodes which are immediate predecessors of n_i. Letting A_i denote the set of available definitions at node n_i and

Table I.

Node	R_i	U_i	L_i
1	$\varnothing$	X	$\varnothing$
2	X_7	X	X_7
3	X_5X_7	X	X_5X_7
4	X_5X_7	X	X_5X_7
5	X_5X_7	$\varnothing$	$\varnothing$
6	X_5X_7	X	X_5X_7
7	X_5X_7	$\varnothing$	$\varnothing$

R_i denote the set of definitions which reach n_i then:

$$R_i = \bigcup_p A_p \text{ for all } n_p \text{ which are immediate predecessors of } n_i . \quad (1)$$

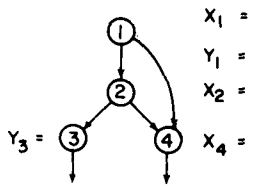
In order to formalize the construction of available definitions A_i for a node n_i, let DB_i be the set of locally available definitions of all data items defined in the node. Let PB_i be the set of all definitions in the program which are preserved through node i. Thus, if data item X is defined in the node then PB_i will not contain any definitions of X; on the other hand, if X is not defined anywhere in the node, then all the definitions of X in the program will be represented in PB_i. A_i is constructed by formula (2).

$$A_i = (R_i \cap PB_i) \cup DB_i . \quad (2)$$

(The B appended to P and D in the notation is used here and throughout the paper to indicate that the information expressed has been collected from the interior of the node.)

Consider the example in Figure 3. (The definitions are indexed with the node name in order to distinguish them.)

Fig. 3.



The DB and PB sets for each node are:

$$DB_1 = [X_1, Y_1], \quad PB_1 = [\varnothing],$$
$$DB_2 = [X_2], \quad PB_2 = [Y_1, Y_3],$$
$$DB_3 = [Y_3], \quad PB_3 = [X_1, X_2, X_4],$$
$$DB_4 = [X_4], \quad PB_4 = [Y_1, Y_3].$$

Using formulas (1) and (2) to process the nodes in the order 1, 2, 3, 4 yields:

$$R_1 = [\varnothing], \quad A_1 = [X_1, Y_1],$$
$$R_2 = [X_1, Y_1], \quad A_2 = [X_2, Y_1],$$
$$R_3 = [X_2, Y_1], \quad A_3 = [X_2, Y_3],$$
$$R_4 = [X_1, X_2, Y_1], \quad A_4 = [X_4, Y_1].$$

In the preceding example we were able to determine the definitions reaching each node in a single pass through the graph. This was possible because the graph did not contain cycles and we could therefore order the nodes so that a node was not processed until all of its predecessors were processed. Program control flow graphs usually contain cycles, however. There is no way in the presence of cycles to predictably determine the reaching information in one pass through the graph. In Figure 2, the definition of X in node 5 reaches node 3 (as well as 4, 5, 6, and 7). Since 3 is both a predecessor and successor of 5 a node ordering which allows reaching information to be propagated forward through the

paths of the graph without visiting the nodes more than once does not exist. In fact we need to say what happens to formula (1) when all of the predecessors of a node have not been visited and hence all of the A_i are not known.

The simplest algorithm for deriving the reaching information in a general control flow graph is given in the Basic Reach Algorithm.

Basic Reach Algorithm

Inputs: the PB_i and DB_i for each node in the control flow graph
Output: R_i, the set of definitions which reach each node in the graph. The set A_i of definitions available from each node is also created.
Method:
1. Initialize all of the A_i and R_i to the null set.
2. Perform step 3 while there is no change in any R_i or A_i.
3. Apply formula (1) followed by formula (2) to the nodes of the graph. □

In an excellent paper on "A Unified Approach to Global Program Optimization," Kildall [17] gives a generalization of this algorithm and proves that the information does indeed stabilize.

Clearly the rapidity with which the information stabilizes for a given graph very much depends on the order in which the nodes of the graph are examined. In [10], Hecht and Ullman use a node ordering established by applying Tarjan's depth first spanning tree algorithm to the control flow graph. The procedure given in this paper is based upon the use of intervals [3, 4, 5, 7].

4. Intervals

Given a node h, an *interval* $I(h)$ is the maximal, single entry subgraph in which h is the only entry node and in which all closed paths contain h. The unique interval node h is called the *interval head* or simply the *header node*. An interval can be expressed in terms of the nodes in it: $I(h) = (n_1, n_2, \ldots, n_m)$.

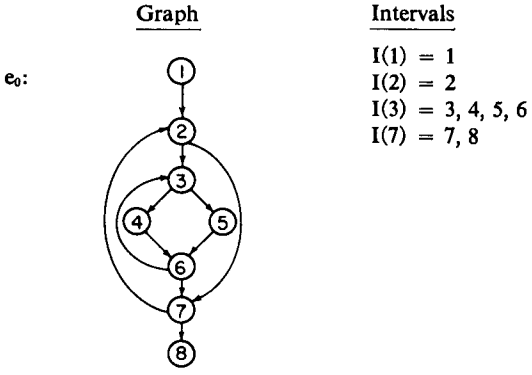
By selecting the proper set of header nodes, a graph may be partitioned into a unique set of disjoint intervals $\mathcal{I} = \{I(h_1), I(h_2) \ldots\}$. An algorithm for such a partition is:

Algorithm for Finding Intervals

1. Establish a set H for header nodes and initialize it with n_0, the unique entry node for the graph.
2. For $h \in H$ find $I(h)$ as follows:
   2.1. Put h in $I(h)$ as the first element of $I(h)$.
   2.2. Add to $I(h)$ any node all of whose immediate predecessors are already in $I(h)$.
   2.3. Repeat 2.2 until no more nodes can be added to $I(h)$.
3. Add to H all nodes in G which are not already in H and which are not in $I(h)$ but which have immediate predecessors in $I(h)$. Therefore a node is added to H the first time any (but not all) of its immediate predecessors become members of an interval.
4. Add $I(h)$ to a set $\mathcal{I}$ of intervals being developed.
5. Select the next unprocessed node in H and repeat steps 2, 3, 4, 5. When there are no more unprocessed nodes in H, the procedure terminates. □

Figure 4 illustrates the partitioning of a graph into intervals.

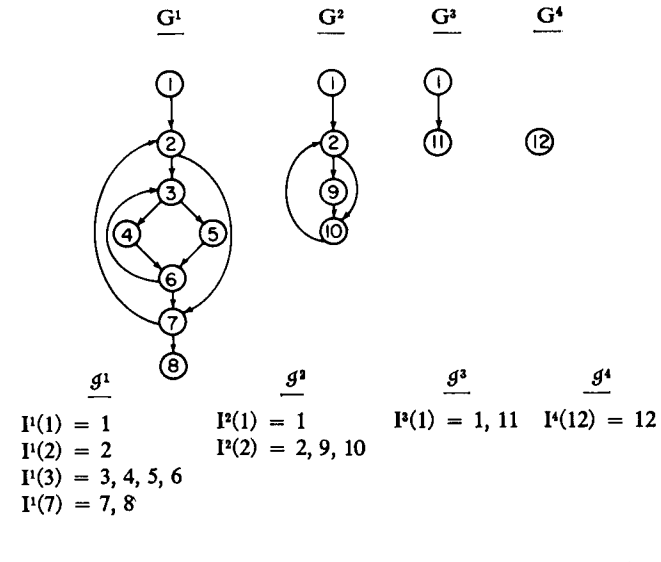
Fig. 4.



A property of intervals which is of particular interest in data flow analysis is the order of nodes in an interval list (called the *interval order*). The order is such that if the nodes on a interval list are processed in the order given then all interval predecessors of a node reachable along loop-free paths from the header will have been processed before the given node.

The intervals described thus far have been formed from the nodes given in the initial control flow graph. For reasons which will be apparent shortly, these intervals are called the *basic* or *first order intervals* and the graph from which they were derived is called the *basic* or *first order graph*. A superscript notation is used to designate the order, e.g. $I^1(h) \in \mathcal{I}^1$.

Fig. 5.

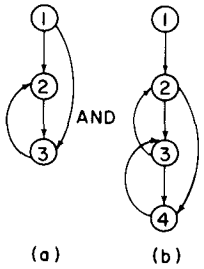


The *second order graph* is derived from the first order graph and its intervals by making each first order interval into a node and each interval exit edge into an edge in the second order graph. *Second order intervals* are the intervals in the second order graph. Since the nodes of the second order intervals are first order intervals,

this nesting can be used when determining inter-interval relationships. Successively higher order graphs can be derived until the nth order graph consists of a single node. Figure 5 illustrates such a sequence of derived graphs.

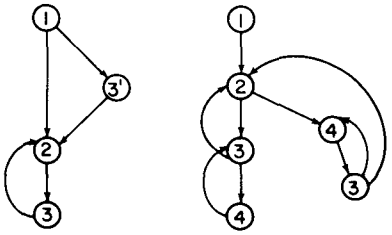
Not all graphs reduce to a single node by the iterative derivation indicated. A *reducible graph* is a graph whose *n*th order derived graph is a single node. A *irreducible graph* is a graph for which there does not exist an *n*th order derived graph consisting of a single node. Examples of irreducible graphs are given in Figure 6.

Fig. 6.



Methods for "splitting" (copying) certain nodes in an irreducible graph to produce an equivalent graph containing intervals exist [1, 7, 20]. Figure 7 illustrates

Fig. 7.



split graphs for the two irreducible forms given in Figure 6. It should be emphasized that splitting is an analysis technique and does not imply that the code in the nodes will appear more than once in the generated program. It is interesting to note that a program written by one of the authors to analyze the control flow of 75 "real" Fortran programs found that over 90 percent of the control flow graphs are reducible [7]. Further, Knuth [18] found no irreducible graphs in 50 Fortran programs.

The data flow analysis described in the next section is based on the interval construct just described.

5. Data Flow Analysis Using Intervals

By definition and by construction, an interval $I(h)$ has only one entry, h, and all closed paths go through h. The definitions which reach each basic node in the interval could be determined, therefore, by formulas (1) and (2) if all of the definitions which reach h, both from predecessors outside the interval and from predecessors in the interval, were known. These definitions can be found by a two-phase process.

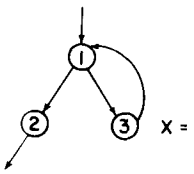
The first phase of the process collects information and the second phase propagates it to the appropriate places. The first phase first collects information about what is defined and preserved in each first-order interval by processing the nodes in each interval in their interval order. The information collected is then posted to the node representing each interval in the second order derived graph. The second order intervals are then processed in the same way as the first order intervals and the process iterates.

For each interval at each level during the first phase, two items of information are collected from the nodes which comprise it (three items if the interval contains a loop):

1. The definitions which are defined in the interval and locally available from it. This becomes the *DB* set for the node representing the interval in the next higher order graph.
2. The definitions which are preserved by the interval. This becomes the *PB* set for the representative node.

If the interval contains a loop, then a third item of information is collected: which definitions in the interval can reach the interval head. A set R_h associated with the interval head expresses this. During the second phase the R_h sets will make definition information available before the nodes containing the definitions have been processed. They are also used in the first phase to determine what definitions are available at interval exit points in cases such as depicted in Figure 8. There the

Fig. 8.



definition of X in node 3 reaches the interval head and, assuming all definitions of X are preserved on the path from 1 through 2, X_3 is available on exit from the interval. The effect of the first phase then is to percolate the influence of each node outwards into an increasingly more global context.

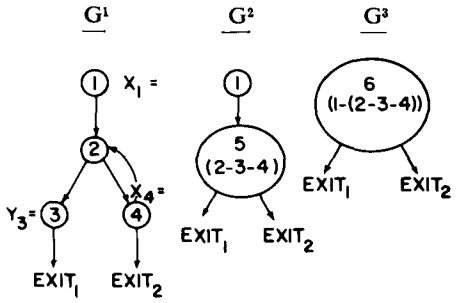
The second phase uses the information collected by the first phase and, by processing the graphs from high order to low order, generates the set of definitions reaching each node. Before processing an interval the R_h set associated with the header is modified to reflect those definitions that can reach the interval from outside. These are given in the R set generated for the interval's representative block in the next higher order graph. The modified R_h set is used to initiate processing the inter-

val. The order of the process is the interval order—the operations of the process are those given in formulas (1) and (2).

Since a node can represent collections of instructions having internal flow and thus different paths from the entry to the different exits, the data items defined and preserved on the different exits will generally be different. In order to reflect these distinctions, the definitions defined and preserved, both those initially given and those accumulated within an interval during phase I, will be associated with the edges of the graph rather than with the nodes.

Furthermore, when formula (2) in phase II determines the definitions available from a node and its predecessors, that information will be associated with the exit edges of the nodes since it may be different for the different exits from the node. Consider the example in Figure 9.

Fig. 9.



Assuming all the nodes in G^1 represent basic blocks, then the initial DB and PB sets are given in Table II. (e_1 represents exit$_1$ and e_2 represents exit$_2$.) As a result of determining (in phase I of the analysis) what may be defined and preserved through the nodes in the graphs, we get the results given in Table III.

Furthermore at the end of phase I we have $R_2 = \{X_4\}$. Assuming no definitions reach the entry node, the definitions reaching each node (and available on each edge) after phase II are given in Table IV.

It will be noted that we have not only determined the data items which reach each node in the first order graph and which are available on each edge in that graph but we have also determined that information for the nodes and edges of the higher order graphs. Furthermore, we have determined the data items defined and preserved along the edges of the higher-order graphs. We have thus collected up data flow relationships between large sections of the program and indeed know what data items are defined on exit from the program and what data items may be preserved during an execution of the program. Interprocedural data flow analysis as described in [6] exploits these characteristics of the analysis method. An algorithm which determines what definitions may reach each node in the graphs is now given.

Reach Algorithm (Interval Based)

Inputs

1. The ordered set of graphs $(G^1, G^2, \ldots, G^n)$ determined by interval analysis.
2. The intervals in each graph with their nodes given in interval process order.
3. The definitions defined and preserved on each edge in the first order graph. These are expressed in the DB and PB sets.

Outputs

A set R of the definitions that reach each node.
A set A of the definitions available on each edge.

Steps

Phase I

1. For each graph, G^g, in the order $G^1, G^2, \ldots, G^{n-1}$, perform steps 2 and 3.
2. If the current graph is not G^1 then initialize the PB and DB sets for the edges of the graph. This is done by first identifying the edge in G^{g-1} to which each edge in G^g corresponds (these will be interval exit edges). Then, using the information generated during step 3 for G^{g-1}, for each edge i in G^g with corresponding exit edge x from interval with head h in G^{g-1}, set:
   2.1. $PB_i = P_x$ and
   2.2. $DB_i = (R_h \cap P_x) \cup D_x$
3. For each exit edge of each interval in G^g determine P, the definitions preserved on some path through the interval to the exit, and D, the definitions in the interval that may be available on the exit. These are determined by finding P and D for each edge in the interval:
   3.1. For each exit edge i of the header node:

$$P_i = PB_i$$
$$D_i = DB_i$$

   3.2. For each exit edge i of each node j ($j = 2, 3, \ldots$) in interval order:

$$P_i = (\underset{p}{\cup} P_p) \cap PB_i$$

$$D_i = ((\underset{p}{\cup} D_p) \cap PB_i) \cup DB_i \text{ for all } p \text{ input edges to node } j.$$

While processing an interval determine the set of definitions, R_h, that can reach the interval head, h, from the inside the interval by:

$$R_h = \underset{l}{\cup} D_l$$

for all interval edges l which enter h, (sometimes called *latching edges* or *latches*). If there are none set $R_h = \varnothing$.

Between phase I and II the R vector for the single node in the nth order derived graph is initiated: $R_1 = \Phi$ or whatever set of definitions is known to reach the program from outside.

Phase II

1. For each graph, G^g, in the order $G^{n-1}, \ldots, G^2, G^1$, perform steps 2 and 3.
2. For each node i in G^{g+1} form $R_h = R_h \cup R_i$ where h is the head of the interval in G^g which i represents in G^{g+1}.
3. For each interval process the nodes in interval order determining the definitions reaching each node and available on each node exit edge as follows:
   3.1. For each exit edge i of the header node h

$$A_i = (R_h \cap PB_i) \cup DB_i$$

   3.2. For each node j ($j = 2, 3, \ldots$) in interval order first form

$$R_j = \underset{p}{\cup} A_p \text{ for all input edges } p \text{ to } j$$

then for each exit edge i of j form

$$A_i = (R_j \cap PB_i) \cup DB_i$$

□

Table II.

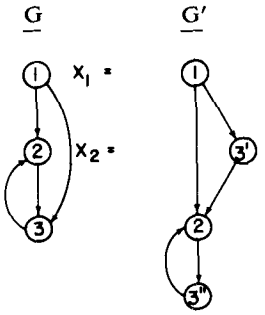
Edge	For G¹ DB	PB
1–2	X_1	Y_3
2–3	$\varnothing$	$X_1X_4X_3$
2–4	$\varnothing$	$X_1X_4X_3$
3–e_1	Y_3	X_1X_4
4–2	X_4	Y_3
4–e_2	X_4	Y_3

Table III.

Edge	For G² DB	PB	Edge	For G³ DB	PB
1–5	X_1	Y_3	6–e_1	$X_1X_4X_3$	$\varnothing$
5–e_1	X_4Y_3	X_1X_4	6–e_2	X_4	Y_3
5–e_2	X_4	Y_3			

We need to account for the effects of irreducible graphs on the algorithm. There are, in fact, no effects. Figure 10 shows a graph in its irreducible then split forms.

Fig. 10.



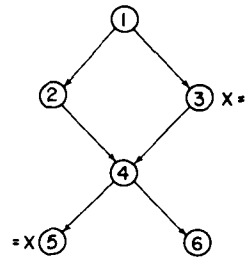
The irreducible graph G is either the initial graph or is a graph derived from the initial graph. It does not matter since in either case the nodes can be treated as interval heads. (Some efficiency may accrue from eliding the actual treatment of these as heads of single node intervals if this is not the initial graph.) Treating these as intervals heads, phase I will derive P_i and D_i for each edge. In order to continue to propagate outwards the effects of each node we treat the split form of the graph, G', as if it were a higher order graph. Step 2 of phase I will initialize the DB_i and PB_i sets for the edges in the higher order graph as usual and in the process the D and P sets for edge 3–2 will be picked up twice—once for 3′–2 and once for 3″–2. During phase II the data definitions which reach node 3 in the irreducible graph will be found by step 2. By this step we will, as regards node 3, form $R_3 = R_3 \cup R_{3'}$, and then $R_3 = R_3 \cup R_{3''}$. In this case, since R_3 was initially null, we have, in effect, formed $R_3 = R_{3'} \cup R_{3''}$. Thus the data items that are known to reach node 3 are those that could reach it on either path.

It should be apparent from this description that the existence of an irreducible graph is transparent to the algorithm and further that node splitting is an analysis device and does not result in duplicated code.

We now turn to the derivation of the use and live information.

An upwards exposed use of a data item at a node b_i is, it will be recalled from Section 2 of this paper, a use which is either locally upwards exposed from b_i or from some successor of b_i which can be reached without going through a redefinition of the data item. Given the upwards exposed uses U_i and the definitions R_i which reach a point in the program, the live information can be readily determined. This, however, gives the data items that are live on entry to a node. While this is useful it is frequently desirable, during register allocation, for example, to know what is live on exit from a node. Rather than retain both sets, the live information can be found for the edges and the node related live information trivially constructed when necessary. Consider the example in Figure 11.

Fig. 11.



By storing the fact that X is live on edges 3–4 and 4–5, the fact that X is live on entry to node 4 and on exit from node 4 as well as the fact that X is not live on exit from node 2 and on entry to node 6 can be readily deduced.

Table IV.

Node	For G³ R	Edge	A	Node	For G² R	Edge	A	Node	For G¹ R	Edge	A
6	$\varnothing$	6–e_1	$X_1X_4X_3$	1	$\varnothing$	1–5	X_1	1	$\varnothing$	1–2	X_1
		6–e_2	X_4	5	X_1	5–e_1	$X_1X_4X_3$	2	X_1X_4	2–3	X_1X_4
						5–e_2	X_4	3	X_1X_4	2–4	X_1X_4
								4	X_1X_4	3–e_1	$X_1X_4X_3$
										4–2	X_4
										4–e_2	X_4

In order to compute the live information for the edges of the graph rather than for the nodes, the set of the definitions, A, available on each edge is used. For a given edge e, incident into node i

$$L_e = A_e \cap U_i.$$

Since A_e was derived by the Reach Algorithm, the basic item of information which must be derived is U_i. The process of determining which uses reach backwards through the graph is analogous to that of determining which definitions reach forwards. Hence to determine what is upwards exposed from a node involves knowing what is upwards exposed from the successors of a node and what data items are preserved through the node.

The algorithm for generating the U_i sets for the nodes of the graph again requires two phases, the first of which can be conveniently imbedded in the first phase of the Reach Algorithm. Given the locally upwards exposed uses from each node in the basic graph, the purpose of the first phase is to find the upwards exposed uses from each interval and hence the locally upwards exposed uses for each node in the derived graph. The upwards exposed uses from each interval is found as follows:

1. Prior to processing each interval initiate a set U_h which will contain the uses upwards exposed in the interval:

   $$U_h = UB_h$$

2. For each node i in the interval ($i = 2, 3, \ldots$) update U_h with the locally exposed uses UB_i in i which can be preserved along some path from h to i:

   $$U_h = U_h \cup ((\bigcup_p P_p) \cap UB_i$$

   where P_p is the set of data items preserved on input edge p to node i and whose computation is given in the Reach Algorithm.

Each U_h is used to initiate the UB_i of the node node (or nodes in case of node splitting) which represents the interval in the next higher order graph.

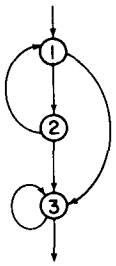
Having determined the set of uses locally upwards exposed from each node in all of the graphs, the second phase can determine the U_i for each node. This is done by processing the graphs from high order to low order and going backwards through the nodes in each interval. The data uses upwards exposed at a node are those which are locally upwards exposed or are upwards exposed from its successors and are preserved by the node. The basic formula used in the second phase is

$$U_i = ((\bigcup_s U_s) \cap PB_i) \cup UB_i$$

where U_s is the set of upwards exposed uses from the immediate successors of node i.

When traversing backwards through the nodes of an interval (i.e. in inverse interval order), the U sets of all immediate successors which are interior to an interval will have been computed before the U_i of a node needs to be computed. Two other kinds of immediate successors can exist. In Figure 12, which has intervals (1, 2) and (3), node 2 has as immediate successors

Fig. 12.

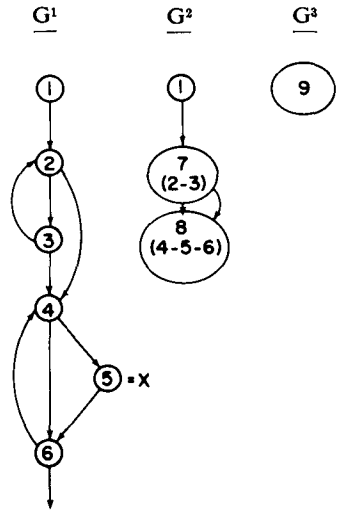


the head of the interval it belongs to, node 1, and the head of another interval, node 3. When computing U_2, the values of U_1 and U_3 are known as are the values of the U_i at all the interval heads for the graph being processed. These sets are known because:

1. During phase I the uses U_h upwards exposed from within each interval are determined then.
2. During phase II each U_h is updated with the value of U_I, the upwards exposed uses found for the node representing the interval in the next higher order graph.

Figure 13 shows a series of derived graphs with a use of X in node 5.

Fig. 13.



After phase I, the U_h values for G^1 are $U_1 = \varnothing$, $U_2 = \varnothing$, and $U_4 = X$ and for G^2 they are $U_1 = X$. Furthermore, after phase I $UB_1 = \varnothing$, $UB_7 = \varnothing$ and $UB_8 = X$. Going backwards through G^2 during phase II leads to

$$U_8 = X, \quad U_7 = X, \quad U_1 = X,$$

The U_h values for nodes 1, 2, and 4 are then updated before the intervals in G^1 are processed. This leads to $U_1 = X$, $U_2 = X$, and $U_4 = X$. Even if the interval con-

sisting of (2, 3) is processed before (4, 5, 6), the U_i for the two successors of node 3 are known since $U_2 = X$ and $U_4 = X$ have already been established.

As has already been observed, phase I of the live analysis algorithm can be embedded in phase I of the Reach Algorithm. Since the second phase of the live algorithm requires a backwards pass through the graph it cannot be embedded in phase II of the Reach Algorithm. The combined algorithm therefore requires three phases. We will not give any more details on this in algorithmic form since the procedure in the next section gives a more definitive version of the combined data flow analysis.

6. A Data Flow Analysis Procedure

A PL/I program, DATFLOW, which finds the definitions that reach each node, the uses that are upwards exposed from a node and its successors, and the definitions that are live on each edge of the control flow graph is given in this section. Before giving the procedure, two representational techniques which greatly affect the actual speed of the algorithm are described. We first describe the representation of the data sets and then the representation of the control flow information.

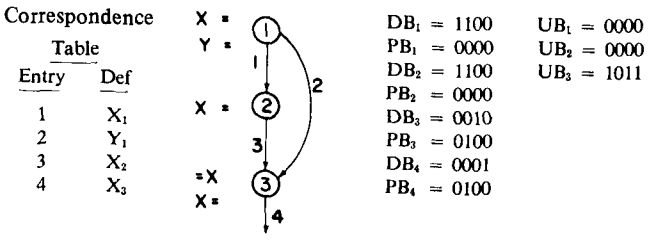
Associated with each node i on input to the procedure is the set of uses, UB_i, which are locally upwards exposed from the node; associated with each edge e are the DB_e and PB_e sets. DB_e contains the data definitions defined in the node and available on the exit edge e. PB_e contains the data definitions in the program which may be preserved through the node when exit edge e is taken. The operations on these sets and on others formed by the analysis are almost entirely intersection and union. By expressing the sets as comparable bit vectors, the more rapid boolean operations of *and* and *or* can be used.

Each bit in the vectors represents a definition in the program. A correspondence table is used to correlate the bit vector entry with the definition point: the ith bit in a bit vector corresponds to the ith entry in the correspondence table where the program location of the definition is recorded.

An obvious question arises: how are the UB and U sets encoded so that they can be meaningfully intersected with the vectors of available definitions? A use of a data item is multiply encoded: if X is the data item being used then that use is encoded by using all the bit entries for definitions of X. Figure 14 shows the DB, PB, and UB vectors for the three nodes and four edges.

We now turn to the representation of the control flow. The nodes of the graphs have been arbitrarily numbered from 1 to n with all the nodes in a given graph having a contiguous sequence of numbers which is larger than those of the next lower order graph. These numbers are used to uniquely identify the node and in the DATFLOW procedure to index any data associated

Fig. 14.



with it. The edges are similarly though independently numbered from 1 to m. A table, GRAPHS, has an entry for each graph which gives the first and last node and the first and last edge in the graph. The table is ordered by graph level with the initial (low order) graph as the first entry.

Associated with the graphs is a node ordered edge listing (called EDGES TABLE in the program). It has the following characteristics:

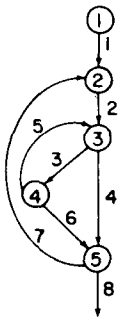
1. Every edge in the graph appears in the table twice: once as the in-edge of a node and once as an out-edge.
2. All of the edges incident to a node are given in sequential entries with all in-edges appearing before all out-edges.
3. Each entry contains the node index and the edge index as well as indicators to indicate whether or not the node is an interval header, whether the edge is an in-edge or an out-edge and if an out-edge whether or not it is a latch (is an in-edge to the head of the interval containing it).
4. The interval process order on the nodes is used to determine the order of the entries in the table. All entries for an interval are grouped together with the entries for the header node first. The order of entries for the different intervals in a given graph is not important for the procedure.

Kennedy [16] has recently proposed a node listing approach to data flow analysis which is of $O(l)$ where l is the length of the list. Aho and Ullman [2] have given an algorithm for producing such a listing for reducible graphs. In their algorithm a listing whose length is $O(n \log n)$ can be found for a reducible graph of n nodes. Furthermore if the number of edges is of $O(n)$, then the listing can be found in $O(n \log n)$ time. Since the procedure given here associates information with both edges and nodes it is roughly $O(l' + n')$ where l' is the number of entries in EDGES TABLE and n' is the number of nodes in the collection of graphs.

Figure 15 shows the entries for the first order graphs; the entries for the second order would follow.

Processing a graph in a phase of the algorithm involves making a pass through the EDGES TABLE—forwards for the phases that compute reaching information and backwards for the second phase of the live computation (phase III in the procedure). The node and edge fields index the tables where the information is

Fig. 15.



Edges Table

Node	Edge	In-edge	Out-edge	Latch	Head
1	1		√		√
2	1	√			√
2	7	√			√
2	2		√		√
3	2	√			√
3	5	√			√
3	3		√		√
3	4		√		√
4	3	√			
4	5		√	√	
4	6		√		
5	4	√			
5	6	√			
5	7		√		
5	8		√		

accumulated. Since all in-edges to a node appear before all out-edges and since all interval predecessors of a node appear before the node, the in-edges for a node will index information calculated when the edges appeared as out-edges of interval predecessors. Consider, for example, the phase II calculation of the reach and available vectors for the nodes and edges of the graph given in Figure 15. Assuming that the reach vectors for the interval heads have been established and the reach vectors for the other nodes have been initialized to zero, we would make one pass through the EDGES TABLE. For each entry i with edge index, EDGE, and node index, NODE:

1. If the entry is for an in-edge to a node which is not a header then $R(\text{NODE}) = R(\text{NODE}) \vee A(\text{EDGE})$
2. Otherwise if the entry is for an out-edge $A(\text{EDGE}) = R(\text{NODE}) \wedge (PB(\text{EDGE}) \vee DB(\text{EDGE}))$.

Consider the three entries for node 4 in Figure 15. processing the first entry gives the reach for node 4, the next two entries cause the calculation of the definitions, A, available on edges 5 and 6. As we continue through the table we will use A_6 when calculating R_5.

The Appendix gives the PL/I procedure DATFLOW, which determines the reach, live, and use information.

7. Summary and Final Remarks

A procedure has been given which determines the data flow relationships in a program by a static, global analysis. Given a control flow graph representation of a program and information about the data items used, defined and preserved by each node in the graph the following information can be determined by the described methods:

1. What data definitions reach each node in the graph and can therefore affect uses in the block represented by the node.
2. What uses of data items are upwards exposed from each node and its successors and hence can be affected by definitions of the items.
3. What definitions are "live" on each edge of the graph.

The approach used is based on intervals and has the following features:

1. Certain information is associated with edges rather than nodes.
2. Irreducible and reducible graphs are treated indistinguishably.
3. The data flow characteristics in large program units (the nodes of the higher order graphs) are subsummed.
4. An interval ordered edge listing together with a bit vector representation of the relationships is used to give a fast implementation.

Received February 1975; revised May 1975

References

1. Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling, Vol. 2.* Prentice-Hall, Englewood Cliffs, N.J., 1973.
2. Aho, A.V., and Ullman, J.D. Node listings for reducible flow graphs. Proc. 7th Ann. Symp. on the Theory of Computing, Albuquerque, N.M., May 1975, pp. 177–185.
3. Allen, F.E. Control flow analysis. SIGPLAN Notices (ACM Newsletter) *5* (July 1970), 1–19.
4. Allen, F.E. A basis for program optimization. Proc. IFIP Congress. North-Holland Pub. Co., Amsterdam, 1971, pp. 385–390.
5. Allen, F.E. A method for determining program data relationships. In *Lecture Notes in Computer Science, Vol. 5, International Symposium on Theoretical Programming*, Springer-Verlag, Berlin, 1974, pp. 299–308.
6. Allen, F.E. Interprocedural data flow analysis. Proc. IFIP Congress, North-Holland Pub. Co., Amsterdam, 1974, pp. 398–402; and IBM Research Rep. RC 4633, Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1973.
7. Allen, F.E., and Cocke, J. Graph theoretic constructs for program control flow analysis. IBM Research Report RC 3923, Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1972.
8. Cocke, J., and Schwartz, J.T. *Programming Languages and their Compilers: Preliminary Notes.* Courant Institute of Mathematical Sciences, New York U., N.Y., 1969.
9. Graham, S.L., and Wegman, M., A fast and usually linear algorithm for global flow analysis. Conf. Rec., 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, Calif., Jan. 1975, pp. 22–34.
10. Hecht, M.S., and Ullman, J.D. Analysis of a simple algorithm for global flow problems. Conf. Record, ACM Symp. on Principles of Programming Languages, Boston, Mass., Oct. 1973, pp. 207–217.
11. Hecht, M.S., and Ullman, J.D. Flow graph reducibility. *SIAM J. Computing 1*, 2 (June 1972), 188–202.
12. Hecht, M.S., and Ullman, J.D. Characterizations of reducible flow graphs. *J.ACM 21*, 3 (July 1974), 367–375.
13. Kam, J.B., and Ullman, J.D. Global optimization problems and iterative algorithms. TR–146, Computer Sci. Lab., Princeton U., Princeton, N.J., 1974.
14. Kennedy, K. A global flow analysis algorithm. *Internat. J. Computer Math. A 3* (1971), 5–15.
15. Kennedy, K. A comparison of algorithms for global flow analysis. Tech. Rep. 476–093–1, Dep. of Mathematical Sciences, Rice U., Houston, Texas, 1974.
16. Kennedy, K. Node listings applied to data flow analysis. Conf. Rec., 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, Calif., 1975, pp. 10–22.
17. Kildall, G.A. A unified approach to global program optimization. Conf. Record, ACM Symp. on Principles of Programming Languages, Boston, Mass., 1973. pp. 194–206.
18. Knuth, D.E. An empirical study of FORTRAN programs. *Software–Practice and Experience 1*, 2 (1971), 105–134.
19. Kou, L.T. On live-dead analysis for global data flow problems. IBM Research Rep. RC 5278, Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1975.
20. Schaefer, M. *A Mathematical Theory of Global Program Optimization.* Prentice-Hall, Englewood Cliffs, N.J., 1973.

Appendix. The PL/I Procedure DATFLOW

```
/************************  DATFLOW  ********************************/
/* DATFLOW DETERMINES THE DATA DEFINITIONS THAT REACH EACH NODE FROM */
/*   PREDECESSOR NODES, THE USES THAT ARE UPWARDS EXPOSED AT A NODE  */
/*   FROM BOTH THE NODE AND FROM SUCCESSOR NODES, AND THE DATA       */
/*   DEFINITIONS THAT ARE ALIVE ALONG EACH EDGE. THIS INFORMATION    */
/*   IS EXPRESSED IN THE REACH, USED AND LIVE VECTORS RESPECTIVELY.  */
/* DATFLOW INPUT IS A DESCRIPTION OF THE PROGRAM CONTROL FLOW        */
/*   RELATIONSHIPS, AND LOCAL INFORMATION ABOUT THE DATA ITEMS USED, */
/*   DEFINED AND PRESERVED ON THE NODES AND EDGES OF THE GRAPH.      */
/*   THE PROGRAM CONTROL FLOW RELATIONSHIPS ARE GIVEN IN GRAPHS,     */
/*   EDGES_TABLE, FROM_TO_TABLE, CORRES_HEAD, AND CORRES_EDGE.       */
/*   THESE RELATIONSHIPS ARE DETERMINED BY THE INTERVAL ANALYZER.    */
/*   THE LOCAL INFORMATION ABOUT THE DATA ITEMS IS GIVEN IN          */
/*   EDGE_DATA AND IN NODE_DATA.                                     */
/* GRAPHS CONTAINS ONE ENTRY FOR EACH GRAPH (BASIC, DERIVED, OR      */
/*   SPLIT). EACH ENTRY CONTAINS INDICES FOR THE FIRST AND LAST NODE */
/*   AND THE FIRST AND LAST EDGE.                                    */
/* EDGES_TABLE CONTAINS 2 ENTRIES FOR EACH EDGE IN THE GRAPHS.       */
/*   THE TABLE IS ORDERED SO THAT ALL NODES IN AN INTERVAL ARE IN THE*/
/*   TABLE IN INTERVAL ORDER. FURTHER ALL THE ENTRIES FOR EDGES      */
/*   ENTERING A NODE IMMEDIATELY PRECEDE THE EXIT EDGES. ALL NODES   */
/*   FOR A GRAPH ARE GROUPED. THE TABLE ORDER IS USED AS THE ORDER   */
/*   FOR PROCESSING THE INFORMATION.                                 */
/* FROM_TO_TABLE CONTAINS 2 ENTRIES FOR EACH EDGE IN THE GRAPHS:     */
/*   THE NODE THE EDGE COMES FROM AND THE NODE IT ENTERS.            */
/* CORRES_HEAD CONTAINS AN ENTRY FOR EACH NODE IN THE HIGHER ORDER   */
/*   GRAPHS: THE HEAD OF THE INTERVAL WHICH THIS NODE REPRESENTS.    */
/*   IF THE GRAPH IS THE RESULT OF SPLITTING THEN MORE THAN ONE NODE */
/*   MAY INDEX THE SAME NODE IN THE NEXT LOWER ORDER GRAPH.          */
/*   THIS TABLE IS USED WHEN NODE RELATED INFORMATION IS POSTED UP   */
/*   (DURING PHASE I) OR DOWN (DURING PHASE II).                     */
/* CORRES_EDGE CONTAINS AN ENTRY FOR EACH EDGE IN THE HIGHER ORDER   */
/*   GRAPHS: THE EDGE IN THE NEXT LOWER ORDER GRAPH WHICH THIS EDGE  */
/*   REPRESENTS. IF THE GRAPH IS THE RESULT OF SPLITTING THEN MORE   */
/*   THAN ONE EDGE MAY INDEX THE SAME EDGE IN THE NEXT LOWER GRAPH.  */
/*   THIS TABLE IS USED WHEN EDGE RELATED INFORMATION IN POSTED UP   */
/*   (DURING PHASE I) OR DOWN (DURING PHASE II).                     */
/* EDGE_DATA CONTAINS THE EDGE RELATED LOCAL INFORMATION:            */
/*   DB ARE THE DEFINITIONS IN THE NODE  WHICH ARE AVAILABLE ON EXIT */
/*   PB ARE THE DEFINITIONS IN THE PROGRAM WHICH ARE PRESERVED BY    */
/*   THE NODE. ON ENTRY TO DATFLOW THE DB AND PB FOR ONLY            */
/*   BASIC NODES  IS GIVEN. DATFLOW DETERMINES DB AND PB FOR THE     */
/*   NODES OF THE HIGHER ORDER GRAPHS.                               */
/* NODE_DATA CONTAINS THE NODE RELATED LOCAL INFORMATION:            */
/*   UB ARE THE USES IN THE NODE WHICH ARE UPWARDS EXPOSED, ON ENTRY */
/*   TO DATFLOW THE UB FOR BASIC NODES ARE GIVEN. DATFLOW DETERMINES */
/*   THE UB FOR THE NODES OF THE HIGHER ORDER GRAPHS.                */
/*******************************************************************/

DATFLOW:PROC(GRAPHS,EDGES_TABLE,FROM_TO_TABLE,CORRES_HEAD,CORRES_EDGE,
             EDGE_DATA,NODE_DATA,REACH,USED,LIVE);

  DCL 1 GRAPHS (NO_OF_GRAPHS) CTL,        /*FOR EACH GRAPH:            */
        2 FIRST_NODE    FIXED BIN(15),    /*INDX OF 1ST  NODE          */
        2 LAST_NODE     FIXED BIN(15),    /*INDX OF LAST NODE          */
        2 FIRST_EDGE    FIXED BIN(15),    /*INDX OF 1ST  EDGE          */
        2 LAST_EDGE     FIXED BIN(15);    /*INDX OF LAST EDGE          */

  DCL 1 EDGES_TABLE (2*NO_OF_EDGES) CTL, /*2 ENTRIES FOR EACH EDGE IN*/
        2 INDICATORS,                     /*GRAPHS.                    */
          3 IN_EDGE     BIT(1),           /*ON IF EDGE ENTERS NODE     */
          3 OUT_EDGE    BIT(1),           /*ON IF EDGE LEAVES NODE     */
          3 LATCH       BIT(1),           /*ON IF OUT EDGE AND LATCH   */
          3 HEADER      BIT(1),           /*ON IF NODE IS AN INT HEAD  */
        2 NODE          FIXED BIN(15),    /*INDX TO NODE RELATED INFO  */
        2 EDGE          FIXED BIN(15);    /*INDX TO EDGE RELATED INFO  */

  DCL 1 FROM_TO_TABLE(NO_OF_EDGES) CTL,  /*FOR EACH EDGE:             */
        2 FROM          FIXED BIN(15),    /*  THE NODE IT COMES FROM   */
        2 TO            FIXED BIN(15);    /*  THE NODE IT GOES  TO     */

  DCL CORRES_HEAD (NO_OF_NODES) CTL      /*FOR EACH NODE--THE HEAD OF*/
                      FIXED BIN(15);     /*THE INTERVAL WHICH THIS   */
                                         /*NODE REPRESENTS           */

  DCL CORRES_EDGE (NO_OF_EDGES) CTL      /*FOR EACH EDGE; EDGE IN THE*/
                      FIXED BIN(15);     /*NEXT LOWER ORDER GRAPH IT */
                                         /*CORRESPONDS TO            */

  DCL 1 EDGE_DATA (NO_OF_EDGES) CTL,
        2 DB            BIT(NO_OF_DEFS),
        2 PB            BIT(NO_OF_DEFS);

  DCL 1 NODE_DATA (NO_OF_NODES) CTL,
        2 UB            BIT(NO_OF_DEFS);

/****          DECLARATIONS OF  OUTPUT DATA                        ****/

  DCL   USED (NO_OF_NODES)  BIT(NO_OF_DEFS) CTL;
  DCL   REACH(NO_OF_NODES)  BIT(NO_OF_DEFS) CTL;
  DCL   LIVE (NO_OF_EDGES)  BIT(NO_OF_DEFS) CTL;

/****          DECLARATIONS OF LOCAL DATA                          ****/

  DCL   NO_OF_GRAPHS    FIXED BIN(15);
  DCL   NO_OF_NODES     FIXED BIN(15);
  DCL   NO_OF_EDGES     FIXED BIN(15);
  DCL   NO_OF_DEFS      FIXED BIN(15);
  DCL   HEAD            FIXED BIN(15);
  DCL   G               FIXED BIN(15);
  DCL   I               FIXED BIN(15);
```

```
/*******************   START OF PROGRAM   ***************************/
  NO_OF_GRAPHS=DIM(GRAPHS.FIRST_NODE,1);
  NO_OF_NODES=DIM(NODE_DATA.UB,1);
  NO_OF_EDGES=DIM(EDGE_DATA.DB,1);
  NO_OF_DEFS=LENGTH(DB(1));
  ALLOCATE REACH INIT((NO_OF_NODES)'0'B);
  ALLOCATE USED;
  USED=UB; /*INIT. USED TO UB FOR BASIC NODES AND TO 0 FOR ALL OTHERS*/



/**********  PHASE I - LOW ORDER GRAPH TO HIGH ORDER GRAPH ***********/
  BEGIN;
  DCL   P     (NO_OF_EDGES)  BIT(NO_OF_DEFS);     /*PRESERVED ON PATH */
  DCL   D     (NO_OF_EDGES)  BIT(NO_OF_DEFS);     /*DEFINED   ON PATH */
  DCL   PIN   (NO_OF_NODES)  BIT(NO_OF_DEFS)      /*PRESERVED ON ENTRY*/
                             INIT((NO_OF_NODES)'0'B);
  DCL   DIN   (NO_OF_NODES)  BIT(NO_OF_DEFS)      /*DEFINED   ON ENTRY*/
                             INIT((NO_OF_NODES)'0'B);
  DO G=1 TO NO_OF_GRAPHS-1;
     IF G>1
        THEN DO; /* PICK UP DATA FROM LOWER GRAPH */
           DO I=FIRST_EDGE(G) TO LAST_EDGE(G);
              PB(I)=P(CORRES_EDGE(I));
              DB(I)=REACH(CORRES_HEAD(FROM(I))) & PB(I) |
                    D(CORRES_EDGE(I));
           END;
           DO I=FIRST_NODE(G) TO LAST_NODE(G);
              UB(I)=USED(CORRES_HEAD(I));
              REACH(I)=REACH(CORRES_HEAD(I));
           END;
        END;
     HEAD=0;
     DO I=2*FIRST_EDGE(G)-1 TO 2*LAST_EDGE(G);
        IF HEADER(I)
           THEN DO;
              IF HEAD ¬= NODE(I) THEN HEAD=NODE(I);
              IF OUT_EDGE(I)
                    THEN DO;
                       P(EDGE(I))=PB(EDGE(I));
                       D(EDGE(I))=DB(EDGE(I));
                    END;
           END;
           ELSE  /* NOT A HEADER NODE */
              IF IN_EDGE(I)
                 THEN DO;
                    PIN(NODE(I))=PIN(NODE(I)) | P(EDGE(I));
                    DIN(NODE(I))=DIN(NODE(I)) | D(EDGE(I));
                    USED(HEAD)=USED(HEAD)|(P(EDGE(I)) & UB(NODE(I)));
                 END;
                 ELSE DO;  /* FOR THE OUT EDGE */
                    P(EDGE(I))=PIN(NODE(I)) & PB(EDGE(I));
                    D(EDGE(I))=DIN(NODE(I)) & PB(EDGE(I)) |DB(EDGE(I));
                 END;
        IF LATCH(I) THEN REACH(HEAD)=REACH(HEAD) | D(EDGE(I));
     END;  /* OF PROCESSING A GRAPH */
  END;  /* OF PROCESSING ALL GRAPHS BY PHASE I */
  END;  /* OF BEGIN BLOCK (LOCAL DATA IS FREED) */
/******************* THE END OF PHASE I ******************************/



/******* PHASE II - PROCESS GRAPHS FROM HIGH ORDER TO LOW ORDER ******/
  BEGIN;
   DCL   A    (NO_OF_EDGES)  BIT(NO_OF_DEFS) INIT((NO_OF_EDGES)'0'B);
   REACH(NO_OF_NODES)='0'B;  /*INITIALIZE TO DEFS THAT REACH PROGRAM */
   DO G=NO_OF_GRAPHS-1 TO 1 BY -1;
      DO I=FIRST_NODE(G+1) TO LAST_NODE(G+1);
         REACH(CORRES_HEAD(I))=REACH(CORRES_HEAD(I)) | REACH(I);
      END;
      DO I=2*FIRST_EDGE(G)-1 TO 2*LAST_EDGE(G);
         IF IN_EDGE(I) & ¬HEADER(I)
               THEN REACH(NODE(I))=REACH(NODE(I)) | A(EDGE(I));
         IF OUT_EDGE(I)
               THEN A(EDGE(I))=REACH(NODE(I)) & PB(EDGE(I)) |
                                                DB(EDGE(I));
      END; /* OF PROCESSING GRAPH */
   END;  /* OF PROCESSING ALL GRAPHS BY PHASE II */
/****************** THE END OF PHASE II ****************************/



/***************** PHASE III - LIVE ANALYSIS ***********************/
  ALLOCATE LIVE;
  USED(NO_OF_NODES)=USED(CORRES_HEAD(NO_OF_NODES));
  UB(NO_OF_NODES)=USED(CORRES_HEAD(NO_OF_NODES));
  DO G=NO_OF_GRAPHS-1 TO 1 BY -1;
     DO I=FIRST_NODE(G+1) TO LAST_NODE(G+1);
        USED(CORRES_HEAD(I))=USED(CORRES_HEAD(I)) | UB(I);
     END;
     DO I=2*LAST_EDGE(G) TO 2*FIRST_EDGE(G)-1 BY -1;
        IF OUT_EDGE(I) & ¬HEADER(I) THEN
           USED(NODE(I))=USED(NODE(I))|(USED(TO(EDGE(I)))&PB(EDGE(I)));
        IF IN_EDGE(I) THEN
           LIVE(EDGE(I))=USED(NODE(I)) & A(EDGE(I));
     END; /* OF PROCESSING GRAPH */
  END;  /* OF LIVE ANALYSIS */
  END; /* OF BEGIN BLOCK */
  RETURN;
END;
/**********************END OF DATFLOW ********************************/
```