An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language

CARLOS CHRISTENSEN

Massachusetts Computer Associates, Inc.
Wakefield, Massachusetts

I. Introduction

AMBIT/G (AMBIT for Graph Manipulation) is a language for the manipulation of *directed graphs*. The data on which an AMBIT/G program operates is displayed as an actual diagrammatic representation of a directed graph, and not some sequential encodement of a graph. Furthermore, AMBIT/G statements are themselves written as directed graphs. The programmer works in terms of these diagrams throughout the conception, composition, checking, and execution of an AMBIT/G program.

AMBIT/G arose out of two simple observations. First, a diagram with nodes and links is often used as the last resort by a programmer who is trying to understand a complicated data structure. The LISP literature [1] has made this sort of diagram familiar to many programmers, but it is effective wherever complicated structure is involved. Second, an effective method for programming symbol manipulation is the use of "pattern and replacement" rules which specify data manipulation by giving a diagram of the data before and after the required manipulation. This kind of rule is the basis for AMBIT/S (AMBIT for String Manipulation) [2, 3] as well as several other languages for string manipulation.

AMBIT/G combines the use of directed graph diagrams for data with the use of pattern-replacement rules for program statements to produce a simple language for symbol manipulation. Because diagrams rather than character

strings are used for data and program, the input/output requirements are unusual. The full implementation of AMBIT/G represents a challenging future application of an interactive system for computer graphics.[1]

AMBIT/G has been implemented in a preliminary form by Peter Moskovites. The implementation is on the SDS 940 time-sharing system at Harvard University. The system uses teletype input/output and therefore requires translation by the user (instead of the system) between directed graph diagrams and teletype character strings.

II. The Data Graph

An AMBIT/G program operates on a single data object, the *data graph.* An example of a data graph is the diagram given in Fig. 7. This diagram is, as it happens, the output of the example AMBIT/G program discussed later in this paper; however, it is representative of the data graph as it might appear at any point during the execution of a program.

The data graph is a collection of *nodes* which are displayed on a two-dimensional medium called the *data field.* Each node consists of a *boundary*, a *name*, and a collection of any number (possibly zero) of *links*. A node boundary is a closed geometric figure, such as a rectangle or a circle. A node name is a character sequence written inside the node boundary, such as "ALPHA" or "3769." A link is a "flexible" line segment which is attached to the node boundary at one end and which has an arrowhead at the other end. The point on the node boundary at which a link is attached is the *origin* of the link.

The nodes used in a particular data graph are classified into *types*. Two nodes are of the same type if their boundaries are congruent without rotation. If two nodes are of the same type, then they must have link origins in corresponding positions on their boundaries.

When the data graph is created, each link of each node is arranged so that it *points to* some node, that is, the arrowhead end of the link is placed on an arbitrary point on the boundary of a node. The node pointed to may be, as a special case, the node from which the link originates. Note that the point of origin of a link distinguishes that link from other links, but that its point of attachment to the node pointed to has no significance.

[1] *Author's Note:* A second and fully graphic implementation of AMBIT/G was completed in May of 1968, after this paper had been submitted. This implementation was constructed for the TX-2 computer at the Lincoln Laboratories by Martha Greenberg, Austin Henderson, and Paul Rovner. A Sylvania tablet is used for the input of diagrams and a CRT screen for output. The program given in this paper was input and executed as a test case, and the diagrams drawn as computer input differered from those in this paper chiefly in aesthetic details.

Two restrictions are placed on the construction of the data graph, as follows:

R1. No two instances of the same type of node may have the same node name; and,

R2. No two links of the same node may have the same point of origin on the node boundary.

These restrictions, together with the obvious restriction that a link can point to only one node at a time, provide for the specification of unique access paths through a data graph.

Once a data graph has been created, the only significant change which can be made in the graph is a change in a link; that is, the link may be redrawn so that the arrowhead end points to a different node than before. A node cannot be added to or deleted from a graph; the origin of a link cannot be changed; and, the name of a node cannot be changed. On the other hand, certain changes which do not affect the interpretation of the graph may be made, such as the translation without rotation of a node on the data field.

The data graph must be drawn in a way which avoids certain obvious ambiguities. A node name must be written horizontally. A node boundary must lie entirely outside of all other node boundaries. Each link must lie outside of all node boundaries except at its end points and must not be tangent to any other link. Aside from these rules, the node names, boundaries, and links may be laid out on the data field in any convenient way.

The data graph just defined has been designed so that it can be represented and accessed efficiently in the immediate-access memory of a conventional computer. Although the design of the data graph will be extended and refined in the future, only a radical change in the design of computer hardware will produce a radical change in the design of the data graph.

III. The Example Program

In this section, a complete AMBIT/G program and its output are given. It represents an algorithm which is used for "garbage collection" in some implementations of LISP. It is not a trivial program, and it assumes some familiarity with LISP [1]. Thus we have chosen, as an example, a program from LISP implementation; but we strongly emphasize that AMBIT/G is in no way restricted to or specialized for LISP structures or LISP programming.

The program and its output are given in seven figures. Each figure has diagrams written in the AMBIT/G language; beneath each diagram is an alphanumeric encodement of the diagrams appropriate to teletype communication with a computer. This encodement is necessary because AMBIT/

G has not yet been implemented with true graphic I/O; and, it is slightly more complicated than necessary because the present implementation of AMBIT/G stands in need of revision.

Part A: Creation of Types

For any class of nodes under consideration, there exists a *type* and any number of *tokens*. The type is an abstract description of all the nodes in the class while a token is a specific and concrete instance of a node which is a member of the class. The program begins with a *TYPES statement* (Fig. 1) which creates a type for each distinct class of nodes which will be used in the data graph.

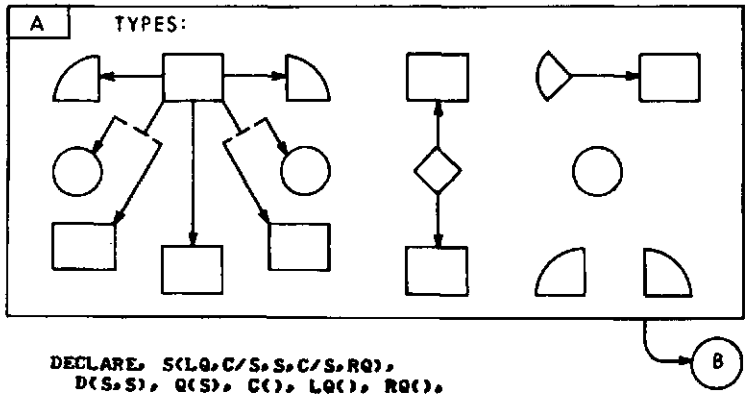


Fig. 1. Creation of types.

In the following paragraphs, the essential information given in Statement A is expressed in English. In order to do so, English names are given to the node boundaries and links which appear in the diagram. These names are not, themselves, a part of the information conveyed by the diagram; rather, they are chosen for convenience of discourse. Thus, for example, a rectangular node boundary of proportions 2 to 3 is referred to as a "square."

In this particular program, the height of all of the node types is the same. There are six node types, as follows:

1. A square node (S) has a boundary which is a 2 by 3 rectangle with its long edge horizontal. The node has five links, as follows:
   a. the *left link* originates at the middle of the left side and must point to a left-quadrant node;
   b. the *left-son link* originates at the lower left corner and must point to a circle or a square node;

c. the *down link* originates at the middle of the base and must point to a square node;
d. the *right-son link* originates at the lower right corner and must point to a circle or a square node; and,
e. the *right link* originates at the middle of the right side and must point to a right-quadrant node.

2. A *diamond node* (D) has a boundary which is a square with a side at 45° to the horizontal. The node has two links, as follows:
a. the *up link* originates at the upper corner and must point to a square node; and,
b. the *down link* originates at the lower corner and must point to a square node.

3. A *quadrant node* (Q) has a boundary which is the arc of a circle between 135° and 225° and the radii at the ends of this arc. The node has one link, as follows:
a. the *link* originates at the right corner and must point to a square node.

4. A *left-quadrant node* (LQ) has a boundary which is the boundary of the second quadrant of a circle. The node has no links.

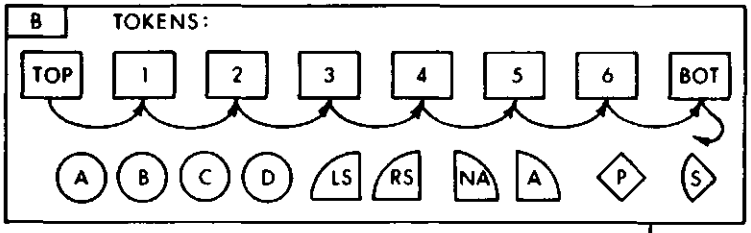
5. A *right-quadrant node* (RQ) has a boundary which is the boundary of the first quadrant of a circle. The node has no links.

6. A *circle node* (C) has a boundary which is a circle. The node has no links.

The programmer has certain uses in mind for the nodes and links declared in the TYPES statement. Most important, he wishes to represent a tree. The forks of this (bifurcate) tree will be the squares; the branches will be the left- and right-son links which emerge from a square; and the leaves will be the circles. For the purpose of garbage collection, he wishes to have the squares organized in a single sequence (at the same time they are being used in the tree), and the down link of the squares will be used for this purpose. In addition, he needs certain pointers to keep track of scans and walks through the data field, and the links of the diamond and quadrant nodes will be used for this purpose. Finally, he requires certain temporary indicators for the algorithm, and the left and right links of the squares will be used for these indicators.

Part B: Creation of Tokens

The program continues with a *TOKENS statement* (Fig. 2) which creates all of the instances of nodes which will be required in the data graph and which sets the constant links of the data graph.




```
    ALLOCATE. S(8), C(4), LQ(2), RQ(2), D(1), Q(1),
B:  C.S/C,
    R.S.TOP(*,*,S.1), S.1(*,*,S.2), S.2(*,*,S.3),
      S.3(*,*,S.4), S.4(*,*,S.5), S.5(*,*,S.6),
      S.6(*,*,S.BOT), S.BOT(), C.A(), C.B(), C.C(), C.D(),
      LQ.LS(), LQ.RS(), RQ.NA(), RQ.A(), D.P(), Q.S().
```

FIG. 2. Creation of tokens.

In Statement B the data graph on which the program will execute is established. The required instances of each type of node are created and placed in the data field; each instance of a node is given a name which is unique within its type; and those links which are explicitly shown are interpreted to be *constant* links and are permanently set. All of these characteristics of the data graph remain until program execution is complete. On the other hand, the links which are specified by the TYPES statement but not shown in the TOKENS statement are the variable links; they are created with the nodes of which they are a part, but their setting is left undefined.

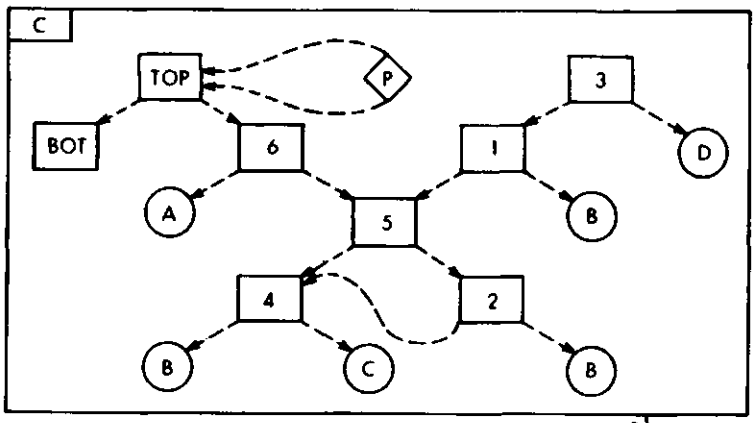
PART C: INITIALIZATION

In this part of the program (Fig. 3), some of the variable links of the program are set to initial values. This initialization establishes a special case of the data on which the program is intended to operate; as such, it may be thought of as input to the program.

Statement C is a *modification rule*. It specifies (by means of broken lines) the setting of some of the previously undefined links. All of the remaining statements of the program are modification rules, and a discussion of the interpretation of modification rules is deferred until Part D, where an appropriate example is available.

Note that three instances of the Circle B node appear in Statement C. All three of these instances represent the same unique instance of Circle B in the data graph. Circle B is thus pointed to by three links after the execution of this rule.

The initialization establishes a particular configuration of square nodes for which garbage collection would be required. Free squares, if there were any, would be chained by means of left-son links between TOP and BOT; but




```
C:  C.S/D1, D.P(), S.TOP(),
      S.6(), S.5(), S.4(),
      S.3(), S.2(), S.1();
    R.D.P(S.TOP,S.TOP), S.TOP(*,S.BOT,*,S.6),
      S.6(*,C.A,*,S.5), S.5(*,S.4,*,S.2), S.4(*,C.B,*,C.C),
      S.3(*,S.1,*,C.D), S.2(*,S.4,*,C.B), S.1(*,S.5,*,C.B),
```

FIG. 3. Initialization.

since the left son of TOP is BOT, the free list is empty. The right son of TOP is the root of the tree which exists at this time. There are two square nodes, Square 1 and Square 3, which are not part of this tree; that is, they cannot be accessed by any sequence of son links from the root. These two squares are the *garbage*, and it is the object of this program to enter such square nodes into the free list.

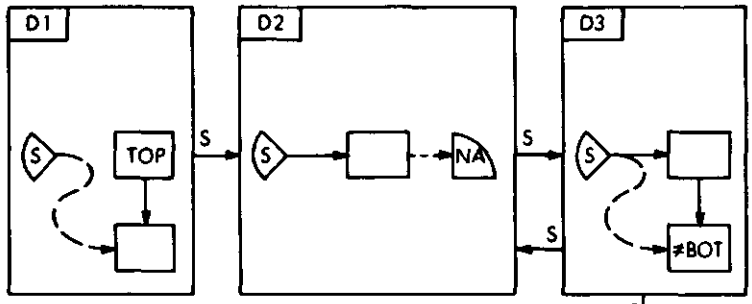
Note that the tree established here is reentrant; that is, Square 4 is the left son of both Square 5 and Square 2. The program given here is designed for any form of reentrant tree, including those which have circular linkage (for example, a node which has its grandfather as a son).

PART D: MARK ALL SQUARES "NOT ACCESSIBLE"

This part of the program (Fig. 4) scans through the square nodes, selecting each square (except TOP and BOT) exactly once. Each square thus selected is marked "not accessible." This marking refers to accessibility from the root of the tree by means of son links and is performed tentatively, subject to correction in Part E.

Consider, for a moment, the modification rule D3. The step-by-step interpretation of this rule is as follows:

1. Find the node pointed to by the link of Quadrant S and call this node *Dummy 1*.




```
D1: C.S/D2, S.TOP(*,*,S*D1), Q.S(),
    R.S.TOP(), Q.S(S*D1),
D2: C.S/D3, Q.S(S*D1), S*D1(),
    R.Q.S(), S*D1(*,*,*,*,RQ.NA),
D3: C.S/D2,F/E1, Q.S(S*D1), S*D1(*,*,~S.BOT), S*D1(*,*,S*D2),
    R.Q.S(S*D2), S*D1(),
```

FIG. 4. Mark all squares "not accessible."

2. If *Dummy 1* is not a square node, then go to E1 (since the rule has failed). However you can skip this step because the TYPES statement specified that the link of a quadrant must point to a square.

3. Find the node pointed to by the down link of *Dummy 1* and call this *Dummy 2*.

4. If *Dummy 2* is not a square node, then go to E1. However, you can skip this step because the TYPES statement specified that the down link of a square must point to a square.

5. If *Dummy 2* is named BOT, then go to E1; otherwise, continue.

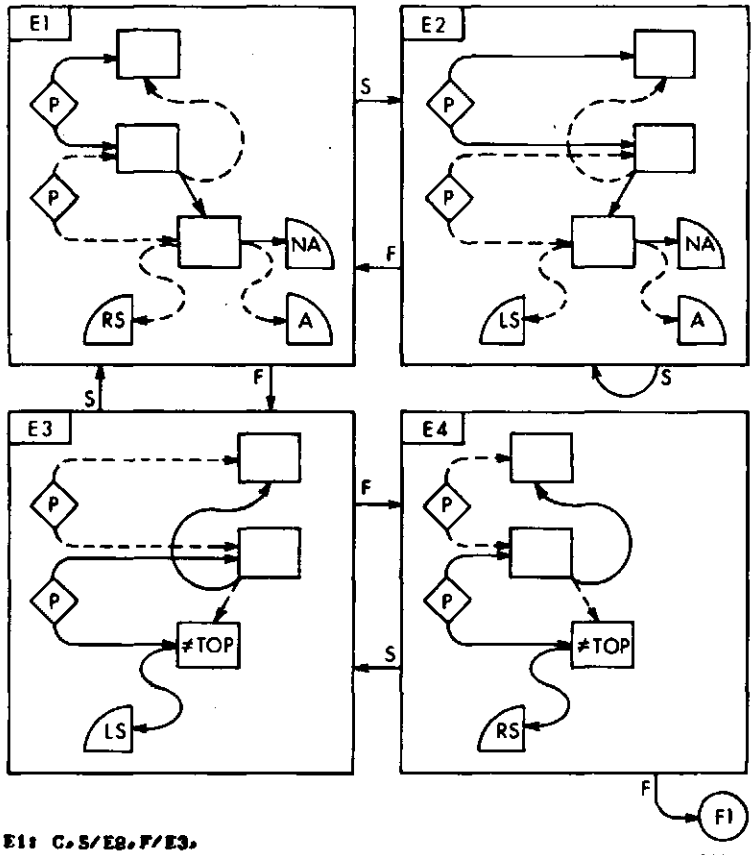
6. You have successfully matched the rule to the data and the rule is bound to succeed. Now make the required modification of the data by setting the link of Quadrant S to point to *Dummy 2*. Go to D2.

Note that solid links represent links which must be found if the rule is to succeed and broken links represent links which are to be set (as solid links in the data) if the rule does succeed.

The scan is effected by means of the down links which, according to the TOKENS statement, have been permanently set to arrange the squares in a single sequence. The first modification rule, D1, sets the link of Quadrant S to point to the first square after TOP in the sequence defined by the down links. The second rule, D2, sets the right link of the square pointed to by the link of Quadrant S to point at Right-Quadrant NA. This marks the square "not accessible." The third rule, D2, advances the link of Quadrant S to by the next square in the sequence unless that square is BOT. Thus Part D executes D1 once to initialize the scan, and then loops between D2 and D3 until the scan has been completed. At that time, exit to Part E occurs.

PART E: MARK ACCESSIBLE SQUARES "ACCESSIBLE"

This part of the program (Fig. 5) begins at the root of the tree (the right son of TOP) and then walks the tree, selecting every square which is accessible from the root square by way of left- and right-son links. Each square thus selected is marked "accessible," thereby overriding the erroneous setting established in Part D.




```
E1: C.S/E2.F/E3.
      D.P(S*D1.S*D2). S*D2(*.*.*.S*D3). S*D3(*.*.*.*.RQ.NA).
    R.D.P(S*D2.S*D3).S*D2(*.*.*.S*D1).S*D3(LQ.RS.*.*.*.RQ.A).
E2: C.S/E2.F/E1.
      D.P(S*D1.S*D2). S*D2(*.S*D3). S*D3(*.*.*.*.RQ.NA).
    R.D.P(S*D2.S*D3). S*D2(*.S*D1). S*D3(LQ.LS.*.*.*.RQ.A).
E3: C.S/E1.F/E4. D.P(*.-S.TOP).
      D.P(S*D2.S*D3). S*D2(*.S*D1). S*D3(LQ.LS).
    R.D.P(S*D1.S*D2). S*D2(*.S*D3). S*D3().
E4: C.S/E3.F/F1. D.P(*.-S.TOP).
      D.P(S*D2.S*D3). S*D2(*.*.*.S*D1). S*D3(LQ.RS).
    R.D.P(S*D1.S*D2). S*D2(*.*.*.S*D3). S*D3().
```

FIG. 5. Mark accessible squares "accessible."

A tree walk is a bit more complicated than a sequential scan. When a particular square is selected, it is not possible to walk to both of its sons "simultaneously." Rather, the walk must proceed to one of the sons (say the left son) and somehow must provide a means to return later to walk to the other son (the right son).

This process is sometimes organized around a push-down stack which is used to record those sons for whom selection has been deferred. However, this pushdown stack requires an amount of memory which depends on the size and shape of the tree being walked. Thus the use of a pushdown stack is not appropriate for a garbage collection algorithm which is, after all, invoked because available memory has been exhausted.

The method used in Part E is a different one. It is based on a backtracking technique which was invented by Peter Deutsch. As the walk moves down the tree, links are bent backward so that a link which normally points to the son of a square is caused to point to the father of that square. Thus it is possible to walk down the tree until a circle is encountered, back up to a new downward path, walk down that path, and so on until the entire tree has been walked.

The up and down links of Diamond P are used to control the walk. It is convenient to refer to the two squares pointed to by these links as the *selected father* and the *selected son*. Since TOP does not have a father, it is assumed to be its own father, and the algorithm begins and ends with TOP as both the selected son and selected father. After each step of the walk the links which are bent back are exactly those which would normally trace the line of descent from TOP to the current selected son. Two links are required to control the walk because the tree structure is always broken between the selected father and the selected son.

The four modification rules of Part E represent the four steps used in the tree walk: father to right son (E1), father to left son (E2), left son to father (E3), and right son to father (E4). The father-to-son steps must record whether the new selected son is a right son (RS) or a left son (LS) by setting the left link of the square. This is necessary because this information is otherwise lost when the son link is bent back by the execution of the step.

The father-to-son steps each fail under either of two conditions. First, if the new selected son would be a circle, then the step is not taken because a leaf of the tree has been reached. Second, if the new selected son would be one which is marked "accessible," then the step is not taken because that son is a root of a subtree which has already been walked by way of some reentrant link.

The son-to-father steps each fail under either of two conditions. First, if the selected son is not the correct son (left for E3, right for E4), then the step is not be taken. Second, if the selected son is TOP, then the step is not be taken because the walk is complete.

For the reader who wishes to experiment with a desk execution of Part E, the following trace is included:

E1, S; E2, F; E1, S; E2, S; E2, F; E1, F; E3, S; E1, S; E2, F;
E1, F; E3, F; E4, S; E3, F; E4, S; E3, F; E4, S; E3, F; E4, F.

This trace is read as "modification rule E1 is attempted and succeeds; modification rule E2 is attempted and fails; ...."

Part F: Place Squares Marked "Not Accessible" on Free List

This part of the program (Fig. 6) scans through the square nodes, selecting each square (except TOP and BOT) once. Each square thus selected is linked into the top of the free list if and only if it is marked "not accessible."

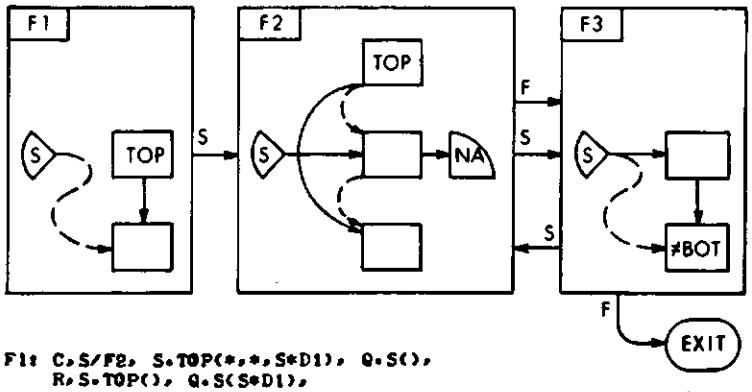


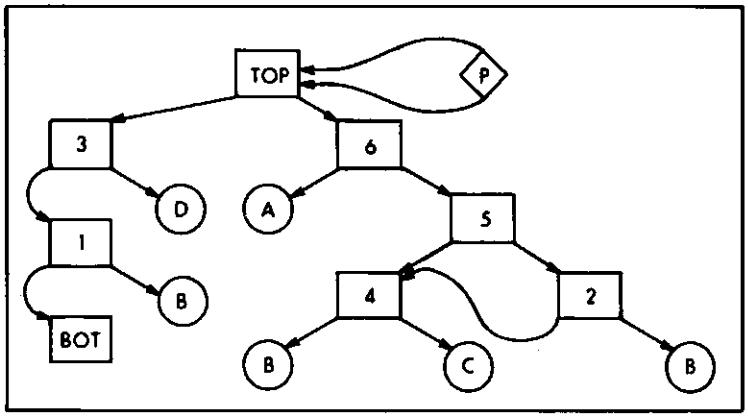
FIG. 6. Place squares marked "not accessible" on free list.

The scan is effected in the same way as the scan of Part D. In F2, a test is made of the selected square to see whether its right link points to Right-Quadrant NA. If this is the case, the square is entered as the top square on the current free list. Note that the free list is linked together by means of left-son links. The left son of TOP is the first square on the free list, and the square whose left son is BOT is the last square on the free list.

When the scan is completed, garbage collection is complete and exit from the program occurs. The program presented here would, in practice, be a subroutine of a larger program such as a LISP interpreter. It would be called when the free list was exhausted, and it would return with the free list replenished.

THE OUTPUT

The 13 boxes given in Figs. 1–6 constitute a complete input to the AMBIT/G interpreter. As we have seen, this input includes the description and creation of the data graph, the initialization to a particular data configuration, and a collection of modification rules which operate on the data field. The single box given in Fig. 7 is the corresponding output of the AMBIT/G interpreter; it represents the data field after program execution has been completed.




```
S.TOP(*,S.3,S.1,S.6,*),
S.1(*,S.BOT,S.2,C.B,RQ.NA),
S.2(LQ.RS,S.4,S.3,C.B,RQ.A),
S.3(*,S.1,S.4,C.D,RQ.NA),
S.4(LQ.LS,C.B,S.5,C.C,RQ.A),
S.5(LQ.RS,S.4,S.6,S.2,RQ.A),
S.6(LQ.RS,C.A,S.BOT,S.5,RQ.A),
S.BOT(*,*,*,*,*),
C.A(), C.B(), C.C(), C.D(),
LQ.LS(), LQ.RS(), RQ.NA(), RQ.A(),
D.P(S.TOP,S.TOP), Q.S(S.6),
```

FIG. 7. The output.

We have taken some liberties in drawing the data graph of the output; that is, links and nodes which are not of interest have been deleted from the diagram. When the program was executed by the current implementation of AMBIT/G, it produced the teletype output which is shown below the box. In this form, the nodes and links of the data graph are shown without deletions.

IV. Applications of AMBIT/G

It is intended that AMBIT/G be applied both to abstract mathematical problems in the manipulation of directed graphs and to practical programming problems in the implementation of computer software.

On the theoretical side, AMBIT/G is currently being used at Harvard University by T. E. Cheatham, Jr. for teaching a wide variety of algorithms for syntactic analysis. Since a single rule of a BNF grammar can be represented as an n-ary fork and a complete analysis can be represented as a tree composed of these forks, AMBIT/G is an appropriate language for the representation of these algorithms.

On the practical side, we are currently experimenting with writing an incremental compiler for a conventional language in AMBIT/G. The present implementation of AMBIT/G has considerable efficiency in its storage of structured data in memory. Further developments are planned to incorporate into AMBIT/G, in a graph-oriented and machine-independent way, some of the tricks systems programmers use to optimize their programs.

AMBIT/G is related to systems for graphical displays in two ways. First, AMBIT/G should ultimately be implemented in a graphical display system which will permit the programmer to interactively compose and execute the diagrams which constitute an AMBIT/G program. Second, AMBIT/G is appropriate for the implementation of graphical display systems since these systems make use of directed graphs for the internal analytical representation of the lines and surfaces which are displayed. Thus we hope to implement AMBIT/G on a system which is itself implemented in AMBIT/G.

REFERENCES

1. McCarthy, J., Recursive Functions of Symbolic Expressions and their Computation by Machine, I, *Comm. ACM* 3 184–195 (1960).
2. Christensen, C., Examples of Symbol Manipulation in the AMBIT Programming Language, *Proc. ACM Natl. Conf., 20th, Cleveland, Ohio, August 1965*, pp. 247–261.
3. Christensen, C., On the Implementation of AMBIT, a Language for Symbol Manipulation, *Comm. ACM* 9, 570–573 (1966).