AD 744700

DDC
RECEIVED
JUL 10 1972
B

APPLIED DATA RESEARCH, INC.

259

1

APPLIED DATA RESEARCH, INC.

LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

SEMI-ANNUAL TECHNICAL REPORT
(1 October 1971 - 31 March 1972)

FOR THE PROJECT
AMBIT/L DOCUMENTATION

Principal Investigator:	Anatol W. Holt (617) 245-9540
Project Scientist:	Anatol W. Holt (617) 245-9540

ARPA Order Number - ARPA 1228
Program Code Number - 8D30

Contractor:	Mass. Computer Associates, Subsidiary of Applied Data Res.
Contract No.:	DAHC04-68-C-0043
Effective Date:	21 June 1968
Expiration Date:	30 September 1972
Amount:	$891,975.00

Sponsored by
Advanced Research Projects Agency
ARPA Order Number - ARPA 1228

APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

AMBIT/L Programming System

Users' Guide

by

Michael S. Wolfberg

CA-7201-1711

January 17, 1972

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by U. S. [illegible] Research Office - Durham, Box CM, Duke Station, Durham, North Carolina 27706, under Contract DAHC04 68 C 0043.

SUMMARY

This User's Guide is a collection of separately-written sections which describe how to use the AMBIT/L Programming System as implemented on the DEC PDP-10 under the DECsystem-10 time-shared operating system. This document supplements the basic reference manual which is a paper by Carlos Christensen, the creator of AMBIT/L, entitled "An Introduction to AMBIT/L, A Diagrammatic Language for List Processing".*

The sections in this collection are arranged in an order suggested for initial reading. The title sheet for each section includes the date corresponding to its most recent revision. Each page is identified by its section letter and page number within that section (e.g., C-4).

In the initial release of this document one section which is in preparation is not included - "Using DAMBIT/L: the AMBIT/L Debugging System". Any questions concerning this part of the system should be directed to the author.

*A full bibliographic reference to this paper is given in Section A.

CONTENTS

Section A

An Introduction to AMBIT/L

January 14, 1972

This section should be read by users of the AMBIT/L Programming System as a supplement to the paper which serves as a reference manual, "An Introduction to AMBIT/L, A Diagrammatic Language for List Processing".

The paper by Carlos Christensen entitled "An Introduction to AMBIT/L, A Diagrammatic Language for List Processing" is the current reference manual for the AMBIT/L Language.* It was written to be a self-contained tutorial introduction, however, and, therefore, the author took the liberties of occassionally being imprecise and incomplete. Until the paper is rewritten, this memo should be used as a supplement to the paper by those learning AMBIT/L so they can use it as a programming language within the PDP - 10 implementation of the AMBIT/L Programming System. The slight discrepancies in detail between the paper and the characteristics of the implemented language may be overlooked by the reader simply interested in understanding the essence of AMBIT/L.

The following notes are ordered according to page numbers in the paper.

Page 8: Concerning mark nodes, each one is declared in some program block by the programmer. AMBIT/L programs have a block structure which scopes mark nodes exactly as ALGOL 60 block structure scopes own variables.

Page 8: Concerning basic symbol nodes, the AMBIT/L Compiler and DAMBIT/L Debugging System currently acknowledge the following names for non-printing characters:

*Christensen, Carlos. "An Introduction to AMBIT/L, A Diagrammatic Language for List Processing", Proceedings of SYMSAM/2, the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March 1971, available as CA-7102-2211 (February 22, 1971) from Applied Data Research, Inc., Wakefield, Massachusetts 01880.

Subname	Representing ASCII	Octal Code
%CR	CR	015
%LF	LF	012
%VT	VT	013
%FF	FF	014
%TAB	HT	011
%ESC	ESC (or ALT)	033
%SUB	SUB (or ↑Z)	032

Note that the character SPACE is represented by the basic symbol node whose subname is the character '%' followed by the character SPACE.

Page 9: Concerning integer nodes, although there is a unique canonical subname for each integer node, both the AMBIT/L Compiler and DAMBIT/L Debugger accept +13 and +013 as allowable input syntax. The programmer can consider that some macro converts such non-canonical forms into the canonical one.

Page 9: Concerning real nodes, although there is a unique canonical subname for each real node, both the AMBIT/L Compiler and DAMBIT/L Debugger accept several non-canonical forms as allowable input syntax. For example:

```
  -34     34E5
  .34     .34E -5
+ 34.     34.E56
   3.4    3.4E+ 5
```

Page 9: Concerning pointer nodes, each one is declared in some program block by the programmer in one of two ways: as a temporary pointer (as an ALGOL 60 local variable) or as a permanent pointer (as an ALGOL 60 own variable).

Page 10: Concerning token nodes, the only nodes which cannot contribute to the subname of a token are pointers and cells other than the null cell.

Page 11: There has been no conscious attempt at showing the canonical subnames of real nodes.

Pages 1[illegible]8: The current implementation permits the programmer to suffix an exclamation point (!) to a type-set. This has the meaning that if the pattern match fails due to an incompatibility between a node in the data and the specified type-set, then rather than causing failure of the rule, a normal interpreter will produce an error condition which either traps or aborts execution; if a production" interpreter were ever used, it would ignore the test entirely.

Page 18: On the seventh line of text, the "P" should be eliminated from the type-set, thus reading "type-set CST is implied."

Page 21: In the second step for adding a function link the word "if" should have been "is". However, the entire step should be changed to be:

> "2. Make the lower side of the node boundary into a double line; the result is a <u>call node</u>."

Also, the third and fourth steps should be changed to be:

> "3. From distinct points along the top and/or bottom side, draw <u>m</u> ordinary links to appropriate destinations. Consider these links ordered by their origins (from left to right), and call their destinations the <u>origins</u> [sic] of the function link.

4. From distinct points along the left and/or right side, draw n ordinary links to appropriate destinations. Consider the links ordered by their points of origin (from top to bottom) and call their destinations the destinations of the function link."

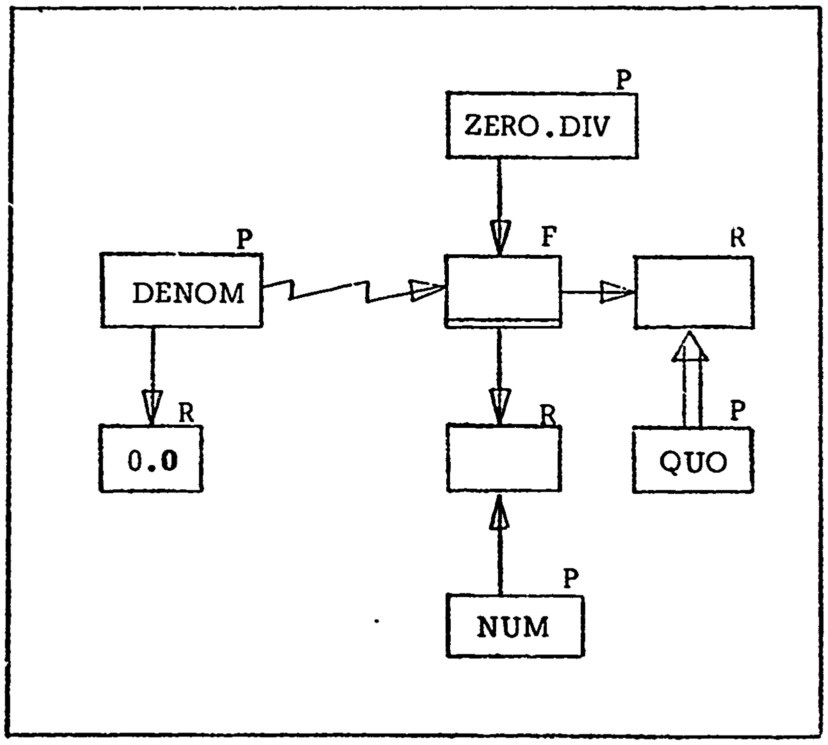
Also note that pointer nodes may not be either origins or destinations of a call node.

Page 22: The example in RULE E8 includes a function link with the subname "DIVIDE". Although the built-in function ADD may be applied as shown in that rule, if the function to perform the division is also meant to be the appropriate built-in one, then its name should be "DVQ", for "divide yielding quotient".

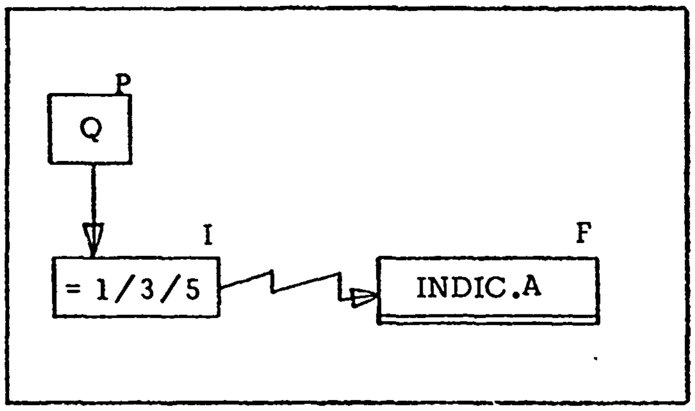
Page 23: Although the example in rule E9 may serve a tutorial purpose, it is ill-formed according to the current implementation, since an argument of a function call may not be a pointer node. There could be a more restricted definition of the function DOWN which reads the down link of a string, token, or cell.

Page 24: Note that the comment that the rule E9 is ill-formed (on page 23) also applies to rule E10. Although the paper avoids a discussion of flow links among ordinary nodes, this topic will be covered here. The programmer should understand that when a rule is interpreted nodes in the rule are matched with nodes in the data essentially one at a time. When the interpreter directs its attention at a node, i.e., "visits" that node, it follows links originating from that node (if any) to their destinations. If a node at the destination is either a fully-named node or one with just a type-set, then these destinations are tested and if the match continues to succeed we say that the original node has been visited. The notion of visiting also applies to a call node: when all of its origins have been matched then a call node may be visited (also see page 28). This process begins by performing the specified function call and then all destination links from that call node are followed as described for an ordinary node. After all destinations of the call node have been matched we say that the call node has been visited.

With these definitions made, it can be stated that a flow link between any two nodes in a rule causes the node at the tail of the flow link to be visited before the node at the head of the flow link. Thus rule E10 can be correctly redrawn as:



There is an additional confusion when using flow links associated with nodes whose names cause some macro to be invoked such as the use of the name set notation as shown in the upper diagram on page 38. If a flow link had emanated from the pointer Q (see page 38) then it would not be ordering very much; since the expansion occurs as it does, such a flow link would mean that only the type of the node pointed to by Q is tested when Q is visited. In such a construction it is more likely that such a flow link should emanate from the node employing the name set notation; this would cause the membership function to be called before the node at the head of that flow link is visited. The following rule shows such a use of a flow link as an example:



Note from the comments associated with page 38 that literal token nodes and string nodes cause macro expansions.

Pages 30-34: Some built-in functions are briefly mentioned here. A much more complete description of these and others is found in the AMBIT/L User Memo entitled "AMBIT/L Built-in Functions for the Programmer".

Page 30: The fourth line of the second paragraph under "Arithmetic" should begin with "base 2^{34}."

Page 31: The second and third paragraphs describe trap functions. Input/output traps are handled this way, but traps for arithmetic functions have not yet been implemented. Thus the example presented using P/ZERO.DIV is at present hypothetical.

Page 33: The function GET.CELL referenced in the section on "Cell Management" is used as a tutorial means for describing these concepts. There is no such function upon which the programmer may call; instead, the programmer may employ the new cell notation and the gather arguments notation described on pages 35 - 37.

In the second paragraph under "Cell Management" the first word on the fifth line should be "rendered". In that paragraph the description of the trap function being called for a garbage collection choke is wrong; instead the system pointer GCOL.CHOKE is expected to point a label node corresponding to somewhere in the program where there is a recovery routine, and thus the interpreter performs a transfer of control via this pointer when a choke occurs. Such a recovery routine should be placed high enough in the block structure so that a transfer to it will tend to release cells which are accessible only via local (TEMP) pointers.

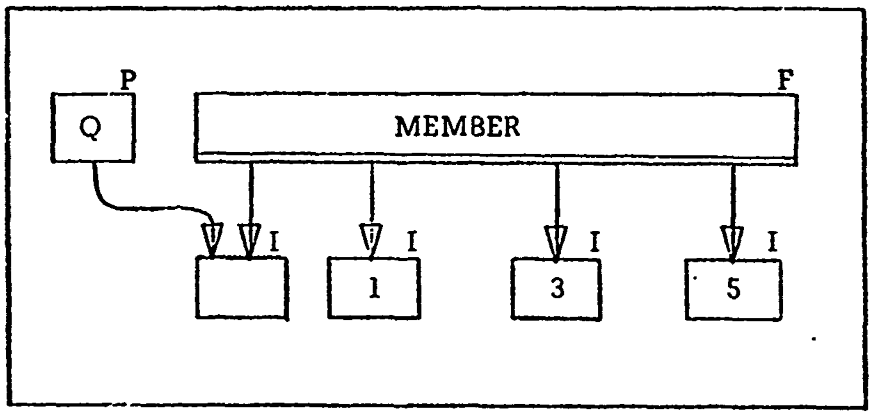
Page 34: The structure input/output routines mentioned in the last paragraph do exist and are being employed in the IAM program. However, these functions have not been documented and are thus considered unsupported. If any interest (or requirement) develops, an attempt will be made to support them.

Pages 35-39: The description of the built-in macros is not complete; the following supplements the paper by presenting additional macros. Also, this section makes use of the GET.CELL function as a tutorial aid. The reader should recall there is no such function upon which the programmer may call.

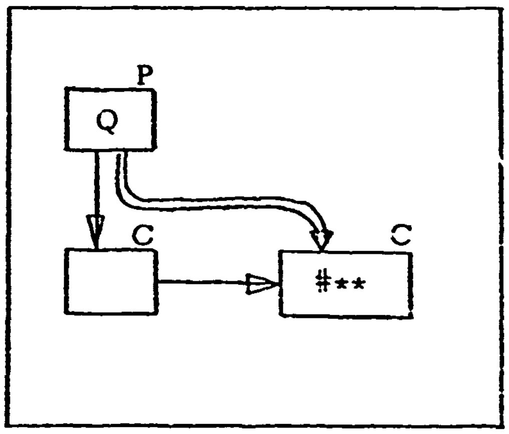
Page 36: Use of the "new cell" notation always yields a cell whose right link and down link both point to the null cell. Thus the right modification link on the rightmost cell in both of the diagrams on page 36 is redundant. Incidentally, including them in the rule does not affect the amount of work to be done by the interpreter when the rule is executed.

Page 37: In describing the Gather Arguments Notation the GATHER function has been employed as a tutorial aid. There is no such function upon which the programmer may call. The example rule of this page includes a call node with a subname IS.MEMBER. Note that this is not a built-in function of the AMBIT/L Programming System; there is, however, a MEMBER function which is essentially the same predicate, but it accepts an arbitrary number of arguments (as illustrated in following comments). If a cell in a rule is used to gather argument(s) then it must be the destination of exactly one solid link (including the one coming from a call node); such a cell may be the destination of any number of modification links.

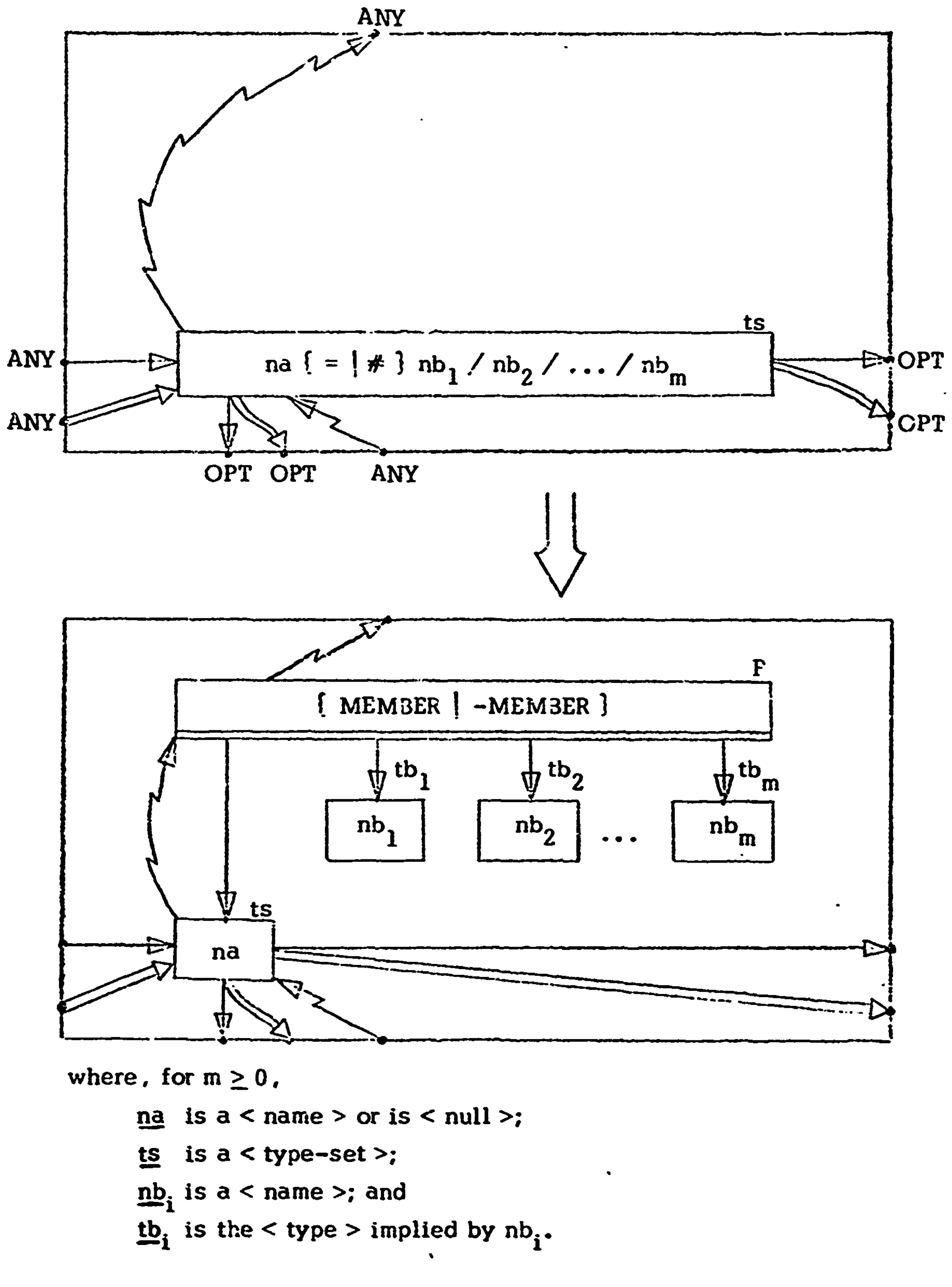
Page 38: The current implementation of Name Set Notation employs the MEMBER function, rather than the IS.MEMBER function as shown. MEMBER accepts all arguments directly and is the only AMBIT/L function which accepts an arbitrary number of arguments. Thus the lower diagram on page 38 should be as follows:



The formal way of expressing these macros referenced at the bottom of page 38 will be introduced here with minimal explanation since the intent should be obvious. First the name set notation macro will be presented. Note that by using the number sign (#) rather than the equal sign (=), the programmer may assert that a node is not contained in a specified set. A common use of this notation in AMBIT/L programs is the advancement of a pointer along a list unless the end is reached; this is shown in the following rule:



The diagram on the page after next represents the general transformation performed by the name set notation macro. The "ANY" labels of the upper diagram are associated with those links where any number (including zero) of such links may be present. The "OPT" labels, for optional, are associated with those links which optionally may be present (i.e., either zero or one occurrence). The pair of diagrams demonstrates how such links are transformed. The curly braces in the upper diagram indicate that there may be either an equal sign or number sign; the vertical bar separates these two characters. In macro descriptions which follow such a choice is shown by vertical positioning within braces. The curly braces in the lower diagram are used to be in direct correspondence with the ones in the upper one. Namely, if a number sign had been in the original node then the expansion would use "-MEMBER".



where, for $m \geq 0$,

<u>na</u> is a < name > or is < null >;

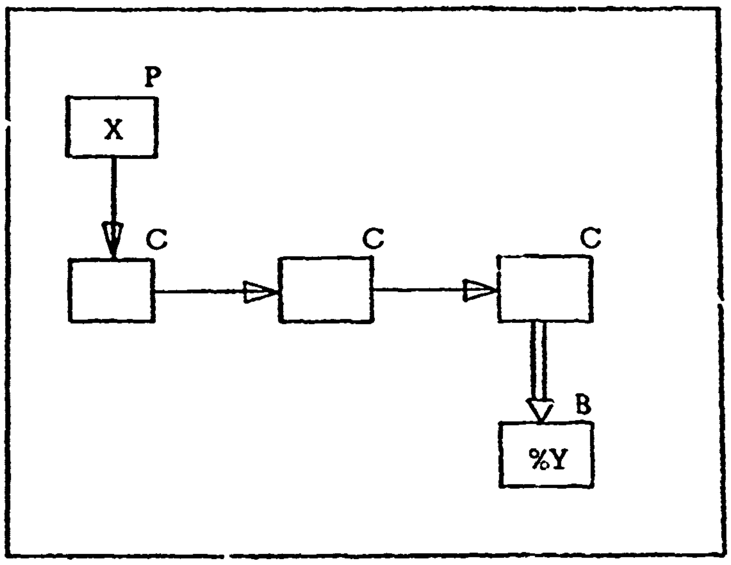
<u>ts</u> is a < type-set >;

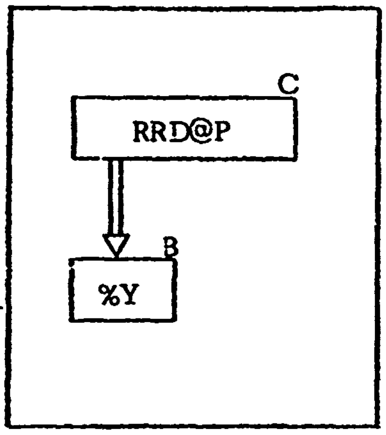
<u>nb_i</u> is a < name >; and

<u>tb_i</u> is the < type > implied by nb_i.

The non-terminal names within angle brackets ('<' and '>') are defined in the complete syntax presented in the AMBIT/L User Memo "The Syntax of the Encodement of AMBIT/L Programs". For example, one production of the grammar indicates that a <name> is either a <dummy>, or an <indirect>, or a <literal>.

Indirection: There is a notation by which a node may be named by a path via which it can be reached from a pointer. For example, the following two rules are equivalent:





This notation is explained in several steps as follows:

a. Default Link

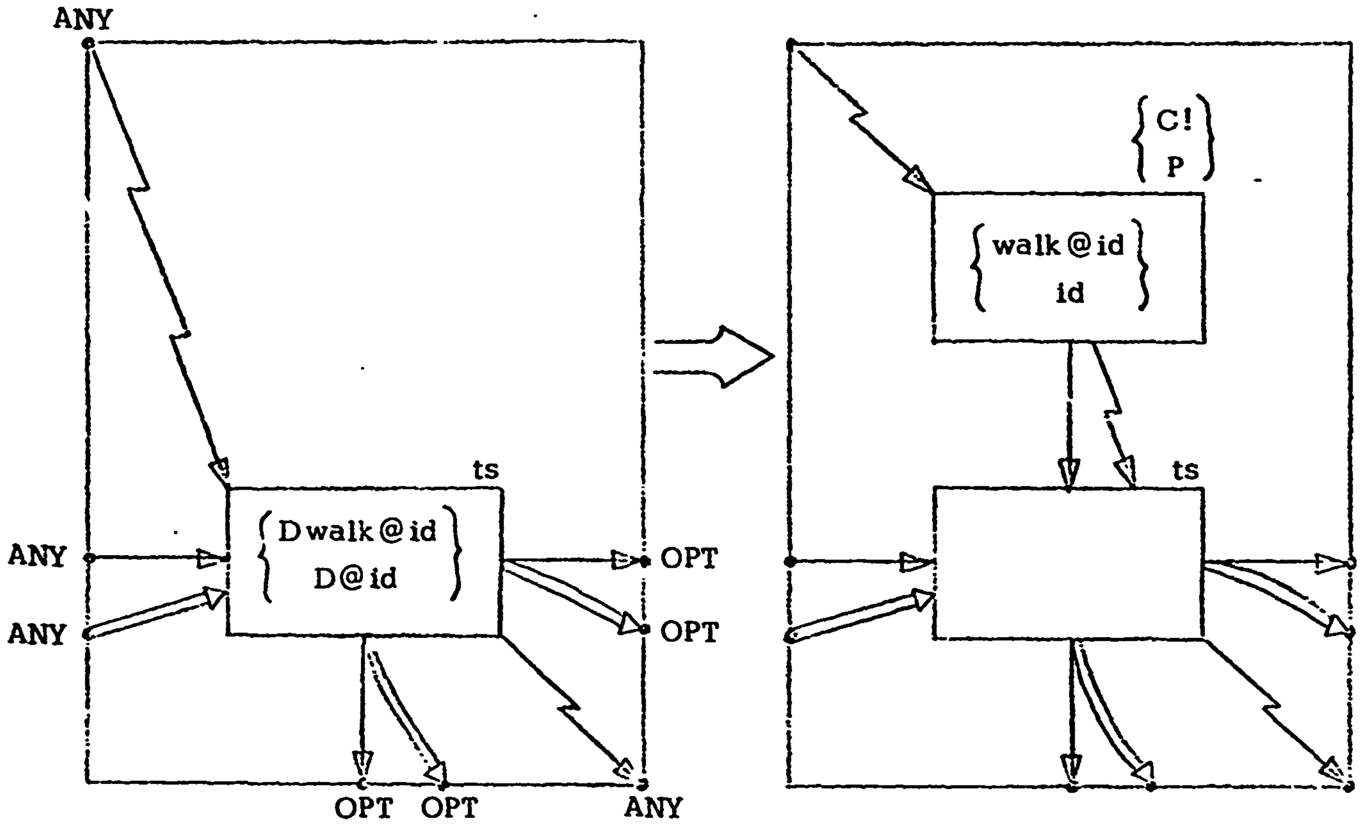
For a data node or call node, if its subname is of the form

@ id

where id is an <identifier>, then transform the subname to

D @ id

b. <u>Indirect Down (data node)</u>



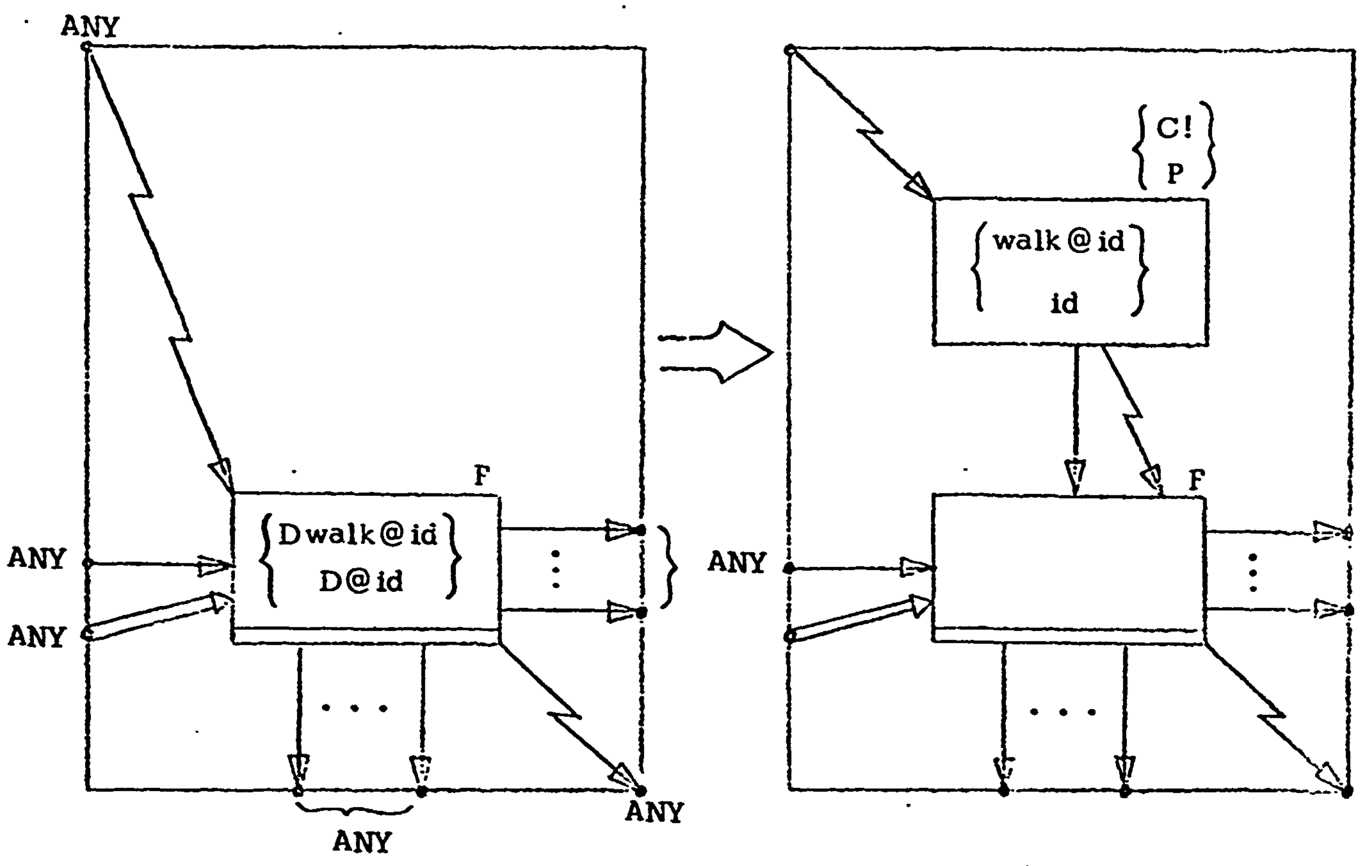
where

<u>walk</u> is an < indirect-walk >;

<u>id</u> is an < identifier >; and

<u>ts</u> is a < type-set >.

c. <u>Indirect Down (call node)</u>

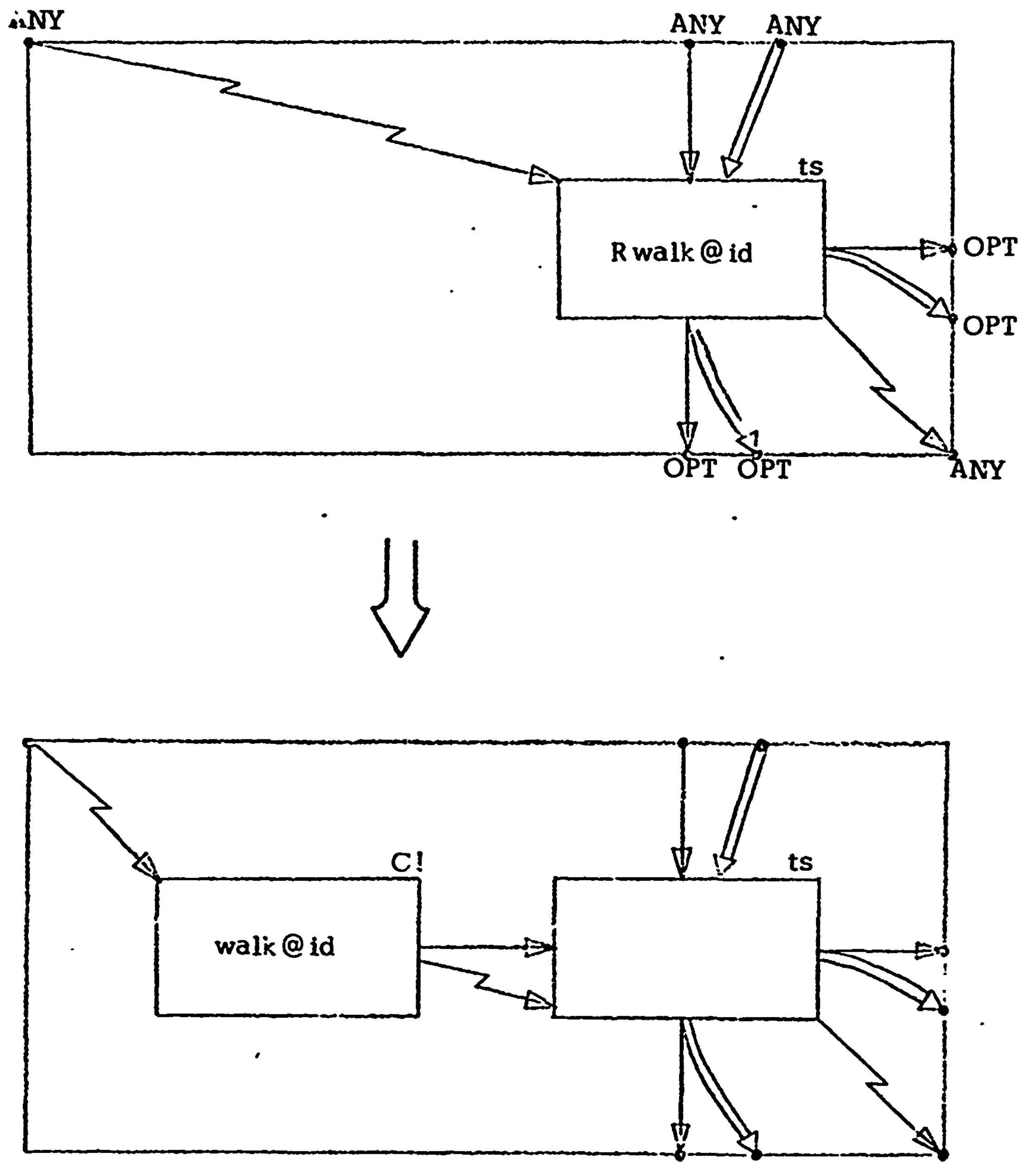


where

<u>walk</u> is an < indirect-walk >; and

<u>id</u> is an < identifier >.

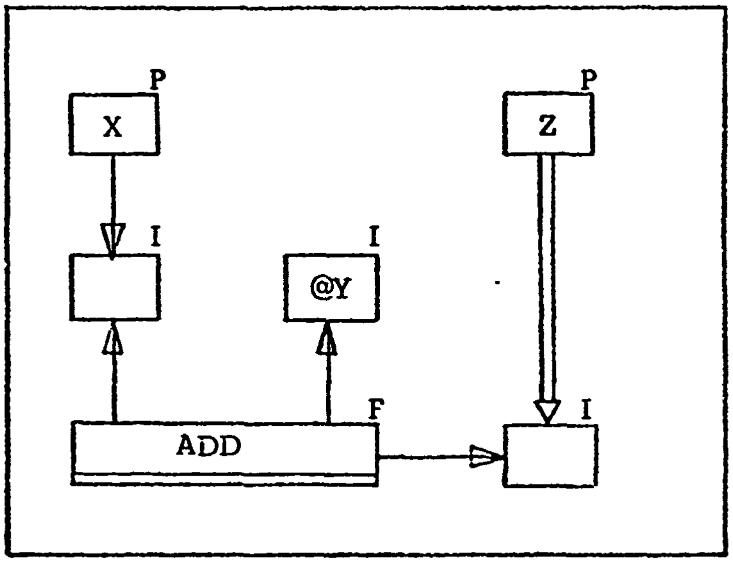
d. Indirect Right

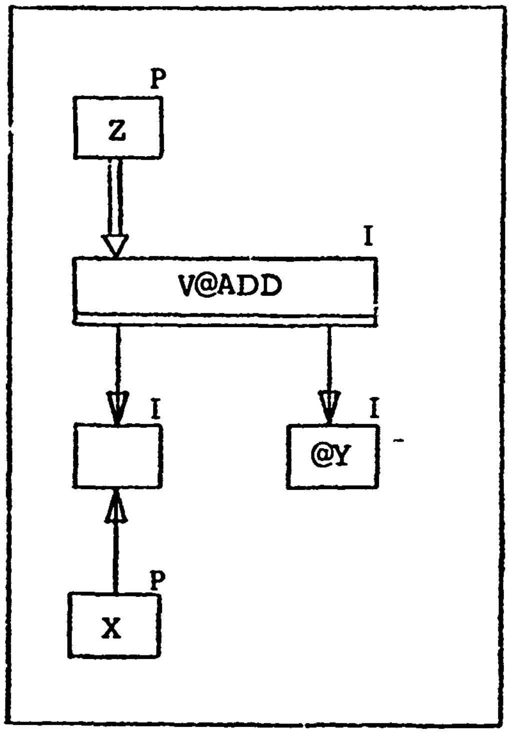


where

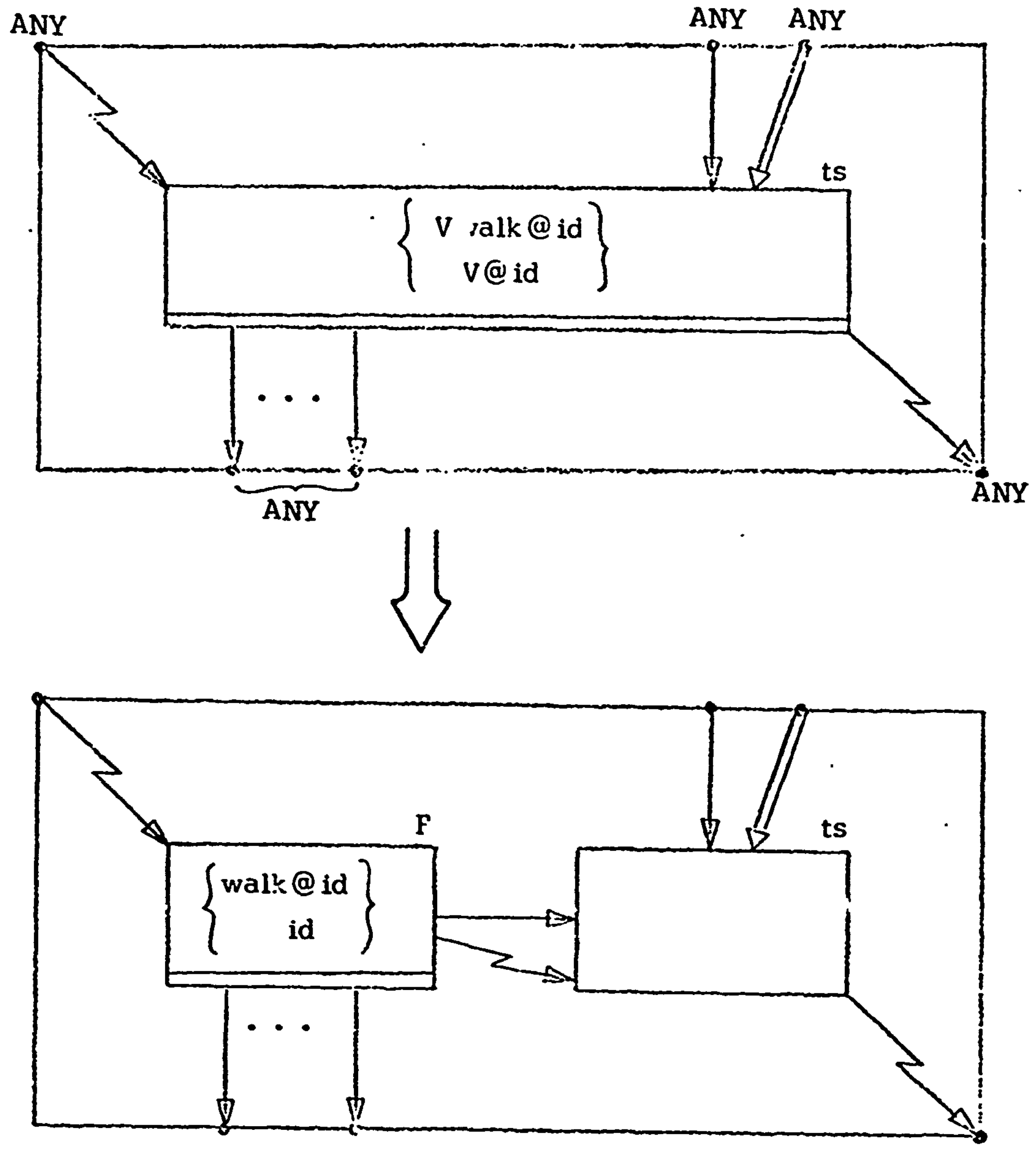
walk is an < indirect-walk >;
id is an < identifier >; and
ts is a < type-set >.

Value Call: There is a notation by which the result of a function (which has exactly one result) can be named by the name of the function itself. For example, the following two rules are equivalent:





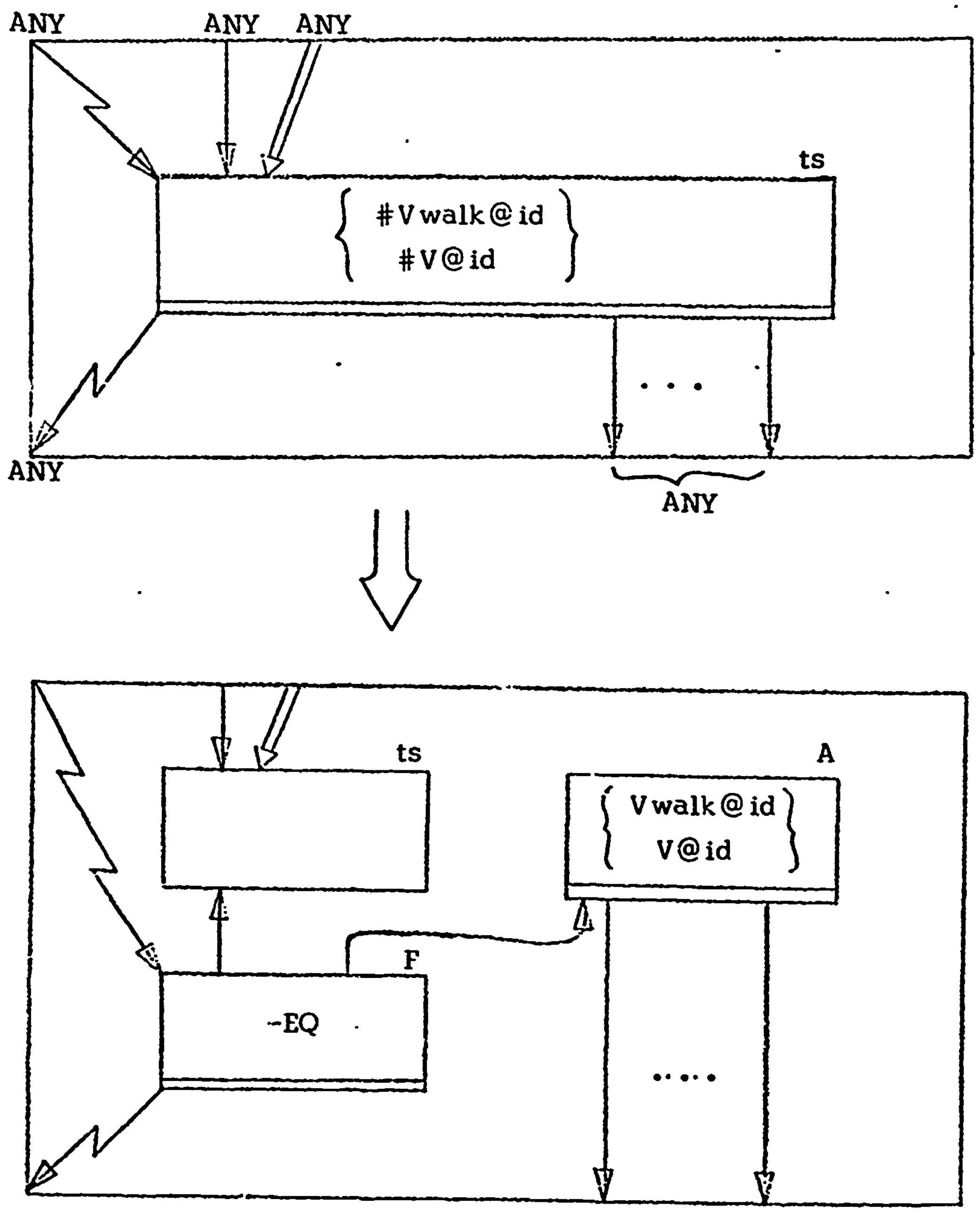
This notation is explained by the f›llowing transformation diagram:



where

walk is an < indirect-walk > ;
id is an < identifier > ; and
ts is a < type-set >.

Negative Value Call: If a node is accessible via some link from an accessible node its name may have the same form as a value call node, but with an additional number sign (#) prefixed. This notation is used to assert the node in question is not the result of the function specified. This notation is explained by the following transformation diagram:



where

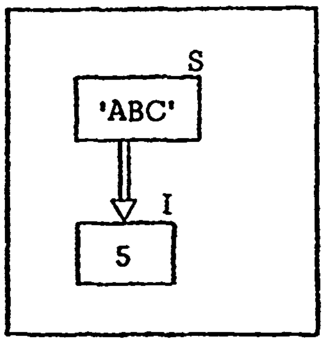
walk is an < indirect-walk >;

id is an < identifier >; and

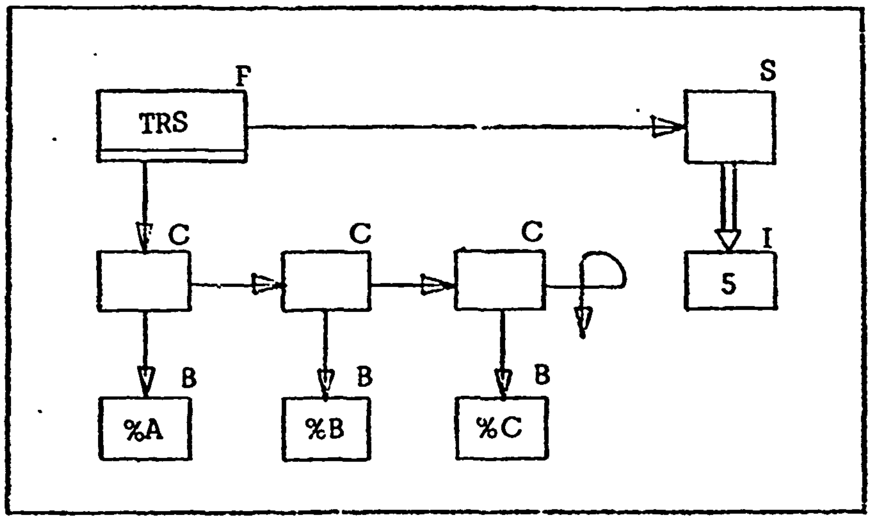
ts is a < type-set >.

The EQ function employed above is a predicate which succeeds if and only if its two arguments are the same; since its negation is called above (-EQ), that call is testing whether its two arguments are not the same.

String and Token Names: Although the programmer is expected to consider literal names of string nodes and token nodes as atomic, the implementation actually expands such names into calls on the built-in functions TRS and TRT. (These functions were mentioned on page 33.) Furthermore, gather notation is used to collect the argument(s) to these functions. For example, although the programmer writes the following rule:



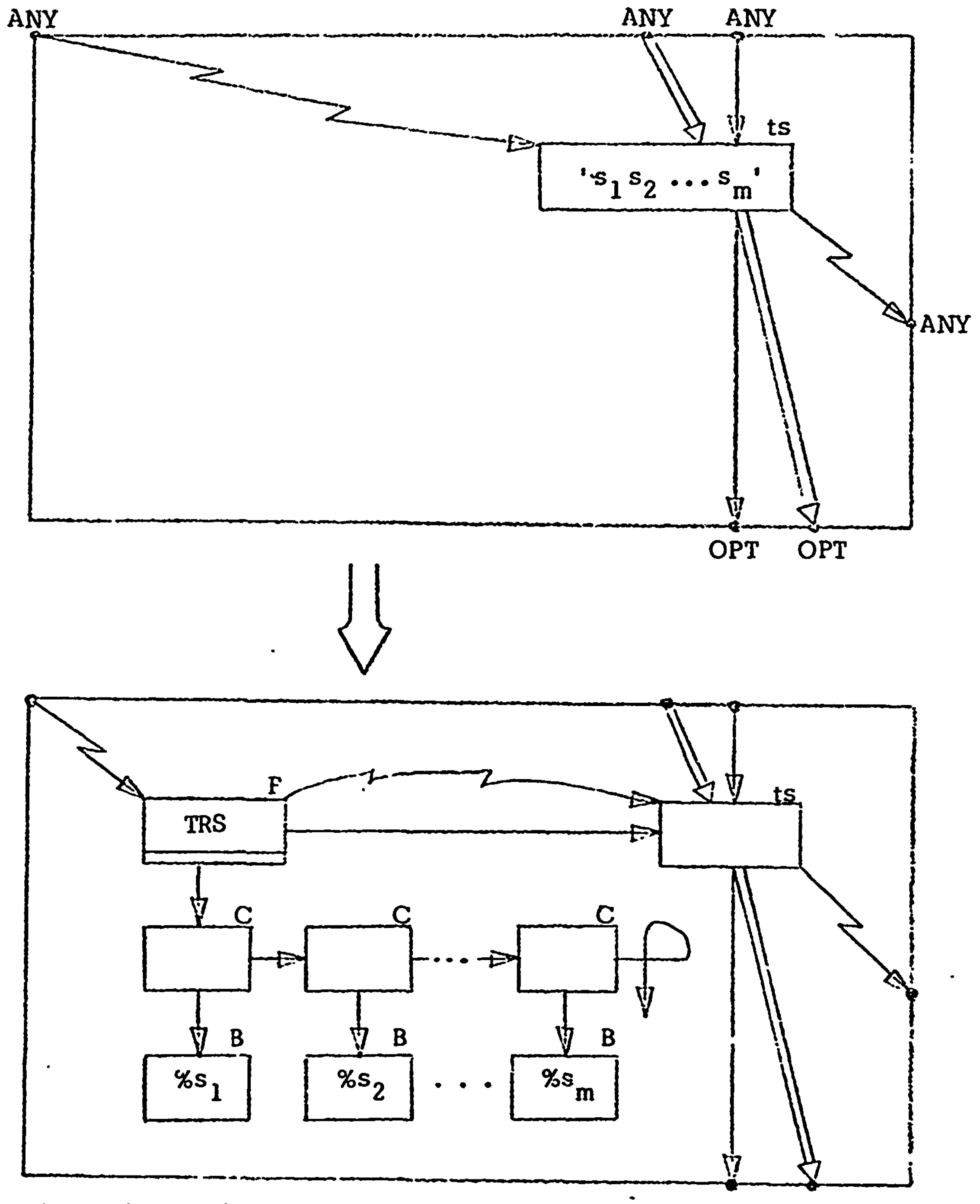
a macro transforms it to be executed as:



These macros are given by the following transformation diagrams:

24

String Names

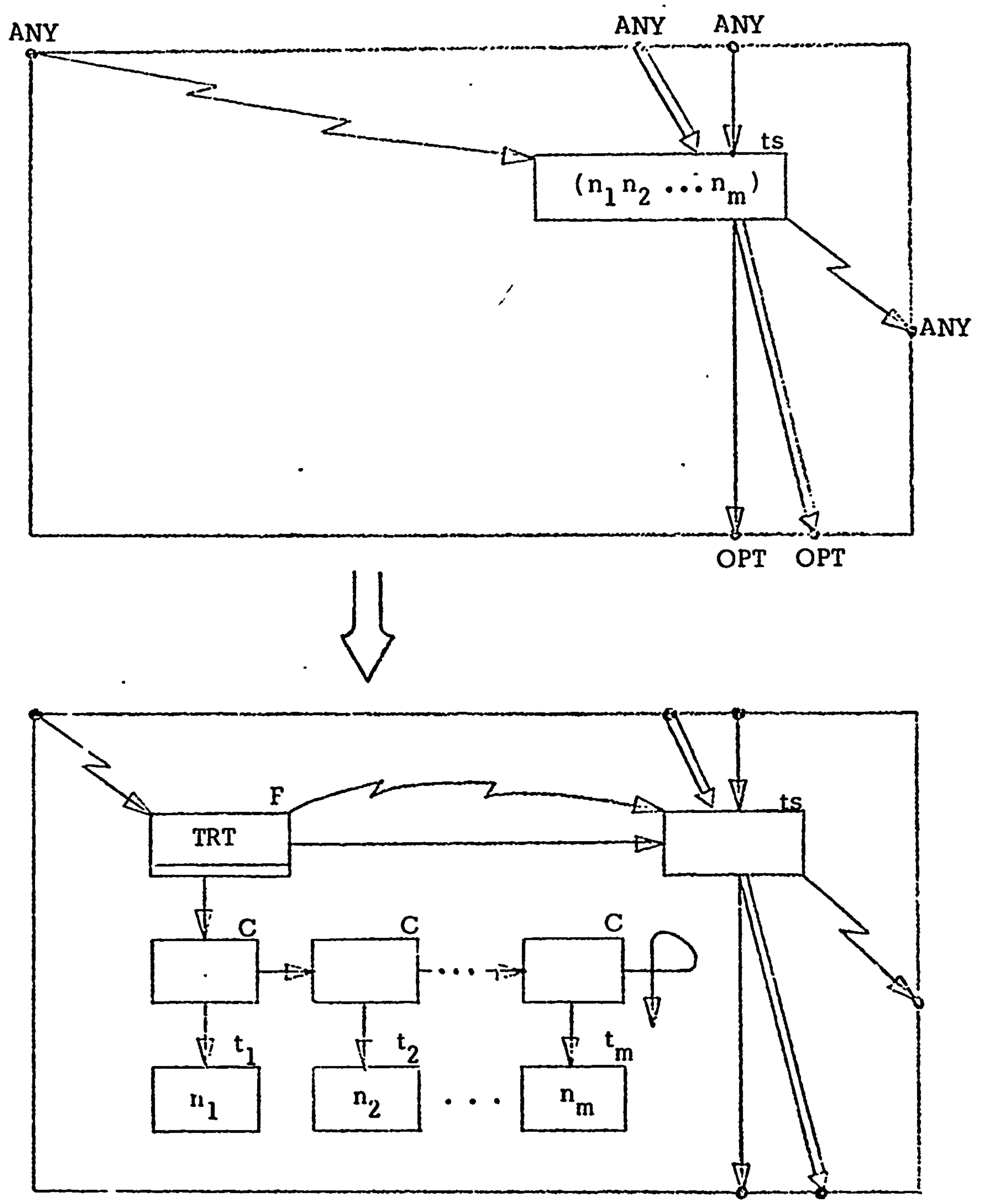


where, for $m \geq 0$,

$\underline{s_i}$ is a < symbol >; and

$\underline{ts}$ is a < type-set >.

<u>Token Names</u>



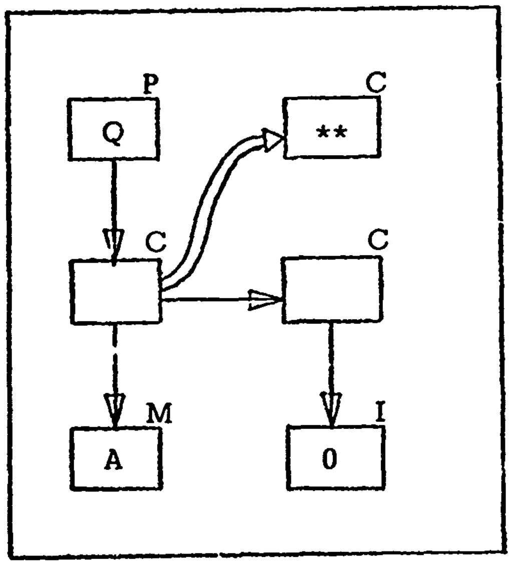
where, for $m \geq 0$,

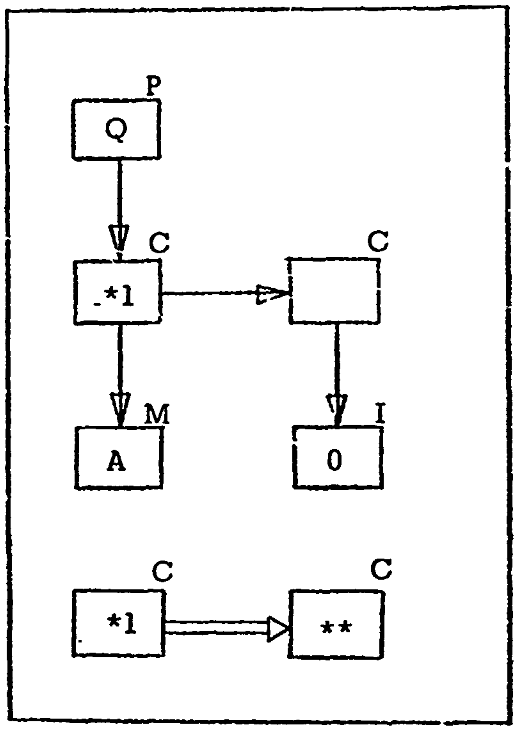
$\underline{n_i}$ is a < name >;

$\underline{t_i}$ is the < type > implies by n_i; and

<u>ts</u> is a < type-set >.

<u>Dummy Names</u>: There is a notation by which a node in a rule without a subname (including one employing name set notation) may be "split" for convenience into two or more instances in the rule. A dummy name begins with an asterisk (*) and that must be followed by either an unsigned integer or an identifier. For example, the following two rules are equivalent:





The usual reason for employing dummies is for drawing rules in ways which help demonstrate their operations more clearly. Also, any unnamed node which is not "split" may be named with a dummy name as a documentation convenience. There is, however, one use of dummies in a rule which can alter the interpretation of a rule: each instance of a dummy is separately visited during the pattern match (see comments associated with page 24). Thus by using flow links among dummies and other nodes of a rule, the programmer may perform rather subtle testing.

Pages 40-45: This section presents a good overview of most of the AMBIT/L Programming System. Further details of each part of the system are given in other memos. Other features available in the AMBIT/L not indicated in the paper are mentioned here and detailed elsewhere.

a. When the user initiates execution of an AMBIT/L program he may choose to accept the default memory allocation for the interpreter or he may exercise complete control over alternate allocations for specialized purposes. For example, he may wish to allocate an unusually large control stack for the execution of a highly recursive program.

b. There is an alternate interpreter which can be used to do instrumentation studies of the insert-block activities of an AMBIT/L program in execution. At any time the user may print complete statistics of the total time spent in each particular insert-block, the total number of times control transferred to that block, and the total number of times that block had to be read into core memory from disk. This has been a very useful tool for arriving at optimum memory allocations and for discovering those portions of an AMBIT/L program which deserve re-writing for the purpose of optimization.

c. It is also possible to instrument the total number of calls on each built-in function and the total number of calls on each basic operation of the AMBIT/L interpreter. This kind of instrumentation is normally of value to the AMBIT/L systems programmer, but may in some cases also be of interest to one who is a user of the system.

d. The normal AMBIT/L interpreter performs considerable checks on program validity at execution time and reports a variety of specific diagnostic error messages to the user. For a program which is well debugged and which must run as fast as possible, the user can employ a "production" interpreter which avoids most of this checking.

e. For those working with large AMBIT/L programs there is a cross-referencer which produces five individual listings of references which cross insert-block boundaries; these are separately presented for TEMPs, PERMs, MARKs, LABELs, and FUNCTIONs. The user has control over which insert-blocks of his program are to be considered for any particular application of the cross-referencer.

Pages 48-51:	Appendix A should be read only as a formal definition. The syntax presented does not correspond to that encodement actually used to represent programs in the AMBIT/L Programming System.
Page 49:	Formula F12 should be changed to: F12. unsigned-real $\rightarrow$ unsigned-decimal $\{\text{scale-factor}\}_0^1$ | $\{\text{digit}\}_1^\infty$ scale-factor
Pages 52-56:	Again, the program syntax should be read only as a formal definition since what is presented is incomplete compared with that of the encodement actually used to represent programs in the AMBIT/L Programming System.
Page 61:	The fourth paragraph labeled "Modify memory" should be at the same indentation as the first three paragraphs, and it should also have an equal sign as the other paragraphs.
Page 64:	As a comment on the first paragraph, it is useful to employ resetting of the execution value only when control is about to be transferred to the exit point of either a function body or a block. Otherwise, note (at the bottom of page 60) that the EVR is initialized to <u>success</u> at the beginning of every rule.

(END)

Section B

How to Write an AMBIT/L Program

January 10, 1972

This section provides the AMBIT/L programmer with supplementary information on block structure, declarations, insertions, and transfer lists which he needs to know to be able to write AMBIT/L programs.

This section presents, rather informally, some techniques of writing AMBIT/L programs. The Reference Manual (with its supplementary memo) describes the AMBIT/L Programming Language. Appendix B of the manual, which is admittedly difficult to read, describes how programs are interpreted. The syntax presented in that appendix is not complete; the programmer should refer to Section D, "The Syntax of the Encodement of AMBIT/L Programs" for the complete syntax of the current implementation.

An AMBIT/L program is structured by program blocks in the same method employed by ALGOL 60. Each block is bounded by a BEGIN statement at the beginning and an END statement at the end. A user's AMBIT/L program is organized overall as one block; often it is broken down internally into other blocks. Any contiguous sequence of rules (and/or blocks) within one block may be collected together as a sub-block by enclosing them within a BEGIN - END pair of statements; this organization is, however, of minor utility. When it is employed it appears as one giant rule to its surroundings, and as such the block may have a transfer-list associated with it. The success exit is taken if control ever falls through the end or is transferred to the end via an attached label on the END statement. Note that the identifier 'EXIT' is automatically declared as an attached label before the END statement of every block in the program. Thus the block's success may be caused by the execution of a 'S/EXIT' or 'F/EXIT' from some rule in that block. Failure of the block may likewise be caused by the execution of a 'S/-EXIT' or 'F/-EXIT'. Since label nodes (as well as function nodes) may include a minus sign preceding the identifier used as subname, indirect transfer of control may be employed to exit from a block with an execution value of either success or failure. (Indirect transfer of control means that a success or fail exit of a transfer list is specified as a walk from a pointer.)

More common is the use of a block as a function body; i.e., the executable part of a function definition. A function body may consist of just one rule without a transfer list. During execution that rule's success causes the function's success, and the rule's failure causes the function's failure. It is more efficient to write one-rule functions in this way when possible. Usually, a function definition consists of several rules, in which case they must be enclosed within a block. It is also necessary to enclose a single rule (which is a function body) within a block if its transfer-list must be given explicitly. As one would expect, success or failure of the function is directly affected by the success or failure of the block used as the function body. In addition to 'EXIT', the identifier 'RET' is automatically declared as an attached label before the END statement of every block in the program which is a function body. Thus returning from a function is usually specified by either S/RET or F/RET for success or S/-RET or F/-RET for failure.

Each block begins with any number (including zero) of declarations of identifiers as PERM (for a permanent pointer), TEMP (for a temporary pointer), and MARK (for a mark). Then there may appear any number (including zero) of function declarations. The remainder of the block includes any number (including zero) of rules or blocks, each optionally followed by a transfer list. Thus the block structure is employed (as in ALGOL 60) to delimit those parts of an AMBIT/L program where an identifier can be referenced; i.e., its scope. In AMBIT/L scoping of identifiers is done for PERMs, TEMPs, MARKs, FUNCTIONs, and LABELs. An identifier is declared as a label by following it with a colon and attaching it to either an imperative (rule or block) or to an END statement.

PERM pointers and TEMP pointers are both referenced in the same manner within rules and transfer-lists (when indirect transfer of control is used) ; i.e., simply as pointer nodes. Both types of pointers have identical scope; however, each PERM pointer is allocated exactly once for the duration of the execution of an AMBIT/L program. Initially, each PERM is made to point to the null cell. When the block in which a particular PERM is declared is entered or exited its value is not affected. A PERM acts like list to the TRT (transfer to token) built-in function to locate such a token.

Each function declaration includes a heading which specifies the argument pointers and result pointers to be used within the function body. An AMBIT/L function, therefore, has a fixed number of arguments and a fixed number of results according to how it is declared. All built-in functions in the AMBIT/L Programming System also have this characteristic except for one: MEMBER (alias ONEOF). Each argument pointer and result pointer of an AMBIT/L function are automatically declared as TEMP pointers within the body of the function definition, whether it is a one-rule body or a block. Within a function body the down link of any argument pointer may be modified and used as any other temporary pointer without any external effects.

In review, a block is used within AMBIT/L programs as:

1. the structure of each user program as a whole, or
2. a collection of a sequence of rules and/or blocks, or
3. a function body which cannot be a single rule.

In addition to the scoping of identifiers, there is one other important aspect of a block: in the AMBIT/L Programming System a program can be broken up along the boundaries of its block structure. Suppose, for example, that it is desirable to separate from a program a certain block, b. A copy of b can be made (from 'BEGIN' to its corresponding 'END') and placed in a separate source file. Then one declaration line must be inserted before the BEGIN statement in that file, for example:

```
INSERTION  FILE.NAME;
```

The entire block b in the original program is then replaced by the one corresponding command line:

```
INSERT  FILE.NAME;
```

The transformation on the preceding page may be performed on any or all of the blocks in a program, however deeply nested those blocks may be. Although such transformations have no effects on the logic of a program, they can have drastic effects on the economics of preparing and running that program since:

1. each insertion is compiled separately and independently of the compilation of every other insertion;

2. each insertion is link-edited depending only on the previous link-editing of all containing blocks; and

3. each insertion becomes a separate program segment or "page" of the reentrant binary code which is swapped in from the disk, according to need, automatically, by the interpreter during execution of the program.

Thus the AMBIT/L Programming System works in units of blocks (or insertions) for compiling, linking, and actual run-time program storage.

Notice that the name a programmer chooses for a particular insertion may have the syntax of an AMBIT/L identifier. Namely, it consists of alphanumeric characters and perhaps individually embedded periods, and it must begin with an alphabetic character; there is no practical restriction on its length. There is a convention that users' insertion names should not begin with the letter 'Z'; all built-in environmental insertion names start with 'Z'. The name chosen must match exactly in the corresponding INSERT and INSERTION commands, except the programmer is free to choose any name for the outermost or main block.

In the PDP - 10 implementation the name of each source file of an AMBIT/L program must correspond with its insertion name as follows: ignoring the periods of the insertion name, the primary name of the file must be the first six characters (or less if there are fewer) of the insertion name. The file name extension may be anything the user chooses (including null). For example, for the insertion FILE.NAME, the source file may be FILENA.AL . The extension "AL" is for "AMBIT/L" and is the default extension name assumed by the AMBIT/L Compiler and Diagram Generator. As another example, for the insertion A.B.C1, the source file may be ABC1.SRC . To avoid ambiguity, the primary names of the various insertions of one AMBIT/L program must be distinct.*

Now a rough example is given of a simple program which is composed of two insertions. First is presented the main or outermost block of the user program as might be contained in the file MYPROG.AL:

```
INSERTION MYPROG;
$ THIS IS AN EXAMPLE OF A COMMENT
BEGIN
TEMP A B C;

F(X) Y:
BEGIN    $ANOTHER COMMENT!
TEMP Z;
RULE  ⎫
  .   ⎬     several rules as body of F
  .   ⎪
  .   ⎭
END;

G(A) :
INSERT FUNC.G;

LOOP: RULE  ⎫
   .        ⎬     several rules as the main program
   .        ⎪
   .        ⎭
END
```

*This rule of naming a source file need not be strictly followed at compilation time or diagram generation time, but it is a requirement that the primary name of the REL file conforms to this rule when the insertion is linked by the AMBIT/L Link Editor.

Next is given the general format of the one inserted block as might be contained in the file FUNCG.AL:

```
INSERTION FUNC.G;
BEGIN
H():
RULE  }
  :   }   one rule as body of H
RULE  }
  :   }   several rules as body of G
  .   }
END
```

The reader should notice above that pointers A, B, and C are declared for use throughout the entire program. Function F is declared for use throughout the program; pointer X is used to receive the argument and Y is used to return its one result. Pointer Z is used locally within function F. Function G is declared with one argument and no results for use throughout the program. Notice that the argument pointer A overrides the other use of A within the body of function G. Function H is declared with no arguments or results for use only within the body of G.

Notice that each insertion begins with an INSERTION declaration and ends with an END statement which is not followed by a semicolon. All other appearances of an END statement in AMBIT/L programs are followed by a semicolon.

Earlier, it was stated that the identifier 'RET' is automatically declared as an attached label (in addition to 'EXIT') before the END statement of every block in the program which is a function body. Although this is logically true, it is not done in exactly this way for insertions which are function bodies; the compiler cannot make the distinction when it is compiling a particular insertion whether that insertion is a function body or a collection of rules and/or blocks. Thus 'RET' is really declared in the enclosing block where the INSERT command is specified for the block in question. (The confused reader should ignore this distinction if it disturbs him; it is significant only in making full use of the DAMBIT/L debugging package.) Consistent with this, the identifier 'RET' is also declared for a one-rule function, although the programmer cannot make use of it.

The novice AMBIT/L programmer will probably want to avoid the added complication of organizing his small tutorial programs into separate insertions. However, for any "real" program this activity should usually not be avoided. Beware, however, that probably not every block should be made into an insertion. There are several factors which must be considered to arrive at the appropriate choice for this organization:

1. Since compilation is somewhat expensive, the programmer should avoid an organization which leads to frequent re-compilations of large insertions due to, for example, a program which has potentially many bugs or which is gradually being modified. At the present time the DAMBIT/L debugging system cannot be used to patch programs or even modify the AMBIT/L data structure; thus recompilation is almost always required to correct a programming bug. A "large" insertion has over 50 rules of average complexity.

2. Even if it doesn't have to be compiled often, very large pages are a handicap because they are more difficult to fit into the limited region of memory used for pages. The automatic paging system of the interpreter makes room for a page which cannot otherwise fit by "kicking out" the oldest page and trying again; very large pages which are not used constantly can cause a lot of overhead activity in the paging system. A special version of the AMBIT/L interpreter is available for instrumentation of page timing characteristics; it also reports on the number of times each page must be read from the disk. Thus the user has the means to find out if such a high overhead situation exists.

3. There is some overhead for each insertion in terms of the number of files kept in the user's directory and the extra effort (and perhaps documentation) there is in maintaining a distinct program component. At execution time, there is some overhead in transfering control from page to page; note that when returning from a function whose body is an insertion, control must pass through RET which is on the page which inserts the function body. Thus rather small insertions should be avoided, and instead, small blocks ought to remain as parts of their parent insertions.

4. Since a page is not swapped in from the disk until needed, rarely invoked blocks should be made into insertions. Likewise, two insertions which are not very large might be merged if one is inserted by the other and both are nearly always executed together.

5. To help analyze the running characteristics of a program, a user may organize insertions in such a way that timings reported on the basis of duration on each page can yield meaningful results. A special version of the AMBIT/L interpreter is available for such instrumentation.

6. To help examine a program (perhaps written by someone else), a user may organize insertions in such a way that results of applying the AMBIT/L Cross-Reference Mapper can be useful. Such results are restricted to reporting only the existence of references which cross insertion boundaries.

7. The ultimate form of an AMBIT/L user program as a collection of pages of binary code is called a "DMP" (for "dump") file. Each page is represented as an integral multiple of 128 36-bit words. Therefore, if the length of a DMP file is to be minimal, a user may wish to organize insertions so that resulting pages

do not result in significant wastage, such as would result from several 257-word pages. The number of words on a page is reported to the user by the compiler and again by the link editor. A fanatic programmer may wish to try to shorten certain pages for this purpose by rewriting some rules or transfer lists.

Finally, it should be re-emphasized that all organizations of insertions of a program are logically equivalent, and that the choices available affect only the efficiency of the programmer and of the programming system.

Next, a different topic is introduced concerned with the writing of AMBIT/L programs. Following any rule or block which may have a transfer-list specified can be one of the following forms:

Form	Meaning	
S/α	S/α	F/?
F/β	S/NEXT	F/β
S/α F/β	S/α	F/β
F/β S/α	S/α	F/β
SF/α	S/α	F/α
(if not given)	S/NEXT	F/?

where α and β are label-references. Note there are several predefined labels, the last three of which are relative to the rule in which they are used:

Label	Use
EXIT	This identifier is automatically declared as an attached label before the 'END' of every block in the program.

Label	Use
RET	This identifier is automatically declared before the 'END' of every block in the program which is known to be a function body.
?	This special label should be employed in a rule exit which the programmer expects will never be taken. At execution time flow of control to this label causes the interpreter to initiate an error trap and print a diagnostic message on the terminal.*
PREV	This relative label refers to the previous imperative (i.e., rule or block) if it exists; otherwise its use is an error.
CUR	This relative label refers to the current imperative (i.e., rule or block).
NEXT	This relative label refers to the next imperative (i.e., rule or block) in the current block, or to the end of the block (the label EXIT) if the current imperative is the last one of the block.

*The error trap 'TUL' is caused by the interpretation of a S/? exit. The error trap 'F/?' is caused by the interpretation of a F/? exit.

Note that although the use of '?' as a label is restricted to be within the transfer list, the other five predefined labels may be used within rules (as label nodes).

One special construction is allowed which permits a dummy name to be used as the success exit of a rule, as shown in the following example:

```
RULE
(00020)

      +---A
      I   I
      I@Q I
      I   I
      +---+
        A
        I
        I
        I
        I
      +-------------F      +---A
      I             I      I   I
      I   GET.LAB   I----->I*I I
      I             I      I   I
      +=============+      +---+

S/*I;
```

Such a dummy name cannot be given as the fail exit since it might not be bound when failure of the rule is detected during the rule's interpretation.

(END)

Section C

The Drawing of AMBIT/L Programs and their Encodement

January 11, 1972

This section presents, by example, the forms of diagrammatic listings which the Diagram Generator of the AMBIT/L Programming System can produce. Thus, the programmer should first use this as a guide to drawing rules of his programs. Also described is the method of translating the diagrams into a linear encodement language. Finally, a recommended canonical encodement is provided.

It was indicated in the Reference Manual that a programmer sketches his program on paper and then inputs both the textual and diagrammatic portions of his program via a typewriter-like terminal (often a Model 33 Teletype). Thus an entire program is reduced to being a string of characters in the PDP - 10 implementation of the AMBIT/L Programming System. The programmer, however, initially prepares textual specifications for overall program structure and he draws diagrams for rules. It is important the programmer understands how the Diagram Generator produces diagrammatic listings from the character input so that he may organize the layout of each rule to be easy to understand. Selection of a good rule layout is important as a documentation aid, and the simpler structure there is to rule the more likely it will not have hard-to-find errors. To avoid over-entanglement of links the user should not hesitate to employ explicit link routing and dummy nodes.

The Diagram Generator (DIAGEN) is a translator in the AMBIT/L Programming System which reads as input an encodement of one insertion (or block) of an AMBIT/L program and produces as output a listing of that insertion. All textual material is passed right through DIAGEN without checking or formating, except for an attempt to introduce new pages (forms) where appropriate. When it comes across a rule, however, it assimilates the character string encodement and produces a listing of that rule as a diagram. When rules are short enough DIAGEN inserts new pages to avoid any rule getting split across the end of a page of the listing. DIAGEN ignores new-page marks (form feeds) in its input.

DIAGEN bases its drawing of rules on the medium it uses as output: the typed page (either by terminal or line-printer). To force the programmer to keep his diagrams within "copyable" size (e.g., by XEROX), the maximum width of a diagram is limited to be less than 72 character positions; this also corresponds to the printing width of the Model 33 Teletype. A rule is considered to be drawn on a rectangular grid of rows (named A, B, C, etc.) and columns (named 1, 2, 3, etc.). To fit on one 8 1/2" x 11" page a rule must use at most

6 rows and 7 columns. Although DIAGEN insists that all rules be no wider than 7 columns, it can accommodate up to a double-page rule which can have as many as 13 rows. The programmer may position a node at a particular grid position, for example, A2, B1, or D5. Each position normally takes up five character positions vertically and horizontally. Between each pair of adjacent 5-position boxes are five character positions available mostly for routing of links. The rule on the next page demonstrates the format just described. See how there are five character positions between the node at B2 and each of its vertical and horizontal neighbors. The left and right sides of each node boundary employ the digit '1' as a close approximation to a vertical bar. The top and bottom sides use a minus sign as an approximation to a horizontal bar. The corner of each boundary is designated by a plus sign. Type-sets are right adjusted as part of the top of a node boundary.

RULE
(00020)

```
+---A     +---A     +---A     +---A     +---A     +---A     +---A
1   1     1   1     1   1     1   1     1   1     1   1     1   1
1*A11     1*A21     1*A31     1*A41     1*A51     1*A61     1*A71
1   1     1   1     1   1     1   1     1   1     1   1     1   1
+---+     +---+     +---+     +---+     +---+     +---+     +---+



+---A     +---A     +---A     +---A     +---A     +---A     +---A
1   1     1   1     1   1     1   1     1   1     1   1     1   1
1*B11     1*B21     1*B31     1*B41     1*B51     1*B61     1*B71
1   1     1   1     1   1     1   1     1   1     1   1     1   1
+---+     +---+     +---+     +---+     +---+     +---+     +---+



+---A     +---A     +---A     +---A     +---A     +---A     +---A
1   1     1   1     1   1     1   1     1   1     1   1     1   1
1*C11     1*C21     1*C31     1*C41     1*C51     1*C61     1*C71
1   1     1   1     1   1     1   1     1   1     1   1     1   1
+---+     +---+     +---+     +---+     +---+     +---+     +---+



+---A     +---A     +---A     +---A     +---A     +---A     +---A
1   1     1   1     1   1     1   1     1   1     1   1     1   1
1*D11     1*D21     1*D31     1*D41     1*D51     1*D61     1*D71
1   1     1   1     1   1     1   1     1   1     1   1     1   1
+---+     +---+     +---+     +---+     +---+     +---+     +---+



+---A     +---A     +---A     +---A     +---A     +---A     +---A
1   1     1   1     1   1     1   1     1   1     1   1     1   1
1*E11     1*E21     1*E31     1*E41     1*E51     1*E61     1*E71
1   1     1   1     1   1     1   1     !   1     1   1     1   1
+---+     +---+     +---+     +---+     +---+     +---+     +---+



+---A     +---A     +---A     +---A     +---A     +---A     +---A
1   1     1   1     1   1     1   1     1   1     1   1     1   1
1*F11     1*F21     1*F31     1*F41     1*F51     1*F61     1*F71
1   1     1   1     1   1     1   1     1   1     1   1     1   1
+---+     +---+     +---+     +---+     +---+     +---+     +---+
```

When specifying that a node be placed at a particular grid point the encodement is done as:

position / type-set / subname

The subname corresponds to the text enclosed within the node boundary; it may be a subname of a data node or a form of macro call. The second slash and subname may be omitted for an unnamed node in the rule; however, if a subname is included a type-set must also be included. The letter 'A' (for "any") is an allowable type-set which means no type testing is done. A node may also be specified with just a position; this is equivalent in meaning to having 'A' as its type-set. The following sample input and associated diagram demonstrate these possibilities. A comma separates each node specification.

```
00020   RULE
00030   A1/IR/@X, A3/P/Y, B1//#LF, B4/M!, C2/A/#**, C4;
```

```
RULE
(00020)

      +--IR                  +---P
      I    I                 I    I
      I@X  I                 I Y  I
      I    I                 I    I
      +----+                 +----+



      +-#LF                              +--M!
      I    I                             I    I
      I    I                             I    I
      I    I                             I    I
      +----+                             +----+



                 +---A                   +----+
                 I    I                  I    I
                 I#** I                  I    I
                 I    I                  I    I
                 +----+                  +----+
```

The integer within parentheses under the word 'RULE' in the diagram indicates the line number or sequence number within the source file on which the rule's encodement begins. DIAGEN treats unsequenced files as if they were sequenced by one.

Up to this point names have been chosen carefully in these examples to not exceed three characters, for that is the limit that will fit inside a standard node which is 5 by 5 character positions. If a node name is given which is longer DIAGEN attempts to stretch the node to the right as required without "bumping into" another node or a link with some vertical component. If it can do so, it stretches the box so the name just fits. Otherwise, it labels the node with a name of its grid position followed by a period; then at the bottom of the rule each such label is repeated along with the full node name (somewhat like a footnote). For example, here is some input and its associated diagram:

```
RULE
A1/LFBS/@ABCD, A2/P/X, A5/T/(A 12), A7/I/1234, B1/M/LISTING,
B2/A/=1/3/5/7, B3/R/3.14159265358979323846, B7/I/12345,
C1/S/'SUPERCALIFRAGILISTICEXPIALIDOCIOUS ENCYCLOPEDIA OF AMERICA';

RULE
(00002)

     +--LFBS   +---P                         +------T            +----I
     1     1   1   1                         1      1            1    1
     1@ABCD1   1 X 1                         1(A 12)1            112341
     1     1   1   1                         1      1            1    1
     +-----+   +---+                         +------+            +----+




     +-------M +---A     +----------------------R                +---I
     1       1 1   1     1                      1                1   1
     1LISTING1 1B2.1     13.141592653589793238461                1B7.1
     1       1 1   1     1                      1                1   1
     +-------+ +---+     +----------------------+                +---+




     +------------------------------------------------------------S
     1                                                            1
     1'SUPERCALIFRAGILISTICEXPIALIDOCIOUS ENCYCLOPEDIA OF AMERICA'1
     1                                                            1
     +------------------------------------------------------------+

B2.=1/3/5/7
B7.12345
```

Notice that a node may stretch to have a name of up to seven characters when it has an immediate right neighbor and no links (vertical or horizontal) intervene.

A call node or value call node is specified by a type-set which begins with an equal sign. DIAGEN draws such node with a bottom side of the node boundary as equal signs so that it looks like a double line. For these nodes only, a larger node boundary may be specified by giving both an upper left and a lower right grid position separated by a minus sign. When such a specification is made DIAGEN does not attempt to stretch a box for a name which cannot fit. For example, here is some input and its associated diagram:

```
00002   RULE
00003   A1-C1/=F/OPEN, A2/=A/VDRRRD@A.LONG.FUNCTION.NAME,
00004   B2-C4/=F/ADD, B5-B6/=I/V@SUB, C5-C7/=F/@FUNC.PTR;
```

```
RULE
(00002)

+---F     +---------------------------A
I   I     I                           I
I   I     IVDRRRD@A.LONG.FUNCTION.NAMEI
I   I     I                           I
I   I     +===========================+
I   I
I   I
I   I
I   I
I   I
I   I     +-----------------------F     +-------------I
I   I     I                       I     I             I
IA1.I     I                       I     I    V@SUB    I
I   I     I                       I     I             I
I   I     I                       I     +=============+
I   I     I                       I
I   I     I                       I
I   I     I          ADD          I
I   I     I                       I
I   I     I                       I
I   I     I                       I     +-----------------------F
I   I     I                       I     I                       I
I   I     I                       I     I       @FUNC.PTR       I
I   I     I                       I     I                       I
+===+     +=======================+     +=======================+

A1.OPEN
```

Now that node specification has been fully described, links will be discussed. Since a standard node is 5 by 5 character positions, the middle three positions of each side are suitable for link origins and destinations. Recall that a solid link or a modification (double-line) link which emanates from either the upper or bottom side of a data node is considered to be a "down" link. Similarly, a solid or modification link emanating from either the left or right side of a data node is considered to be a "right" link. Argument links may likewise emanate from the upper and/or bottom sides of a call node or value node; and result links may emanate from the left and/or right sides of a call node.

The middle link origin of a side of a standard node is the "normal" one. When dealing with <u>down</u> links, a "plus" perturbation means towards the right (towards increasing column numbers), and a "minus" perturbation means towards the left. When dealing with <u>right</u> links a "plus" perturbation means towards the bottom of the page (towards "increasing" row letters), and a "minus" perturbation means towards the top of the page.

Links are routed in lanes on the page. Some normal lanes are those which pass through the normal link origins of each standard node position (both vertical and horizontal). The other normal lanes are those which are half-way between standard node positions. The following diagram shows all normal lanes in the vicinity of the first three rows (by N's) in a background of name-less nodes.

```
RULE
(00200)
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+
N | N | N | N | N | N | N | N | N | N | N | N | N | N |
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
N | N | N | N | N | N | N | N | N | N | N | N | N | N |
N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N   N   N   N   N   N   N   N   N   N   N   N   N   N
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+
N | N | N | N | N | N | N | N | N | N | N | N | N | N |
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
N | N | N | N | N | N | N | N | N | N | N | N | N | N |
N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N   N   N   N   N   N   N   N   N   N   N   N   N   N
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+
N | N | N | N | N | N | N | N | N | N | N | N | N | N |
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
N | N | N | N | N | N | N | N | N | N | N | N | N | N |
N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+ N +-N-+
N   N   N   N   N   N   N   N   N   N   N   N   N   N
N   N   N   N   N   N   N   N   N   N   N   N   N   N
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

Next is presented a similar diagram showing all plus perturbation lanes of the first three rows (by P's). It does not seem necessary to present a third diagram showing minus perturbation lanes.

```
RULE
(00200)
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+
   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1
   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+
   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1
   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+
   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1
   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1   P 1  P1
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+   P +--P+
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
   P    P    P    P    P    P    P    P    P    P    P    P    P    P
```

With this introduction the specification of a link can be explained. First, fully explicit link specification will be given. Then several default cases will be introduced which allow for rather simplified encodement; nearly all link specifications of most AMBIT/L programs take advantage of the defaults, especially for routing.

Links emanating from a particular node are specified (in any order) along with that node's position and name specification. A SPACE or carriage return separates the link specifications from the node specification and from one another. For a data node a link specification begins with a route. However, for a call node or value call node which has a large node boundary encompassing more than one grid position, each link from such a node must first have an origin grid position specified followed by a slash (/), and that is followed by a route.

A route begins with a letter which designates the kind of link:

S	for a SOLID (or normal) link
B	for a BROKEN* (double-line or modification) link
F	for a FLOW link

The remainder of the route consists of a sequence of segments. Each segment begins with a letter which designates its perturbation (i.e., the particular lane along which it travels):

N	for NORMAL (or no) perturbation
P	for PLUS perturbation
M	for MINUS perturbation

*Historically, modification links were drawn as broken lines rather than double lines.

The remainder of a segment consists of one or more occurrences of the same letter which designates direction:

U	for the direction UP
D	for the direction DOWN
L	for the direction LEFT
R	for the direction RIGHT

Each occurrence of a direction letter means that the route continues in that direction to the vicinity of the next normal lane which crosses its path. Although DIAGEN draws links which begin and end on node boundaries, link routes must be thought of as beginning and ending at the crossing of normal lanes inside of the source and destination node boundaries.

The route ends after one or more segment specifications. The complete link specification then ends with a slash followed by the grid position of the node at the destination of the link. If that node is a call node or value call node with a large node boundary, any grid position around the edge of that boundary may be the destination grid position of the link specification. Otherwise, the grid position of a data node as destination must be the destination grid position of the link specification - EVEN IF THAT DATA NODE HAD BEEN STRETCHED TO ACCOMMODATE A LARGE NAME!

Solid links emanating from call nodes and value call nodes indicate arguments and results. Both arguments and results of a function have a particular ordering according to the function's definition. Thus it is essential that there is no ambiguity concerning the order of arguments or order of results. Links used to locate arguments of a function may originate at either the top or bottom side of a call node boundary. Similarly, result links may emanate from either the left or right side of a call node boundary. Arguments are ordered from left to right; results are ordered from top to bottom. There is no restriction which forces all arguments or all results to originate on the same side of a call node. The ordering or arguments and results can easily be shown on large call nodes by using different grid positions as origins. Also, perturbations can be used to show ordering. To avoid any ambiguity, no two argument links (or two result links) may emanate from the same origin point or two points exactly on opposite sides of the node boundary.

A few examples should clarify the above descriptions; here is some input and its associated diagram:

```
00020   RULE
00030   A1/P/A SNDD/B1,  A2/P/B SPDD/B2,
00040   A3/C/@C SNLNDDMR/B3,
00050   A4/P/D FMDDDNLLPD/C3 BNDDDD/C4 FNRR/A5,
00060   A5/T/(E) SNDDNRNDDNLLL/C4,
00070   A6/P/F SMDNLLLLLLPD/C3,
00080   B1/I,  F2/CST,  B3,
00090   C1-C2/=F/DDD@G C1/SNUU/B1 C2/SMUU/B2
00100     C1/SNDD/D1 C2/SNDNLLPD/D1,
00110   C3/S/'ABCD' FNLL/C2 BMUNLLPU/B2 SNUU/B3,
00120   C4/M/NEWNODE,  D1/I/G;
RULE
(00020)

+---P     +---P     +---C     +---P     +---T     +---P
1   1     1   1     1   1     1   1     1   1     1   1
1 A 1     1 B 1  /--1@C 1     1 D 1ZZZZ>1(E)1     1 F 1
1   1     1   1  1  1   1     1   1     1   1     1   1
+---+     +---+  1  +---+     +---+     +---+     +---+
  1          1   1             ZH         1        1
  1          1   1             ZH         1        1
  1          1   1     /-------++---------+--------/
  !          1   1     1       ZH         1
  V          V   1     V       ZH         1
+---I     +-CST  1  +---+      ZH         1
1   1     1   1  \->1   1      ZH         1
1   1     1   1     1   1      ZH         \----\
1   1     1   1     1   1      ZH              1
+---+     +---+     +---+      ZH              1
  A        A A       A         ZH              1
  1        1 H       1         ZH              1
  1        1 \=======\1/ZZZZZZZ/H              1
  1        1         H1Z        H              1
  1        1         H1V        V              1
+-------------T     +------S  +-------M        1
1             1     1      1  1       1        1
1    DDD@G    1<ZZZZ1'ABCD'1  1NEWNODE1<-------/
1             1     1      1  1       1
+=============+     +------+  +-------+
  1         1
  1         1
  1/--------/
  1:
  VV
+---I
1   1
1 G 1
1   1
+---+
```

Note how solid links are drawn with l's and minus signs, and "broken" links are drawn with H's and equal signs as approximations to double lines. Flow links employ the letter 'Z' in both vertical and horizontal directions. Slash and back-slash are used where links turn corners. Angle brackets are used for left and right heads of links, and letters 'A' and 'V' are used for upward and downward heads of links. A plus sign is used where links cross one another.

There is one additional notation for specifying those solid or broken links which employ the twisted link notation (shorthand for a link whose destination is the null cell). Such links have a route which consists of one perturbation letter and one direction letter; then after the separating slash the destination of such a link is specified as two asterisks (**). The next example demonstrates this notation.

The previous example judiciously avoided stretched node boundaries bumping into links. The next example demonstrates such effects. Here is some input and its associated diagram:

```
00020   RULE
00030   A1/P/ABCDE SND/** FNRR/A2,
00040   A2/P/ABCDEF SNDD/B2 FNRR/A3,
00050   A3/P/ABC DND/** SPDNRNUUNRRNDML/A4,
00060   A4/I/1234;
00070   B1/A/@A12345 SNLNDNRRNDNL/C1,
00080   B2/=I/V@SQ SNDD/C2,
00090   B3/A/*1=RD@P FNRNDNLLLPD/C2,
00100   C1/A/*1#**,
00110   C2/I/@BIG.INTEGER.FOR.AREA.OF.SPOT ,NRRRRRRNDNLLLLLLLNU/C1;
```

```
RULE
(00020)
                                /---------\
                                |         |
                                |         :
     +-----P   +------P  +---P  |  +----I ;
     |     |   |      |  |   |  |  |    |</
     |ABCDE|ZZ>|ABCDEF|Z>|ABC|  |  |1234|
     |     |   |      |  |   |  |  |    |
     +-----+   +------+  +---+  |  +----+
       |         |         Hi   |
                 |          |   |
                 |          \---/
                 |
                 V
     +-------A +----|    +---A
     |       | |    |    |   |
  /--|@A12345| |V@SQ|    |B3.|ZZ\
  |  |       | |    |    |   |  Z
  |  +-------+ +====+    +---+  Z
  |              |              Z
  |              |              Z
  \---------\    |/ZZZZZZZZZZZZZ/
            |    |Z
            |    VV
     +---A  |  +-----------------------------I
     |   i  |  |                             |
     |C1.|<-/  |@BIG.INTEGER.FOR.AREA.OF.SPOT|Z\
     |   |     |                             | Z
     +---+     +-----------------------------+ Z
       A                                       Z
       :                                       Z
       \ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ/

B3.*1=RD@P
C1.*1#**
```

Most of the complexity of composing link routings can be simplified by using various default options. First, if the kind of link is omitted then S (for solid) is assumed. Second, if the perturbation is omitted then N (for normal) is assumed. Thus it is customary never to see either of these letters in link specifications.

When a link emanates from a large call node or value call node, its source grid position and separating slash can be omitted in its specification if that position is the upper left position of that node boundary.

The most important default option is the "default route" which permits the user to use just one direction letter to specify all segments of a route. For example, here is some input and its associated diagram:

```
00020   RULE
00030   A1/I/G,
00040   A2-A3/=F/FOO D/A1 A3/D/B2 A3/D/D3 A3/D/C4 A3/FR/A7,
00050   A7/P/P BD/C4 D/B7,
00060   B2/I/1,
00070   B7/I/4,
00080   C4/I/3,
00090   D3/I/2;
```

```
RULE
(00020)
        /---\
        1   1
        V   1
     +---I  1  +-------------F                                  +---P
     1   1  1  1             1                                  1   1
     1 G 1  1  1     FOO     1ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ>1 P 1
     1   1  1  1             1                                  1   1
     +---+  1  +=============+                                  +---+
            1   1         111                                    HI
            1   1         111                                    HI
            \---/ /-------/1\-------\ /==========================/1
                  1        1        1 H                           1
                  V        1        1 H                           V
               +---I       1        1 H                         +---I
               1   1       1        1 H                         1   1
               1 1 1       1        1 H                         1 4 1
               1   1       1        1 H                         1   1
               +---+       1        1 H                         +---+
                           1        1 H
                           1        1 H
                           1        1 H
                           1        1 H
                           1        V V
                           1       +---I
                           1       1   1
                           1       1 3 1
                           1       1   1
                           1       +---+
                           1
                           1
                           1
                           1
                           V
                         +---I
                         1   1
                         1 2 1
                         1   1
                         +---+
```

This example demonstrates several default down links and a default right link. The general idea is that DIAGEN will draw a straight link if possible; when it does, it uses the normal lane (no perturbation). A default link always terminates on its destination node on the side opposite the one from which it originated. A link which turns corners leaves its origin node with a perturbation towards where it will first turn, and its final segment has a perturbation towards from where the link just turned. Except for the first and last segments of a link which turns, all other segments of links specified with a default route travel on normal (no perturbation) lanes. Another heuristic used is that for a link which turns, those turns are made as soon as possible. Finally, if a link must make a complete 360° turn it does so counter-clockwise. For example, the link specified as:

B3 BR/B2,

will be drawn with the following explicit route:

B3 BMRULLLLDMR/B2,

this route is shown in the following diagram:

```
RULE
(00110)




          /==================\
          H                  H
          H                  H
          H  +---+     +---+ H
          \=>I   I     I   I==/
             I   I     I   I
             I   I     I   I
             +---+     +---+
```

Note that all default routes are always drawn independently of other parts of the rule which they may cross or overlay. Thus if a programmer wishes to produce a rule without conflicts he must know what to expect from a default route. Consider the problem of interchanging the values of two pointers. The following input and associated diagram demonstrate three alternatives. Notice that DIAGEN permits the specification of overlay, and indicates it with a '$' at each character position where there is a conflict.

```
00110   RULE
00120   A1/P/A D/B1 BD/B2,
00130   A2/P/B BD/B1 D/B2,
00140   A3/P/C D/B3 BU/B4,
00150   A4/P/D BD/B3 D/B4,
00.60   A5/P/E D/B5 BPDMRRMD/B6,
00170   A6/P/F DPLLPD/B5 BPDD/B6,
00180   B1, B2, B3, B4, B5, B6;
```

```
RULE
(00110)
                       /===\
                       H   H
                       H   H
+---P     +---P     +---P  H  +---P     +---P     +---P
1   1     1   1     1   1  H  1   1     1   1     1   1
1 A 1     1 B 1     1 C 1  H  1 D 1     1 E 1     1 F 1
1   1     1   1     1   1  H  1   1     1   1     1   1
+---+     +---+     +---+  H  +---+     +---+     +---+
  1H       H1         1    H   H1         1H        1H
  1H       H1         1    H   H1         1\=======\1H
  1$$$$$$$$\1         1/===+===/1         1        H1H
  1H       H1         1H   H    1         1/-------+/H
  VV       VV         VV   H    V         VV       V V
+---+     +---+     +---+  H  +---+     +---+     +---+
1   1     1   1     1   1  H  1   1     1   1     1   1
1   1     1   1     1   1  H  1   1     1   1     1   1
1   1     1   1     1   1  H  1   1     1   1     1   1
+---+     +---+     +---+  H  +---+     +---+     +---+
                           H   A
                           H   H
                           \===/
```

The use of '$' to show conflict may be caused also by links conflicting with node boundaries or node names, and nodes conflicting with nodes, as demonstrated in the next diagram. Also shown next is the effect of forgetting to provide a proper link origin of a result link emanating from a large call node. Here is some input and its associated diagram:

```
00020   RULE
00030   A1/P/A D/C1,
00040   A2-B3/=F/DEF,
00050   A2/P/BCD BD/B1,
00060   B1/M/XYZ,
00070   B4/I,
00080   C1,
00090   C2-C3/=F/F R/B4;
```

```
RULE
(00020)

     +---P     $$$$P---------F
     1   1     $   1         1
     1 A 1    ,$BCD1         1
     1   1     $   1         1
     +---+     $---+         1
       1       1H            1
       1       1H            1
       1/======$/     DEF    1
       1H      1             1
       1V      1             1
     +-$-M     1             1    +---I
     1 1 1     1             1    1   1
     1X$Z1     1             1    1   1
     1 1 1     1      /------$---->1   1
     +-$-+     +======$======+    +---+
       1              1
       1              1
       1              1
       1              1
       V              1
     +---+     +------$------F
     1   1     1    --/      1
     1   1     1      F      1
     1   1     1             1
     +---+     +=============+
```

All fundamentals of the diagrammatic parts of rule encodement have been covered. The syntax of AMBIT/L program encodement, which is provided in a separate memo, should be consulted as the authority on rule encodement. A few comments on the textual portion of a program are given here, however, to give some context for the encodement of a rule.

Although the syntax indicates it is optional, for DIAGEN to properly perform, each rule specification must begin with the word "RULE". That is then followed (after a SPACE or carriage return as separator) by one or more node specifications; each of these is separated by a comma. Following the last node specification of a rule may optionally be a transfer list (when one is allowed - a transfer list is not allowed following a rule employed as a function body). If there is none, a semicolon follows the last node specification thus serving as the terminator of the rule. Otherwise, two consecutive slashes follow the last node specification; then a transfer list is specified in one of the forms given below. The transfer list is finally terminated by a semicolon which also serves as the terminator of the rule.

Allowable Forms

S/α

F/α

S/α F/β

F/β S/α

SF/α

where α and β are label-references.

Finally, a suggested canonical form for rule encodement is presented based on the experience of several AMBIT/L programmers. Although the syntax allows for rather free form encodement, a stylized encodement leads to source files which are easier to read (by a human) and to edit using the conventional text editing programs available on the PDP - 10. The encodement language for AMBIT/L programs is meant to serve a one-way communication from the programmer to the system. However, the realities of use usually require the editing of the encodement. The syntax permits optional use of any number of SPACEs, TABs and carriage returns nearly everywhere except where they would break up an identifier or within string quotation marks. The canonical encodement, however, restricts the employment of these separators.

The following is a listing of the canonical encodement of two rules:

```
RULE
A1-A3/=F/OPEN A1/D/B1 A2/D/B2
  A3/D/B3,
B1/M/OUTPUT,
B2/M/OUT,
B3/M/TTY;
LOOP: RULE
A1/P/X D/B1 BD/B3,
B1 R/B2,
B2/A/#** R/B3,
B3//
S/CUR F/NEXT;
```

Note that the first line includes any attached label(s) of the rule and the word 'RULE'. If there are labels then 'RULE' is at the left margin. Then the canonical encodement continues with one line per node specification if that is possible. In the case where two or more lines are required, the split occurs between link specifications. Otherwise, individual SPACEs are used as separators within a node specification. Additional or continuation lines may be indented by a couple of SPACEs for easier recognition. Each node specification of a rule except the last ends with a comma.

If a transfer list of any kind is given, the last node specification is terminated by two slashes, and the next line includes one or two exit specifications terminated by a semicolon. If two exits are included they are separated by a SPACE or a TAB. If no transfer list is given, the last node specification ends in a semicolon.

Although any ordering of node specifications is allowed, a canonical encodement calls for them to be ordered by rows and columns as has been shown above. This convention usually makes it possible to locate a particular node specification within a rule relative to the first line of its encodement (i.e., where 'RULE' is).

Each programmer may wish to decide for himself whether he wants to establish his own conventions for the ordering of links within node specifications. The canonical encodement does not specify any ordering, but the author has prefered to order links as "solid", "broken", and "flow" and then within each category "down" before "right". At least it is preferable to maintain the relative ordering of argument links and result links of call nodes and value call nodes.

(END)

Section D

The Syntax of the Encodement of AMBIT/L Programs

December 14, 1971

This section employs a BNF-like grammar to present the syntax for one insertion of an AMBIT/L Program in its encoded form.

An AMBIT/L program is composed of one or more separately-compiled "insertions". This section presents a BNF-like grammar for the syntax of the encodement of one insertion. The grammar may be consulted as the authority on what is acceptable to the AMBIT/L Compiler.

The grammar is organized in four parts: insertion syntax, rule syntax, name syntax, and symbol syntax. The productions of each part are not self-contained, but all four parts together cover the grammar.

As in BNF, a vertical bar is used as a metasymbol to separate various choices. Lower-case words which may be hyphenated are used as non-terminals of the grammar. A pair of square brackets encloses an optional constituent, i.e., one which may be included zero or one time. A pair of curly braces encloses a constituent which may be included any number (including zero) of times.

In general, a separator may be inserted anywhere in an insertion between two constituents unless the grammar includes the special meta-symbol φ , which means its left and right neighbors must be concatenated. A non-null separator must be used between two constituents if both its left and right immediate neighbors are then either alphanumeric or the character period. The syntax of a separator is:

separator $\longrightarrow$ { space | tab | carriage-return | comment }

comment $\longrightarrow$ $ { true-symbol } carriage-return

The non-terminal "space" represents the character obtained by depressing the space bar on the keyboard. The "tab" represents a horizontal tabulation character (ASCII HT), which is obtained as CTRL I on the Model 33 Teletype. The "carriage-return" represents a new line, i.e., both a carriage-return and a line feed. The "true-symbol" is defined in the symbol syntax as any printable character on the Model 33 Teletype including the space.

Insertion Syntax

```
I1.   insertion          →   prolog  block

I2.   prolog             →   INSERTION  insertion-name ;

I3.   block              →
                             insert-command  |
                             BEGIN
                             { declarative ; }
                             { function-defn ; }
                             { attached-label : | imperative ; }
                             END

I4.   insert-command     →   INSERT  insertion-name

I5.   declarative        →   declarator  identifier  { identifier }

I6.   declarator         →   PERM | TEMP | MARK

I7.   function-defn      →   function-heading :  function-body

I8.   function-heading   →   function-name ( { argument } ) { result }

I9.   function-body      →   free-imperative

I10.  imperative         →   free-imperative  [ // transfer-list ]

I11.  free-imperative    →   block | rule
```

```
I12.   transfer-list      →     S / exit-label [ F / exit-label ] |
                                F / exit-label [ S / exit-label ] |
                                SF / exit-label

I13.   exit-label         →     [ - ] pure-exit-label | dummy | indirect | ?

I14.   pure-exit-label    →     identifier | relative-label

I15.   relative-label     →     RET | EXIT | PREV | CUR | NEXT

I16.   insertion-name   ⎫
                        ⎪
I17.   attached-label   ⎪
                        ⎪
I18.   function-name    ⎬  →    identifier
                        ⎪
I19.   argument         ⎪
                        ⎪
I20.   result           ⎭
```

<u>Rule Syntax</u>

R1.	rule	→	[RULE] node { , node }
R2.	node	→	data-node | call-node
R3.	data-node	→	data-boundary [/ data-content] { data-link }
R4.	data-boundary	→	position
R5.	data-content	→	type-part [/ name-part]
R6.	data-link	→	[link-type φ] route / destination
R7.	link-type	→	S | B | F
R8.	call-node	→	call-boundary / = call-content { call-link }
R9.	call-boundary	→	position [- position]
R10.	call-content	→	F [/ name-part] | type-part / value-call
R11.	call-link	→	[origin /] [S φ] route / destination
R12.	type-part	→	type-set [!]
R13.	type-set	→	[#] type { φ type } | A
R14.	type	→	F | I | R | S | T | B | C | P | L | M
R15.	name-part	→	name | [name] name-test

R16.	name-test	→	= name { / name } | # name { / name }
R17.	value-call	→	[#] V φ indirect
R18.	origin	→	position
R19.	destination	→	position | **
R20.	position	→	letter φ digit
R21.	route	→	segment { φ segment }
R22.	segment	→	[perturbation φ] direction
R23.	perturbation	→	N | P | M
R24.	direction	→	U | D | L | R

Name Syntax

N1.	name	→	dummy | indirect | literal
N2.	dummy	→	* identifier | * unsigned-integer
N3.	indirect	→	[indirect-walk] @ identifier
N4.	indirect-walk	→	{ D φ | R φ } D
N5.	literal	→	token | string | basic-symbol | real | integer | null-cell | function-or-label | pointer-or-mark
N6.	token	→	({ literal })
N7.	string	→	' φ { quoted-symbol φ } '
N8.	basic-symbol	→	% φ symbol
N9.	real	→	[sign] unsigned-real
N10.	integer	→	[sign] unsigned-integer
N11.	null-cell	→	**
N12.	function-or-label	→	[-] identifier
N13.	pointer-or-mark	→	identifier
N14.	unsigned-real	→	unsigned-decimal [φ scale-factor] | unsigned-integer φ scale-factor

N15.	unsigned-decimal	⟶	unsigned-integer φ . [φ unsigned-integer] | . φ unsigned-integer
N16.	scale-factor	⟶	E φ unsigned-integer | E sign unsigned-integer
N17.	unsigned-integer	⟶	digit { φ digit }
N18.	identifier	⟶	letter { φ alphnum } { φ. φ alphnum { φalphnum}}

Symbol Syntax

S1.	symbol	→	true-symbol | control-symbol
S2.	quoted-symbol	→	free-symbol | % φ protected-symbol
S3.	true-symbol	→	free-symbol | special-symbol
S4.	protected-symbol	→	control-symbol | special-symbol
S5.	control-symbol	→	CR | LF | VT | FF | TAB | ESC | SUB
S6.	special-symbol	→	% | '
S7.	free-symbol	→	(any printable character on the Model 33 Teletype including the space)
S8.	alphnum	→	letter | digit
S9.	letter	→	(any upper-case alphabetic character: A | B | ... | Z)
S10.	digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
S11.	sign	→	+ | -

(END)

Section E

AMBIT/L Built-in Functions for the Programmer

October 11, 1971

This section describes the functions predefined or built-in to the AMBIT/L System upon which the AMBIT/L programmer may call to perform standard operations. The built-in functions for input/output are discussed in Section F.

There are well over 100 primitive built-in functions implemented in machine language as part of the AMBIT/L interpreter. Some of these are private to be used by system programs only. To an AMBIT/L programmer most of the primitive functions can be called as built-in functions. In addition, some built-in functions are written in AMBIT/L and defined in the environment. This distinction will not be made in the following descriptions.

The built-in functions of AMBIT/L can be categorized into classes as follows:

a) Arithmetic Computation

ADD	add two numbers
ADD1	add 1 to a number
SUB	subtract one number from another
SUB1	subtract 1 from a number
NEG	negate a number (i.e., change its sign)
ABS	yield absolute value of a number
MUL	multiply two numbers
SQ	square a number
DVQ	divide yielding quotient
DVR	divide yielding remainder
DVQR	divide yielding quotient and remainder
MAX	yield maximum of two numbers
MIN	yield minimum of two numbers

b) Arithmetic Predicates

EQ0	is a number equal to 0?
NE0	is a number not equal to 0?
LT0	is a number less than 0?
LE0	is a number less than or equal to 0?

GT0	is a number greater than 0
GE0	is a number greater than or equal to 0?
EQ	is one number equal to another?
NE	is one number not equal to another?
LT	is one number less than another?
LE	is one number less than or equal to another?
GT	is one number greater than another?
GE	is one number greater than or equal to another?

c) Logical Computation

AND	yield logical AND
OR	yield logical inclusive - OR
XOR	yield logical exclusive - OR
NOT	yield logical NOT
LSHIFT	logical shift

d) Membership Predicates

EQ	are the arguments equal?
NE	are the arguments not equal?
EQNUL	is the argument equal to the NULL CELL?
SGN	is the argument either the BASIC SYMBOL %+ or the BASIC SYMBOL %- (i.e., an arithmetic sign)?
LETTER	is the argument a BASIC SYMBOL which represents one of the 26 upper case letters?
DIGIT	is the argument a BASIC SYMBOL which represents one of the ten decimal digits?
ALPHNUM	is the argument a BASIC SYMBOL which represents either an upper case letter or a decimal digit?
PRINT.CHAR	is the argument a BASIC SYMBOL which represents a printing character?
BEFORE	is one character strictly earlier in the ASCII collating sequence than another?

AFTER	is one character strictly later in the ASCII collating sequence than another?
MEMBER	is the first argument the same as any other argument?

e) List and Structure Processing

LAST	yield the last CELL of a list
LENGTH	yield the length of a list
CYCLE.LIST	is the argument a CELL which heads a cyclic list?
CYCLE.STRUCT	is the argument a structure which includes at least one cycle?
COMPARE.CELL	compare two arguments for equality or for whether they are equivalent CELLs
COMPARE.LIST	compare two arguments for equality or for whether they are CELLs which head equivalent lists
COMPARE.STRUCT	compare two arguments for whether they are equivalent structures
CAT	catenate (or concatenate) two lists
COPY.CELL	copy a CELL or a terminal node
COPY.LIST	copy a list or a terminal node
COPY.STRUCT	copy a structure

f) Free Storage Management

GCOL	invoke garbage collection of free storage
FLTH	update FREE.CT with the free storage length
FREE.CELL	free a CELL
FREE.LIST	free the CELLs of a list
FREE.STRUCT	free the CELLs of a structure

g) Type Transfers

TRI	transfer to an INTEGER
TRR	transfer to a REAL
TRS	transfer to a STRING
TRT	transfer to a TOKEN
TRD	transfer to the display

h) Miscellaneous Functions

LENGTH	yield an integer which indicates the length of the argument
NEXTB	yield the next BASIC SYMBOL
PREVB	yield the previous BASIC SYMBOL
RANDOM	yield a pseudo-random number
PJOB	yield the job number
RUNTIME	yield the running time in KCS
USER.BREAK	cause a user break into the DAMBIT/L debugger
AMBIT.EXIT	exit and terminate execution

The various built-in functions will now be presented by classes. Most functions detect error conditions which lead to error traps. At present, an error trap causes the AMBIT/L System to type a message on the terminal and then enter the DAMBIT/L debugging system (unless it is considered to be "fatal"). In the future the "almost fatal" traps will be implemented as actual traps where a function call is performed so the programmer may substitute his own recovery procedures. Such a trap facility now exists only for the input/output built-in functions.

Arithmetic Computation

All of these functions may be called with either INTEGER or REAL arguments, except for DVR and DVQR which are defined only for INTEGERs. If a function has two arguments their types must agree; otherwise, an error trap occurs. An error trap also occurs if the type of an argument is neither INTEGER nor REAL. Results of these functions are either of type INTEGER or REAL, according to the given arguments.

If a computation involving REALs produces an overflow condition, an error trap occurs. This problem does not arise with INTEGERs, since (in AMBIT/L) an INTEGER has practically unlimited precision.

An attempt to divide by zero causes an error trap to occur.

None of these functions can FAIL.

ADD or ADD.I or ADD.R

Use: add two numbers

Arguments:

#1: an INTEGER or REAL representing the first addend

#2: an INTEGER or REAL (same type as the first argument) representing the second addend

Results: #1: the INTEGER or REAL (same type as the arguments) which represents the sum of the two addends

Notes: The name ADD.I is available as a mnemonic for adding INTEGERs, and the name ADD.R is available as a mnemonic for adding REALs; all three names invoke the same function.

ADD1 or AD1

Use: add 1 to a number

Arguments:

#1: an INTEGER or REAL

Results: #1: the INTEGER or REAL (same type as the argument) which represents the number resulting from adding 1 or 1.0 to the number represented by the argument

SUB or SUB.I or SUB.R

Use: subtract one number from another

Arguments:

#1: an INTEGER or REAL representing the minuend

#2: an INTEGER or REAL (same type as the first argument) representing the subtrahend

Results: #1: the INTEGER or REAL (same type as the arguments) which represents the difference of the minuend minus the subtrahend

Notes: The name SUB.I is available as a mnemonic for subtracting INTEGERs, and the name SUB.R is available as a mnemonic for subtracting REALs; all three names invoke the same function.

SUB1 or SB1

Use: subtract 1 from a number

Arguments:

#1: an INTEGER or REAL

Results: #1: the INTEGER or REAL (same type as the argument) which represents the number resulting from subtracting 1 or 1.0 from the number represented by the argument

NEG or NEGATE or NEG.I or NEG.R

Use: negate a number (i.e., change its sign)

Arguments:

#1: an INTEGER or REAL

Results: #1: the INTEGER or REAL (same type as the argument) which represents the negation of the number represented by the argument

Notes: The name NEG.I is available as a mnemonic for negating an INTEGER, and the name NEG.R is available as a mnemonic for negating a REAL; all four names invoke the same function.

ABS or ABSOLUTE

Use: yield absolute value of a number

Arguments:

#1: an INTEGER or REAL

Results: #1: the INTEGER or REAL (same type as the argument) which represents the absolute value (or magnitude) of the number represented by the argument

MUL or MUL.I or MUL.R

Use: multiply two numbers

Arguments:

#1: an INTEGER or REAL representing the multiplicand

#2: an INTEGER or REAL (same type as the first argument) representing the multiplier

Results: #1: the INTEGER or REAL (same type as the arguments) which represents the product of the multiplicand times the multiplier

Notes: The name MUL.I is available as a mnemonic for multiplying INTEGERs, and the name MUL.R is available as a mnemonic for multiplying REALs; all three names invoke the same function.

SQ

Use: square a number

Arguments:

#1: an INTEGER or REAL

Results: #1: the INTEGER or REAL (same type as the argument) which represents the square of the number represented by the argument

DVQ

Use: divide yielding quotient

Arguments:

#1: an INTEGER or REAL representing the dividend

#2: an INTEGER or REAL (same type as the first argument) representing the divisor

Results: #1: the INTEGER or REAL (same type as the arguments) which represents the quotient resulting from the division of the dividend divided by the divisor

DVR

Use: divide yielding remainder

Arguments:

#1: an INTEGER representing the dividend

#2: an INTEGER representing the divisor

Results: #1: the INTEGER which represents the remainder resulting from the division of the dividend divided by the divisor

DVQR

Use: divide yielding quotient and remainder

Arguments:

#1: an INTEGER representing the dividend

	#2:	an INTEGER representing the divisor
Results:	#1:	the INTEGER which represents the quotient resulting from the division of the dividend divided by the divisor
	#2:	the INTEGER which represents the remainder resulting from the division

MAX

Use: yield maximum of two numbers

Arguments:		
	#1:	an INTEGER or REAL
	#2:	an INTEGER or REAL (same type as the first argument)
Results:	#1:	the INTEGER or REAL (same type as the arguments) which represents the maximum of the numbers represented by the two arguments

MIN

Use: yield minimum of two numbers

Arguments:		
	#1:	an INTEGER or REAL
	#2:	an INTEGER or REAL (same type as the first argument)
Results:	#1:	the INTEGER or REAL (same type as the arguments) which represents the minimum of the numbers represented by the two arguments

Arithmetic Predicates

These functions may be called with either INTEGER or REAL arguments. If a function has two arguments their types must agree; otherwise an error trap occurs. An error trap also occurs if the type of an argument is neither INTEGER nor REAL.

If the arithmetic predicate is TRUE the function SUCCEEDS; if it is FALSE the function FAILS. These functions have no results.

First, the arithmetic predicates with one argument are presented:

Function Name (s)	Condition for SUCCESS
EQ0	argument equal to 0
NE0	argument not equal to 0
LT0 or ISNEG or IS.NEG	argument less than 0
LE0	argument less than or equal to 0
GT0 or ISPOS or IS.POS	argument greater than 0
GE0	argument greater than or equal to 0

Next, the arithmetic predicates with two arguments are presented. Each one is a comparison of the first argument with the second one (in that order).

Function Name	Condition for SUCCESS
EQ*	equal
NE*	not equal
LT	less than
LE	less than or equal
GT	greater than
GE	greater than or equal

*Although EQ and NE may be used to compare the equality of two INTEGERs o REALs, they are predicates which may accept any data nodes as argumen [illegible] they are also classified as membership predicates.

Logical Computation

These functions operate on INTEGERs which represent 36-bit words in the PDP-10 implementation of AMBIT/L. Thus these functions are machine-dependent and implementation-dependent. An AMBIT/L INTEGER is represented in the standard two's complement form which the PDP-10 machine structure expects. Thus those INTEGERs which can represent 36-bit words have values between -2^{35} and $2^{35}-1$. If an INTEGER whose value is outside of this range and which is meant to represent a 36-bit word is given as argument to any logical computation function, the function treats that argument as -2^{35}.

AND

Use: yield logical AND

Arguments:

#1: an INTEGER representing a 36-bit word

#2: an INTEGER representing a 36-bit word

Results: #1: the INTEGER representing the 36-bit word which is the bit-by-bit logical AND of the 36-bit words represented by the arguments

OR

Use: yield logical inclusive - OR

Arguments:

#1: an INTEGER representing a 36-bit word

#2: an INTEGER representing a 36-bit word

Results: #1: the INTEGER representing the 36-bit word which is the bit-by-bit logical inclusive-OR of the 36-bit words represented by the arguments

XOR

Use: yield logical exclusive - OR

Arguments:

#1: an INTEGER representing a 36-bit word

#2: an INTEGER representing a 36-bit word

Results: #1: the INTEGER representing the 36-bit word which is the bit-by-bit logical exclusive - OR of the 36-bit words represented by the arguments

NOT

Use: yield logical NOT

Arguments:

#1: an INTEGER representing a 36-bit word

68

Results: #1: the INTEGER representing the 36-bit word which is the bit-by-bit logical NOT (or one's complement) of the 36-bit word represented by the argument

LSHIFT

Use: logical shift

Arguments:

#1: an INTEGER representing a 36-bit word

#2: an INTEGER representing a shift count

Results: #1: the INTEGER representing the 36-bit word which results from performing a logical shift of the 36-bit word represented by the first argument by the number of bit positions indicated by the value of the second argument; if the second argument is greater than 0 a left shift is done; if the second argument is less than 0 a right shift is done; any bits "dropping out of either end" are lost

Membership Predicates

These functions determine whether a first argument is a member of [illegible] of data nodes. That set may be fixed or it may depend upon other arguments; the set may have only one member. If the predicate is TRUE the function SUCCEEDS; if it is FALSE the function FAILS. These functions have no results.

EQ

Use: are the arguments equal?

Arguments:

#1: any data node

#2: any data node

NE

Use: are the arguments not equal?

Arguments:

#1: any data node

#2: any data node

EQNUL

Use: is the argument equal to the NULL CELL?

Arguments:

#1: any data node

SGN

Use: is the argument either the BASIC SYMBOL %+ or the BASIC SYMBOL %- (i.e., an arithmetic sign)?

Arguments:

#1: any data node

LETTER

Use: is the argument a BASIC SYMBOL which represents one of the 26 upper case letters?

Arguments:

#1: any data node

DIGIT

Use: is the argument a BASIC SYMBOL which represents one of the ten decimal digits?

Arguments:

#1: any data node

ALPHNUM or ALFNUM

Use: is the argument a BASIC SYMBOL which represents either an upper case letter or a decimal digit (i.e., an alphanumeric character)?

Arguments:

#1: any data node

<u>PRINT.CHAR</u>

Use: is the argument a BASIC SYMBOL which represents a <u>print</u>ing <u>char</u>acter (i.e., one whose ASCII octal code is between 40 and 137 inclusive; note that the character SPACE is included)?

Arguments:

#1: any data node

<u>BEFORE</u>

Use: is the character represented by the first argument <u>before</u> (strictly earlier) in the ASCII collating sequence than the character represented by the second argument?

Arguments:

#1: a BASIC SYMBOL which represents an ASCII character

#2: a BASIC SYMBOL which represents an ASCII character

<u>AFTER</u>

Use: is the character represented by the first argument <u>after</u> (strictly later) in the ASCII collating sequence than the character represented by the second argument?

Arguments:

#1: a BASIC SYMBOL which represents an ASCII character

#2: a BASIC SYMBOL which represents an ASCII character

<u>MEMBER or ONEOF</u>

Use: is the first argument the same as any <u>one</u> <u>of</u> the other arguments?

Arguments:

#1: any data node

Others: this function can accept any number of other arguments as any data nodes

Notes: This function FAILS if only one argument is given.

List and Structure Processing

Within this category are various types of functions: those which compute some result based on the given data, predicates for cycle testing, predicates for determining equivalence of structures, and functions used for transforming and copying structures. The COMPARE.CELL and COPY.CELL functions are defined for logical completeness, but they are of minor utility.

LAST

Use:	yield the last CELL of a (non-cyclic) list	
Arguments:		
	#1:	a CELL
Results:	#1:	the CELL which points RIGHT to the NULL CELL and which is accessible by following RIGHT links from the argument
Notes:	If the argument is the NULL CELL it is returned as the result. If the list is cyclic an error trap occurs.	

LENGTH*

Use: yield the length of a (non-cyclic) list

Arguments:

#1: a CELL*

Results: #1: the INTEGER which represents the number of CELLs in the given list other than the NULL CELL; this is 0 when the NULL CELL is given as argument

Notes: If the list is cyclic an error trap occurs.

CYCLE.LIST or CYCLST

Use: is the argument a CELL which heads a cyclic list?

Arguments:

#1: any data node

Results: none

Notes: This predicate FAILs when its argument is not a CELL, or if its argument is the NULL CELL, or if the NULL CELL is accessible by following RIGHT links from the argument. This function SUCCEEDS if its argument is a CELL which heads a list which has a cycle. A list has a cycle if there is a CELL (X) other than the NULL CELL accessible from the initial CELL by following RIGHT links such that there is a (non-null) sequence of RIGHT links leaving X which leads back to X.

*The LENGTH function may also be called with an argument of an INTEGER, a STRING, or a TOKEN. The complete description of this function is given under "miscellaneous functions."

CYCLE.STRUCT or CYCSTR

Use: is the argument a structure which includes at least one cycle?

Arguments:

#1: any data node

Results: none

Notes: This predicate FAILs if its argument is not a CELL, or if its argument is the NULL CELL, or if all "walks" through CELLs beginning at the argument and following RIGHT and/or DOWN links lead to a terminal node. A terminal node is either the NULL CELL or any data node other than a CELL. This function SUCCEEDS if its argument is a CELL which heads a structure which has a cycle. A structure has a cycle if there is a CELL (X) other than the NULL CELL accessible from the initial CELL by following RIGHT and/or DOWN links through CELLs such that there is a (non-null) sequence of RIGHT and/or DOWN links leaving X which leads back to X via CELLs.

COMPARE.CELL or CMPCEL

Use: compare two arguments for equality or for whether they are equivalent CELLs

Arguments:

#1: any data node

#2: any data node

Results: none

Notes: This predicate SUCCEEDs if its two arguments are the same or if they are equivalent CELLs; otherwise it FAILs. Two CELLs are equivalent if they both point DOWN to the same node and if they both point RIGHT to the same node.

COMPARE.LIST or CMPLST*

Use: compare two arguments for equality or for whether they are CELLs which head equu..alent (non-cyclic) lists

Arguments:

#1: any data node

#2: any data node

Results: none

Notes: This predicate SUCCEEDs if its two arguments are the same or if they are CELLs which head equivalent lists; otherwise it FAILs. Two lists are equivalent if they are the same, or if the first CELL (X) of one list and the first CELL (Y) of the other list both point DOWN to the same node and X points RIGHT to a list which is equivalent to the one to which Y points by its RIGHT link. If the arguments are two different CELLs they must each head a list with no cycles; otherwise an error trap occurs.

*This function has another synonym which is considered obsolete: 'COMPARELIST'.

COMPARE.STRUCT or CMPSTR*

Use: compare two arguments for whether they are equivalent (non-cyclic) structures

Arguments:

#1: any data node

#2: any data node

Results: none

Notes: This predicate SUCCEEDs if its arguments are equivalent structures; otherwise it FAILs. Two structures are equivalent if they are the same, or if each is headed by a CELL such that the header CELL (X) of one structure and the header CELL (Y) of the other structure point DOWN to two equivalent structures and X points RIGHT to a structure which is equivalent to the one to which Y points by its RIGHT link. If the arguments are two different CELLs they must each head a structure with no cycles; otherwise an error trap occurs.

*This function has two other synonyms which are considered obsolete: 'COMPARESTRUCTURE' and 'COMPARE.STRUCTURE'.

CAT

Use: catenate (or concatenate) two lists

Arguments:

#1: a CELL

#2: a CELL

Results: #1: the CELL which heads the list produced by catenating the list headed by the first argument to the list headed by the second argument; this function properly handles empty lists

Notes: If the list headed by the first argument is cyclic an error trap occurs. This function may create a cyclic list if its two arguments are already part of the same list. The AMBIT/L function definition presented below and on the next page is equivalent to the built-in CAT function.

```
CAT(A B) C :
BEGIN
RULE
(00030)

      +---P    +---P     +---P
      1   1    1   1     1   1
      1 B 1    1 A 1     1 C 1
      1   1    1   1     1   1
      +---+    +---+     +---+
        1        1         H
        1        1         H
        1        1/========/
        1        1H
        V        VV
      +---C    +--C!
      1   1    1   1
      1** 1    1   1
      1   1    1   1
      +---+    +---+

 S/RET F/NEXT;
```

```
RULE
(00100)

    +---P     +---P     +---P
    1   1     1   1     1   1
    1 A 1     1 B 1     1 C 1
    1   1     1    1     1   1
    +---+     +---+     +---+
      1         1        H
      1         1        H
      1         1/=======/
      1         1H
      V         VV
    +---C     +--C!
    1   1     1   1
    1** 1     1   1
    1   1     1   1
    +---+     +---+

 S/RET F/NEXT;




RULE
(00170)

    +---P     +---P     +---P
    1   1     1   1     1   1
    1 A 1     1 C 1     1 B 1
    1   1     1   1     1   1
    +---+     +---+     +---+
      1        H          1
      1        H          1
      1/=======/          1
      1H                  1
      VV                  V
    +--C!     +---+     +--C!
    1   1     1   1     1   1
    1   1     1   1====>1   1
    1   1  /->1   1     1   1
    +---+  1  +---+     +---+
      A    1
      1    1
      1    1
      1    1
      1    1
    +----F 1
    1    1-/
    1LAST1
    1    1
    +====+

 S/RET;
END;
```

COPY.CELL or CPYCEL

Use: copy a CELL or a terminal node

Arguments:

#1: any data node

Results: #1: the data node which was given as argument if it is not a CELL or if it is the NULL CELL; otherwise an unused CELL is obtained, its DOWN and RIGHT links are made to point to the destinations of the DOWN and RIGHT links of the argument, and it is returned as the result

COPY.LIST or CPYLST*

Use: copy a (non-cyclic) list or a terminal node

Arguments:

#1: any data node

Results: #1: the data node which was given as argument if it is not a CELL or if it is the NULL CELL; otherwise a copy of the given list is returned after being constructed by linking together unused CELLs and making their DOWN pointers point to the destinations of the DOWN links of the corresponding CELLs of the argument

Notes: If the argument is a cyclic list an error trap occurs.

*This function has another synonym which is considered obsolete: 'COPYLIST'.

101a

COPY.STRUCT or CPYSTR

Use:	copy a (non-cyclic) structure	
Arguments:		
	#1:	any data node
Results:	#1:	a copy of the argument constructed as follows: if the argument is not a CELL or if it is the NULL CELL it is returned; otherwise an unused CELL is obtained, its DOWN link is made to point to a copy of the structure at the destination of the argument's DOWN link, its RIGHT link is made to point to a copy of the structure at the destination of the argument's RIGHT link, and it is returned
Notes:	If the argument is a structure which includes a cycle, an error trap occurs. If the given structure contains any CELL which is pointed to by more than one other CELL of that structure some CELLs will be copied more than once.	

For example, if the following structure is the argument to COPY.STRUCT:

```
+---C     +---C     +---C     +---C
1   1     1   1     1   1     1   1
1   1---->1   1---->1   1---->1** 1
1   1  /->1   1  /->1   1     1   1
+---+  1  +---+  1  +---+     +---+
  1    1    1    1    1
  1    1    1    1    1
  \----/    \----/    1
                      1
                      V
                    +---I
                    1   1
                    1 3 1
                    1   1
                    +---+
```

then the result would be:

```
+---C                                   +---C     +---C     +---C
1   1                                   1   1     1   1     1   1
1   1---------------------------------->1   1---->1   1---->1** 1
1   1                                   1   1     1   1     1   1
+---+                                   +---+     +---+     +---+
  1                                       1         1
  1                                       1         1
  1                                       1         1
  1                                       1         1
  V                                       1         V
+---C               +---C     +---C       1       +---I
1   1               1   1     1   1       1       1   1
1   1-------------->1   1---->1** 1       1       1 3 1
1   1               1   1     1   1       1       1   1
+---+               +---+     +---+       1       +---+
  1                   1                   1
  1                   1                   1
  1                   1                   1
  1                   1                   1
  V                   V                   V
+---C     +---C     +---I               +---C     +---C
1   1     1   1     1   1               1   1     1   1
1   1---->1** 1     1 3 1               1   1---->1** 1
1   1     1   1     1   1               1   1     1   1
+---+     +---+     +---+               +---+     +---+
  1                                       1
  1                                       1
  1                                       1
  1                                       1
  V                                       V
+---I                                   +---I
1   1                                   1   1
1 3 1                                   1 3 1
1   1                                   1   1
+---+                                   +---+
```

Free Storage Management

The AMBIT/L programmer normally does not have to be concerned with the management of free storage. The system automatically invokes the Garbage Collector when necessary to reclaim space resources in the implementation which are no longer in use. In the PDP-10 implementation of AMBIT/L free storage is managed as individual 36-bit storage words. Such space is taken up by CELLs, TOKENs, STRINGs, and those INTEGERs whose values are outside of the range 0 to +32767; INTEGERs whose values are within that range take up no free storage space.

Each CELL and each separately created REAL, LABEL, or FUNCTION node occupies one free storage word. Each TOKEN occupies one word for each constituent in its subname plus two words of overhead. Each STRING occupies one word for each BASIC SYMBOL in its name plus two words of overhead. Each separately created INTEGER whose value is between $-2^{35}+1$ and -1 or between 32768 and $2^{35}-1$ occupies one word.

Each INTEGER whose value is less than $-2^{35}+1$ or greater than $2^{35}-1$ is called a "long integer" in the PDP-10 implementation. Although the AMBIT/L programmer considers all INTEGERs as "atomic" nodes, each long integer is internally represented as a list of INTEGERs interpreted as a number with base 2^{34}. If any arithmetic involves a long integer, then an undetermined amount of free storage may be used. An upper bound may be defined, however, for each separately created long integer: for a given integer take its absolute value and compute the number $\underline{N}$ of "digits" base 2^{34} required to represent it; the long integer occupies at most $2\underline{N}+1$ words. A finer upper bound may be determined by subtracting 1 word for each "digit" (in the representation of the long integer) which is less than 32768. Since some of the internal long integer routines attempt to conserve space, no lower bound on the number of words occupied by a long integer can be given.

When the Garbage Collector is invoked it rings the BELL on the user's terminal and then proceeds to make up a free storage list out of those words in the free storage area which are not accessible by a sequence of links beginning at a POINTER, STRING, or TOKEN. Thus the Garbage Collector frees all CELLs which are not accessible and all words used to represent nodes of other types which are no longer referenced (e.g., a REAL). At present, the Garbage Collector makes no attempt to merge separately created equivalent copies of REALs, LABELs, FUNCTIONs, or INTEGERs.

When garbage collection is complete the system PERM POINTER 'FREE.CT' is made to point DOWN to the INTEGER which represents the number of words of free storage available. Then, if that number is 0 an attempt is made to transfer control indirectly via the system PERM POINTER 'GCOL.CHOKE'; if that POINTER points DOWN to the NULL CELL an error trap occurs; otherwise an "indirect goto" is performed under the assumption that the programmer has set the DOWN link of GCOL.CHOKE to point to a LABEL node corresponding to an appropriate place in his program. Since this is a "goto" rather than a function call it may pop the interpreter control stack in such a way that previously referenced structures are made available for garbage collection.

Although the Garbage Collector is automatically invoked when needed, the AMBIT/L programmer is permitted to invoke a garbage collection at any time by calling the GCOL built-in function.

If a garbage collection was invoked automatically, then after FREE.CT is updated an attempt is made to call a trap function via the system PERM POINTER 'TRAP.GCOL'; if that POINTER points DOWN to the NULL CELL no function call is made.

Note that the POINTER FREE.CT is not updated continuously. It is updated after each garbage collection, and also the programmer is permitted to call at any time the FLTH (for Free LengTH) built-in function which updates FREE.CT.

Garbage collection reclaims space taken up by several node types, but most occupation of free storage is by CELLs. Since garbage collection is costly, three built-in functions are available to the AMBIT/L programmer for controlling the freeing of CELLs (only) either individually, in a list, or in a structure. The use of these functions is optional; correct use can produce significant savings. However, erroneous use can produce terribly obscure bugs since these functions "blindly" return to the free storage list whatever is given them as arguments. Any of these functions may be called during (or just after) a garbage collection choke since none attempt to get a free word. Furthermore, since these functions will accept partially-freed as well as cyclic structures, the careful programmer can cause the freeing of rather entangled structures.

The descriptions of the built-in functions associated with free storage management are now presented.

GCOL

Use: invoke garbage collection of free storage

Arguments:

none

Results: none

Notes: The invocation of the Garbage Collector is normally automatic, but this function provides the programmer with a method for causing a garbage collection to occur. As usual, the system PERM 'FREE.CT' is updated to point DOWN to an INTEGER representing the number of words of free storage after garbage collection. If that number is 0 an attempt is made to transfer

105

control indirectly via the system PERM POINTER 'GCOL.CHOKE'; if that POINTER points DOWN to the NULL CELL an error trap occurs. If there is at least one word of free storage after garbage collection, this function SUCCEEDs. There is no way for it to FAIL.

FLTH

Use: update FREE.CT with the free storage length

Arguments:

none

Results: none

Notes: The system PERM POINTER 'FREE.CT' is updated to point DOWN to an INTEGER representing the number of words of free storage. This function always SUCCEEDs. FREE.CT is also updated after each garbage collection.

FREE.CELL or FRECEL*

Use: free a CELL

Arguments:

#1: any data node

Results: none

Notes: If the argument is a CELL other than the NULL CELL it is rendered free; otherwise, no action takes place. This function always SUCCEEDs.

*This function has another synonym which is considered obsolete: 'FREECELL'.

FREE.LIST or FRELST*

Use: free the CELLs of a list

Arguments:

#1: any data node

Results: none

Notes. If the argument is a CELL other than the NULL CELL, it and all non-NULL non-free CELLs accessible by RIGHT links are rendered free. If an already free CELL is encountered the freeing stops; thus a cyclic list may be given as the argument. If the argument is the NULL CELL or not a CELL no action takes place. This function always SUCCEEDs.

*This function has another synonym which is considered obsolete: 'FREELIST'.

FREE.STRUCT or FRESTR*

Use: free the CELLs of a structure

Arguments:

#1: any data node

Results: none

Notes: If the argument is a CELL other than the NULL CELL, it and all non-NULL non-free CELLs accessible by RIGHT and DOWN links through CELLs are rendered free. If an already free CELL is encountered it stops that particular "walk" as if it were a terminal node; thus a cyclic structure may be given as the argument. If the argument is the NULL CELL or not a CELL no action takes place. This function always SUCCEEDs.

*This function has another synonym which is considered obsolete: 'FREESTRUCTURE'.

Type Transfers

Type transfer or conversion functions permit the AMBIT/L programmer to transform or convert an INTEGER, REAL, STRING, TOKEN, or CELL (a "display") into either an INTEGER, REAL, STRING, TOKEN, or CELL (display). Five functions are available which each accept one of these five types of nodes and (potentially) yield a particular type as follows:

Function Name	Type of Result
TRI	INTEGER
TRR	REAL
TRS	STRING
TRT	TOKEN
TRD	CELL (display)

Thus there are 25 transformations which are individually described in this section. For each of the five functions a description of five transformations is presented according to the type of the given argument. Several of these transformations have questionable utility, but all are implemented for logical completeness. If a node of type BASIC SYMBOL, MARK, LABEL, or FUNCTION is given as argument to a type transfer function an error trap occurs. None of these functions FAIL; if some error condition is detected by a function it causes an error trap to occur.

When a CELL is given as argument to these functions it is interpreted as heading a list of terminal nodes. A terminal node is either the NULL CELL or any data node other than a CELL. The TRD function produces a list of terminal nodes as its result. These lists are called "displays" since they represent the external form of subnames of INTEGERs, REALs, STRINGs, and TOKENs disassembled into their constituent parts. For example, if the TRD function is given the INTEGER -62 as argument, it will produce as a result a "display" of that subname as a list of three BASIC SYMBOLs: %- , %6 , and %2 .

Recall that a STRING has a subname whose constituent parts are BASIC SYMBOLs. Similarly, a TOKEN has a subname whose constituent parts are any terminal nodes.

<u>TRI</u>

Use: <u>tr</u>ansfer to an <u>I</u>NTEGER

Arguments:

#1: an INTEGER, REAL, STRING, TOKEN or CELL

Results: #1: five cases are described according to the type of the argument:

a)	INTEGER	the argument is returned as the result
b)	REAL	the result is the INTEGER which represents the integer part (including sign) of the real number represented by the argument; thus a truncation of the fractional part of the real number is performed
c)	STRING	the display of the given STRING is derived and used as if it had been the argument (see the description of the TRD built-in function)
d)	TOKEN	the display of the given TOKEN is derived and used as if it had been the argument (see the description of the TRD built-in function)

e) CELL the argument is interpreted to be a list of BASIC SYMBOLs which represents a string of characters which represents an integer according to the syntax given below; although the syntax allows for a fractional part and/or an exponent the result is the INTEGER which represents the integer part (including sign) of the number represented by the given string; this same syntax of a number is allowed for an argument to the TRR built-in function

Syntax:*

```
number    →    SP [sgn SP] int [.] {dig} [exp] SP |
               SP [sgn SP] {dig} [.] int [exp] SP |

exp       →    E SP [sgn SP] int

int       →    dig {dig}

sgn       →    + | -

dig       →    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

SP        →    {SPACE | TAB}
```

Examples:

```
-34     34E5
.34     .34 E -5
+ 34.   34.E56
  3.4   3.4E+ 5
```

*In the BNF-like grammar used here underlined words are non-terminals; a pair of square brackets encloses an optional constituent; a pair of curly braces encloses a constituent which may be repeated any number of times (including zero).

TRR

Use: transfer to a REAL

Arguments:

#1: an INTEGER, REAL, STRING, TOKEN, or CELL

Results: #1: five cases are described according to the type of the argument:

a) INTEGER — if the argument represents an integer whose magnitude is not larger than the largest possible real (2^{154} - 2^{127}, which has about 38 decimal digits preceding the decimal point) then the result is the REAL which represents that integer (possibly with rounding if its magnitude is greater than 2^{27}); otherwise an error trap occurs

b) REAL — the argument is returned as the result

c) STRING — the display of the given STRING is derived and used as if it had been the argument (see the description of the TRD built-in function)

d) TOKEN — the display of the given TOKEN is derived and used as if it had been the argument (see the description of the TRD built- in function)

e) CELL the argument is interpreted to be a list of BASIC SYMBOLs which represents a string of characters which represents a real according to the syntax given on the following page; although the syntax allows for highly precise numbers and/or just integers the closest real is determined, and the corresponding REAL is returned unless its value is too large (i.e., its magnitude is greater than $2^{154} - 2^{127}$) in which case an error trap* occurs

*As a temporary measure this function FAILs if the value of the REAL is too large; this "bump" in the design is to accommodate the simple implementation of IAM.

Syntax:*

```
number      →        SP [sgn SP] int [.] {dig} [exp] SP |
                     SP [sgn SP] {dig} [.] int [exp] SP |

exp         →        E SP [sgn SP] int

int         →        dig {dig}

sgn         →        + | -

dig         →        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

SP          →        {SPACE | TAB}
```

```
Examples:            -34       34E5
                     .34       .34 E -5
                     + 34.     34.E56
                       3.4     3.4E+ 5
```

*In the BNF-like grammar used here underlined words are non-terminals; a pair of square brackets encloses an optional constituent; a pair of curly braces encloses a constituent which may be repeated any number of times (including zero).

<u>TRS</u>

Use: <u>tr</u>ansfer to a <u>S</u>TRING

Arguments:

#1: an INTEGER, REAL, STRING, TOKEN, or CELL

Results: #1: five cases are described according to the type of the argument:

a) INTEGER — the display of the given INTEGER is derived and used as if it had been the argument (see the description of the TRD built-in function)

b) REAL — the display of the given REAL is derived and used as if it had been the argument (see the description of the TRD built-in function)

c) STRING — the argument is returned as the result

d) TOKEN — the display of the given TOKEN is derived and used as if it had been the argument (see the description of the TRD built-in function)

e) CELL — the argument is interpreted to be a (possibly null) list of BASIC SYMBOLs, and the STRING node is returned whose subname is a pair of single quotes surrounding the catenation of the characters represented by the BASIC SYMBOLs; if the argument is not a (possibly null) list of BASIC SYMBOLs an error trap occurs

TRT

Use: transfer to a TOKEN

Arguments:

#1: an INTEGER, REAL, STRING, TOKEN, or CELL

Results: #1: five cases are described according to the type of the argument:

a)	INTEGER	the display of the given INTEGER is derived and used as if it had been the argument (see the description of the TRD built-in function)
b)	REAL	the display of the given REAL is derived and used as if it had been the argument (see the description of the TRD built-in function)
c)	STRING	the display of the given STRING is derived and used as if it had been the argument (see the description of the TRD built-in function)
d)	TOKEN	the argument is returned as the result
e)	CELL	the argument is interpreted to be a (possibly null) list of terminal nodes (i.e., data nodes other than non-NULL CELLs), and the TOKEN node is returned whose subname is a matching pair of parentheses surrounding the sequence of subnames of the given terminals

TRD

Use: transfer to the display

Arguments:

#1: an INTEGER, REAL, STRING, TOKEN, or CELL

Results: #1: five cases are described according to the type of the argument:

a) INTEGER the display of the given argument is returned as a list of BASIC SYMBOLs which constitutes the canonical subname of the INTEGER; the canonical subname of an INTEGER is the decimal representation without leading zeros and with a leading minus sign if the integer being represented is negative

b) REAL the display of the given argument is returned as a list of BASIC SYMBOLs which constitutes the canonical subname of the REAL; the canonical subname of a REAL is of the following form, where a "d" is a decimal digit:

d.dddddddE[-][d]d

the matching square brackets enclose an optional constituent; the optional digit cannot be a '0'

c) STRING the display of the given argument is returned as a (possibly null) list of BASIC SYMBOLs which represent the characters of the subname of the STRING except for the surrounding single quotes

d)	TOKEN	the display of the given argument is returned as a (possibly null) list of terminal nodes (i.e., data nodes other than non-NULL CELLs), whose sequence of subnames constitute the subname of the TOKEN when surrounded by a pair of matching parentheses
e)	CELL	the argument is returned as the result

Miscellaneous Functions

Presented here is a variety of functions which don't belong to any of the other classes of built-in functions.

LENGTH

Use:	yield an integer which indicates the length of the argument		
Argument:			
	#1:	an INTEGER, STRING, TOKEN, or CELL	
Results:	#1:	an INTEGER; this function always SUCCEEDs; four cases are described according to the type of the argument:	
	a)	INTEGER	the result is the INTEGER which represents the number of significant bits of storage required to represent the absolute value of the integer represented by the argument; thus leading bits of ZERO are not counted; the length of the integer 0 is 0
	b)	STRING	the result is the INTEGER which represents the number of BASIC SYMBOLs in the display of that STRING or (equivalently) the number of characters in the subname of the STRING excluding the surrounding single quotes
	c)	TOKEN	the result is the INTEGER which represents the number of terminal nodes in the display of that TOKEN or (equivalently) the number of constituents in the sequence of subnames which constitutes the subname of the TOKEN

d)	CELL*	the argument is interpreted as a (possibly null) non-cyclic list, and the result is the INTEGER which represents the number of elements of the list or (equivalently) the number of CELLs in the given list other than the NULL CELL; if the list is cyclic. an error trap occurs

NEXTB

Use: yield the next BASIC SYMBOL

Argument:

#1: a BASIC SYMBOL which represents an ASCII character

Results: #1: if the argument represents the ASCII character DEL (whose octal code is 177) this function FAILs; otherwise it SUCCEEDs and returns the BASIC SYMBOL which represents the next character in the ASCII collating sequence (i.e., whose numeric code is 1 greater)

*This form of the LENGTH function was also described under "List and Structure Processing".

PREVB

Use: yield the previous BASIC SYMBOL

Argument:

#1: a BASIC SYMBOL which represents an ASCII character

Results: #1: if the argument represents the ASCII character NUL (whose octal code is 000) this function FAILs; otherwise it SUCCEEDs and returns the BASIC SYMBOL which represents the previous character in the ASCII collating sequence (i.e., whose numeric code is 1 less)

RANDOM

Use: yield a pseudo-random number

Arguments:

none

Results: #1: a REAL which represents a positive real number less than 1.0

Notes: This function uses two system PERM POINTERs which point DOWN to INTEGERs each corresponding to a 36-bit PDP-10 word: 'P.SEED' and 'P.RAND'. Initially these two POINTERs are initialized to point to the NULL CELL. If RANDOM is called and finds that P.RAND points to the NULL CELL it supplies a base number or seed to P.SEED which is the decimal number 1220703125; that number is also supplied to P.RAND.

The programmer may choose his own seed by initializing both POINTERs, or he may change the seed at any time. Using the default seed produces a cycle (the quantity of random numbers produced before the same sequence reappears) of 8589934592. Each call on RANDOM causes the DOWN link of P.RAND to be updated to point to an INTEGER which represents the product of the previous random number (P.RAND) and the current seed (P.SEED). Then the first 27 bits of the low-order 35 bits of the new random number are used to form a real number which is returned as the result. This function always SUCCEEDs (unless it causes an error trap by finding that P.RAND points DOWN to neither an INTEGER less than 2^{35} or the NULL CELL).

PJOB

Use: yield the job number

Arguments:

none

Results: #1: the INTEGER which represents the user's job number in the PDP-10/50 Time-Sharing System on which the AMBIT/L Programming System is operating; this number may be used to create a unique temporary file name

Notes: This function always SUCCEEDs.

RUNTIME

Use: yield the running time in KCS

Arguments:

none

Results: #1: the INTEGER which represents the number of Kilo-Core-Seconds (KCS) which the running program has used since its execution began; a KCS is the basic unit of cost in a PDP-10/50 Time-Sharing System which represents one second of CPU usage per 1K (1024_{10} words) of core memory occupancy.

Notes: This function always SUCCEEDs.

USER.BREAK

Use: cause a user break into the DAMBIT/L debugger

Arguments:

none

Results: none

Notes: This function puts the interpreter into a state such that at the very next rule-entry in the user's program the DAMBIT/L debugger will gain control. The same effect is obtained by a user's temporarily exiting his AMBIT/L program by typing one or two CTRL C characters and then typing the 'REENTER' command to the PDP-10 Monitor.

AMBIT.EXIT

Use: exit and terminate execution of the AMBIT/L program

Arguments:

none

Results: none

Notes: This function terminates execution of the running AMBIT/L program as if it performed an exit through the outermost block. It is not appropriate to discuss this function's SUCCESS or FAILure. Upon termination, the system indicates on the terminal the number of Kilo-Core-Seconds (KCS) used and the number of seconds of real time used since the program execution began. A KCS is the basic unit of cost in a PDP-10/50 Time-Sharing System which represents one second of CPU usage per 1K (1024_{10} words) of core memory occupancy.

(END)

Section F

AMBIT/L Input/Output

July 22, 1971

This section describes to an AMBIT/L programmer the available built-in functions for performing input/output in the AMBIT/L Programming System.

CONTENTS

This section describes to an AMBIT/L programmer the available built-in functions for performing input/output in the AMBIT/L programming system. There are 19 functions which the programmer may call. At present, each of these functions is written in AMBIT/L and defined in the environment. Many of these functions call upon primitive built-in input/output functions which are written in MACRO-10 assembly language and are not available to the programmer. These primitive functions are described in AMBIT/L Internal Memo 3.

Although the input/output functions are presently written in AMBIT/L the programmer should consider them as purely built-in functions since any substructure of these functions cannot be detected. In the future, these functions may be rewritten entirely in MACRO-10 assembly language completely transparently to the programmer. The only difference which may arise is a change in the space-time characteristics of the AMBIT/L interpreter and therefore, AMBIT/L programs.

All errors detected by the built-in input/output functions are reported by an indirect function call using the TEMP POINTER 'TRAP.IO' declared in the environment. Initially, as a default setting, TRAP.IO points to a FUNCTION node 'FTRAP.IO'. The function FTRAP.IO is a built-in function which reports a trap by typing an indicative error message on the terminal. The programmer is encouraged to provide his own trap function(s), which can be enabled by altering the POINTER TRAP.IO .

The following built-in input/output functions are available to the AMBIT/L programmer:

Function Name	Use
OPEN	initiate input, output, or input-output using a logical name on a physical device
CLOSE	terminate input, output, or input-output for a logical name
DELETE	delete a file on disk
RENAME	change name of a file on disk
INW	input one word
INL	input a line of characters
INLS	input a line of characters and a sequence number
OUTW	output one word
OUTS	output a string of characters
OUTL	output a line of characters
OUTLS	output a line of characters and a sequence number
SELWI	select word for input
SELWO	select word for output
RDSELI	read the input word selector
RDSELO	read the output word selector
RDLNGTH	read the length
RDLNMS	read all logical names
RDINFO	read information associated with a logical name
FTRAP.IO	default I/O trap function which reports an error message

The bulk of this document describes each input/output function in a format appropriate for reference use. First, however, some of the jargon is defined and an overview of the philosophy of AMBIT/L input/output is presented. The major influence of the design was the existing specifications of the DEC PDP-10/50 Monitor.

Definitions

Channel: one of sixteen ports (named 0-15) available to a PDP-10 job for either input, output, or input-output. In the AMBIT/L system, channel 0 is initially assigned for Teletype input-output, and channels 14 and 15 are permanently assigned for private use by the AMBIT/L interpreter. A channel may support at most one physical device at a time.

Physical Device: either a real device, such as the Teletype or Printer or Card Reader; or an individual file on a multi-file device, such as DECtape Unit 5 or the disk. In order for input and/or output to take place on a physical device, it must be assigned to a channel (by the function OPEN).

Logical Name: a MARK or INTEGER associated with a physical device (by the function OPEN) in a particular mode of input/output (either input, output, or input-output).

Mode: one of three ways in which a physical device may be opened with an associated logical name; namely, input, output, or input-output.

Degenerate-List Convention: when an argument of an input/output function is a list of either zero or one argument, it may be given without the CELL. Thus, for example, the third argument of an OPEN function call could be the MARK 'TTY'.

Physical Device Name: a list of one or more MARKS and STRINGS. At present, only two types of physical devices may be specified:

a) the MARK 'TTY' for the Teletype, or
b) one or two STRINGS for the name of a file on disk; if only one STRING is given a null extension is assumed. A legitimate name for a file on disk consists of one to six alphanumeric characters and an extension of zero to three alphanumeric characters.

Mode Name: a node which names a mode:

a) the MARK 'IN' or INTEGER '1' names mode input,
b) the MARK 'OUT' or INTEGER '2' names output, and
c) the MARK 'INOUT' or INTEGER '3' names mode input-output.

Sequence Number: five decimal digits at the beginning of a text line normally followed by a TAB. A disk file is said to be sequenced if each line has a sequence number. Sequenced files are required by some PDP-10 editing programs. A sequence number is implemented with an indicator bit which is used to distinguish it from the text line.

Opening and Logical Names

As previously explained, channel 0 is initially assigned for Teletype input-output, and channels 14 and 15 are permanently assigned for private use by the AMBIT/L interpreter. This implies at most 13 other different physical devices may be simultaneously opened for input and/or output. However, the DELETE and RENAME functions each require that there be at least one open channel for proper execution; but the channel is available after the function returns. The functions OPEN, DELETE, and RENAME always use the numerically lowest channel available. The programmer need not be aware of channels and channel numbers, except when approaching the limit of 13 available ones.

In AMBIT/L input/output, a physical device may be open on only one channel at a time in at most one mode. The OPEN function takes three arguments: a logical name, a mode name, and a physical device name. The first call upon

OPEN dictates the mode in which the physical device is opened on a channel. A later call upon OPEN for the same physical device, and with a different logical name does not cause a new channel to be opened; instead, the same channel is used. Each call upon OPEN also includes a mode specification which remains associated with the logical name also supplied in that function call. The programmer must not violate the following two rules when performing multiple openings of the same physical device:

a) The mode specified in a later OPEN may be the same as the mode specified when the physical device was initially opened, or
b) Any mode may be specified in a later OPEN if the physical device was initially opened in input-output mode

The single argument of the CLOSE function is a logical name. When CLOSE is called the associated physical device is closed on its channel only if no other logical names are associated with the physical device.

Word and Line Input/Output

Ultimately, the basic unit of data for input or output is the 36-bit word. The programmer may call the INW function to input one word, and the OUTW function may be called to output one word. More often, however, the AMBIT/L programmer deals with ASCII characters, particularly for Teletype input and output. The INL function may be called to input a line of characters, and the OUTL function may be called to output a line of characters. A few other functions are also available for similar character handling. Although a programmer will normally make use of a file uniformly as either words or characters, the input/output functions operate so that both types of data input/output may be intermixed. For this reason the descriptions of the data input/output functions first present the simpler explanation of uniform input/output, and non-uniform input/output is explained separately.

Files on Disk

A file on disk is viewed as being composed of a one-way potentially infinite number of 36-bit words, numbered 1, 2, 3, etc. At any time, an initial set of these words (possibly none) are considered to exist with meaningful values, and the remaining are considered non-existent. Each file has one input word selector and one output word selector which each select a word (possibly non-existent) of the file. Normally, the input word selector selects the next word to be inputted, and the output word selector selects the next word to be outputted. The programmer is given the power to interrogate and also set these selectors as follows:

a) The input selector of a file may be interrogated and/or set through a logical name which is opened for that file with an associated mode of either input or input-output.

b) The output selector of a file may be interrogated through a logical name which is opened for that file with an associated mode of either output or input-output.

c) The output selector of a file may be set through a logical name which is opened for that file with an associated mode of input-output.

Therefore, note that a file on disk must have been opened initially for input-output in order for "random-access" output to be performed to that file.

An attempt to input a non-existent word will cause the input function call to FAIL. Any output operation may be viewed as overwriting any previously existing word, or extending the length of the file by one or more existing words. A file may be extended by many existing words when a word is outputted after the output word selector has been advanced into the non-existont portion of the file. In this case, all words before the word just outputted which were non-existent are all made to exist with a value of zero.

The <u>length</u> of a file on disk is the number of existent words (possibly zero).

<u>Teletype Input/Output</u>

Although the Teletype is a character device, the AMBIT/L input/output functions treat the Teletype in the same way as the disk. Namely, an input word selector and output w[illegible]elector are maintained as well as a length. This compatibility has been provided mainly for debugging purposes so that a program which normally uses the disk in a "random-access" manner could alternatively perform such input/output on the Teletype.

The Teletype input functions are implemented using the ASCII LINE mode of the DEC PDP-10/50 Monitor. Thus the normal typing conventions apply, such as the use of RUBOUT to delete the previous character. The following list indicates those characters which are interpreted as break characters:

a)	CR	
b)	FF	(↑L)
c)	VT	(↑K)
d)	ESC	(or ALT)
e)	↑Z	(SUB)

In AMBIT/L Teletype character input the ↑Z character is interpreted as an end-of-file as if it were a non-existent word of a file on disk. The ↑Z character itself (octal 032) is not passed to the AMBIT/L program, nor is any partial line of characters which may have been typed in preceding the ↑Z. Instead the call upon the INL (or INLS) function fails when the line being read ends in ↑Z. The next call upon one of these functions, however, will perform independently of the condition, i.e. the Teletype input acts as if it were immediately re-opened. For example, if the user wishes to cause two consecutive calls upon INL to fail due to an end-of-file condition, he must type ↑Z twice.

Note ↑Z acts as any other character (i.e. does not terminate a line) when read from a file on disk. In the rare case that Teletype word input is used, ↑Z acts as a break character, it does get passed to the programmer as part of a word, and it is not interpreted as an end-of-file.

Input/Output of CR

In AMBIT/L input/output the BASIC SYMBOL %CR represents a "new line" character, which represents the characters CR (octal 015) and LF (octal 012).

On Teletype input the DEC PDP-10/50 Monitor echos and also places in the input stream a LF character immediately following any CR typed in. Any disk file created by common methods also has a LF following every CR. Therefore, AMBIT/L character input functions always read and ignore the character immediately following a CR. Thus a line which ends in CR/LF is presented to the AMBIT/L program terminated by %CR. However, if a lone LF is encountered in an input line it is treated as any other text character, thus appearing as %LF.

Consistently, AMBIT/L character output functions interpret %CR as both CR and LF characters for output to the Teletype and to a disk file,

Input/Output of ESC

Although the PDP-10/50 Monitor standardizes on a code of octal 175, AMBIT/L input/output reduces ESC or ALT characters to a code of octal 033. On character input code 175 is internally translated into code 033. On character output AMBIT/L will generate code 033.

I/O Traps

Some of the AMBIT/L built-in input/output functions fail in order to indicate an unusual condition. Any condition which is considered an error, however, causes an I/O trap which is an indirect function call through the POINTER TRAP.IO of the following form:

```
+----------------------F      +-------C
1                      1      1       1
1        @TRAP.IO      1----->1*RESULT1
1                      1      1       1
+======================+      +-------+
  1        1         1
  1        1         1
  1        1         1
  1        1         1
  V        V         V
+-----S  +------S  +---C     +---C
1     1  1      1  1   1     1   1
1*NAME1  1*ERROR1  1   1---->1   1---->  . . .
1     1  1      1  1   1     1   1
+-----+  +------+  +---+     +---+
                     1         1
                     1         1
                     1         1
                     1         1
                     V         V
                   +-----+   +-----+
                   1     1   1     1
                   1*ARG11   1*ARG21
                   1     1   1     1
                   +-----+   +-----+
```

where *NAME is the name of the built-in input/output function;
*ERROR is a short indicative diagnostic of the error;
*ARG1, *ARG2, etc. are the arguments which were used to call the function; and
*RESULT dictates what the input/output function should do next:

a) If it is the NULL CELL, or any other CELL which points down to the NULL CELL, then the input/output function gracefully does nothing, but succeeds as if it had performed properly. Those input/output functions with results will deliver default results. The default results are indicated with the description of each function. Failure of the trap function is equivalent to its returning the NULL CELL.

b) Otherwise, *RESULT is a list of alternate arguments for the input/output function (in the same form as the third argument of the I/O trap function). Another attempt to execute the input/output function will be made if the I/O trap function returns with its result in this form.

The function called to service an I/O trap may be supplied by the programmer; it may intercept any trap and then perform some computation which could remove the cause of the trap. For example, if an attempt to OPEN causes a 'CHANNELS FULL' trap, a programmer-supplied I/O trap routine could CLOSE some physical device and then return with the same list of arguments so that OPEN would then succeed.

Initially, as a default setting, the POINTER TRAP.IO points to a FUNCTION node FTRAP.IO. The function FTRAP.IO is an AMBIT/L built-in function which is equivalent to the AMBIT/L function definition on the next page.

Note that after FTRAP.IO calls upon OUTL to type the error message it calls USER.BREAK and returns to its caller. When it returns, it returns as result the NULL CELL which causes its caller to gracefully do nothing. The USER.BREAK built-in function causes the DAMBIT/L Debugger to gain control at the next rule-enter in the user program.

```
$
$
$
FTRAP.IO(FUNC TYPE ARGS) RES :
RULE


     +-------------F     +-------------F
     1             1     1             1
     1    OUTL     1ZZZZ>1 USER.BREAK  1
     1             1     1             1
     +=============+     +=============+
       1         1
       1         1
       1         1
       1         1
       V         V
     +-------M +---+     +---+     +---+               +---+     +---+
     1       1 1   1     1   1     1   1               1   1     1   1
     1LISTING1 1   1---->1   1---->1   1--------------->1   1---->1   1
     1       1 1   1     1   1     1   1               1   1     1   1
     +-------+ +---+     +---+     +---+               +---+     +---+
                1          1         1                   1         1
                1          1         1                   1         :
        /-------/          1         1                   1         1
        1                  1         1                   :         1
        V                  V         V                   V         V
     +------------S      +-----A   +--------S          +-----A   +---B
     1            1      1     1   1        1          1     1   1   1
     1'I/O TRAP: '1      1@TYPE1   1' IN %''1          1@FUNC1   1%' 1
     1            1      1     1   1        1          1     1   1   1
     +------------+      +-----+   +--------+          +-----+   +---+

$
$
$
$
```

If the programmer wants more than the FTRAP.IO function provides, he may write his own I/O trap function(s). As indicated previously, such a function should have three arguments and one result. In order to activate an alternate I/O trap function, the programmer need only set the POINTER TRAP.IO to point to a FUNCTION node whose name is the name of the alternate function. Thus the programmer may provide a variety of I/O trap functions for various recovery procedures. Of course, the programmer may employ the default I/O trap function FTRAP.IO at any time. Also, the programmer is free to call FTRAP.IO as any other built-in function.

List of Errors

The second argument of the I/O trap function is a STRING which is a short indicative diagnostic of the error which caused the trap. This section lists all of these STRINGS; with each one is a list of those functions which might detect such an error and a more complete explanation of the cause of the trap. The list of STRINGS is presented in alphabetical order.

'ALREADY OPEN'

Detected by:	OPEN
Explanation:	an attempt is made to open an already opened physical device in an inconsistent mode.

'ALREADY WRITING'

Detected by:	OPEN
Explanation:	an attempt is made to open a physical device for output or input-output which is already being written or renamed by another job.

'BASIC I/O ERROR'

Detected by: CLOSE, DELETE, RENAME, INW, INL, INLS, OUTW, OUTS, OUTL, OUTLS

Explanation: a data transmission error has occured or there is a malfunction of the monitor or hardware.

'CANNOT OUTPUT'

Detected by: OUTW, OUTS, OUTL, OUTLS

Explanation: the second argument of the function being called is not of proper form.

'CHANNELS FULL'

Detected by: OPEN, DELETE, RENAME

Explanation: all available 14 channels are in use, and another is needed.

'INCONSISTENT'

Detected by: OPEN, DELETE, RENAME

Explanation: there is an internal inconsistency, probably due to an error in implementation.

'IS OPEN'

Detected by: DELETE, RENAME.

Explanation: the argument of DELETE or the second argument of RENAME (old name) specifies an already open physical device.

'LOG. NAME IN USE'

Detected by: OPEN

Explanation: the first argument is a logical name which is already in use.

'NAME IN USE'

Detected by: RENAME

Explanation: the first argument (new name) already exists in the user's directory.

'NOT DEVICE'

Detected by: OPEN

Explanation: the third argument is not a proper physical device name.

'NOT FILE NAME'

Detected by:	DELETE, RENAME
Explanation:	the argument of DELETE or one of the arguments of RENAME is not a proper physical device name for a file on disk.

'NOT LOG. NAME'

Detected by:	OPEN, CLOSE, INW, INL, INLS, OUTW, OUTS, OUTL, OUTLS, SELWI, SELWO, RDSELI, RDSELO, RDINFO
Explanation:	the first argument is not a MARK or INTEGER, i.e. it is not a logical name.

'NOT MODE'

Detected by:	OPEN
Explanation:	the second argument is not MARK IN, OUT, or INOUT nor INTEGER 1, 2, or 3, i.e. it is not a mode name.

'NOT PERMITTED'

Detected by:	CLOSE, INW, INL, INLS, OUTW, OUTS, OUTL, OUTLS, SELWI, SELWO, RDSELI, RDSELO, RDINFO
Explanation:	the first argument is a logical name not assigned to a physical device or it is assigned in a mode which does not permit the attempted operation.

'NOT SEQ..NO.'

Detected by:	OUTLS
Explanation:	the third argument is not the INTEGER -1 or a positive INTEGER, i.e. it is not a sequence number.

'NOT WORD NO.'

Detected by:	SELWI, SELWO
Explanation:	the second argument is not an INTEGER greater than 0, i.e. it is not a word number.

146

Descriptions of the I/O Functions

Each I/O function is described with allowable arguments and possible results, an example of a function call, traps which may occur (in order of detection), conditions for FAILure (if appropriate), and default results (if appropriate).

OPEN

Use: initiate input, output, or input-output using a logical name on a physical device (see the section "Opening and Logical Names").

Arguments:

#1: a logical name

#2: a mode name

#3: a physical device name

Example:

```
+----------------------F
1                      1
1         OPEN         1
1                      1
+======================+
   1        1        1
   1        1        1
   1        1        1
   1        1        1
   V        V        V
+----M    +---M    +---M
1    1    1   1    1   1
1LSTG1    1OUT1    1TTY1
1    1    1   1    1   1
+----+    +---+    +---+
```

Traps:

NOT LOG. NAME

NOT MODE

NOT DEVICE

LOG. NAME IN USE

CHANNELS FULL

ALREADY OPEN

ALREADY WRITING

INCONSISTENT

Conditions for FAILure:

a) A disk file is specified as the physical device to be opened for input, and either the file doesn't already exist or it already exists and is read-protected.

b) A disk file is specified as the physical device to be opened for output, and the file already exists and is write-protected.

c) A disk file is specified as the physical device to be opened for input-output, and the file already exists and is either read-protected or write-protected.

Description:

Case 1: Open for input

a) If the given physical device is not already open, then open a new channel. Set the input word selector to 1. If the given physical device is a disk file determine the length. If it is the TTY assume a length of 0.

b) If the given physical device is already open for either input or input-output then associate the given logical name and given mode with the device.

Case 2: Open for output

a) If the given physical is a disk file which already existed and is not in use by other jobs for output or input-output, then the old version is either deleted or marked for later deletion after all read references by other jobs are completed.

b) If the given physical device is not already open, then open a new channel. Set the output word selector to 1 and assume a length of 0. If it is a disk file this implies creation of a new file.

c) If the given physical device is already open for either output or input-output associate the given logical name and given mode with the device.

Case 3: Open for input-output

a) If the given physical device is not already open, then open a new channel. If it is the TTY, set the input word selector to 1, set the output word selector to 1, and assume a length of 0.

b) If the given physical device is a disk file which did not already exist, create it, set the input word selector to 1, set the output word selector to 1, and assume a length of 0.

c) If the given physical device is a disk file which already existed, but is not open, determine the length, set the input word selector to 1, and set the output word selector to one more than the length of the file.

d) If the given physical device is already open for input-output, then associate the given logical name and given mode with the device.

CLOSE

Use: terminate input, output, or input-output for a logical name (see the section "Opening and Logical Names").

Arguments:

#1: a logical name associated with an open physical device

Example:

```
+------------F
1            1
1    CLOSE   1
1            1
+============+
  1
  1
  1
  1
  V
+----M
1    1
1LSTG1
1    1
+----+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

BASIC I/O ERROR

Description:

Terminate use of the given logical name for its associated physical device. If no other logical names are associated with that physical device, close the channel being used and return it to the available pool.

DELETE

Use: delete a file on disk

Arguments:

#1: a physical device name for a file on disk

Example:

```
+-------------F
1             1
1   DELETE    1
1             1
+=============+
   1
   1
   1
   1
   V
+---+     +---+
1   1     1   1
1   1---->1   1
1   1     1   1
+---+     +---+
   1         1
   1         1
   1         1
   1         1
   V         V
+-------S +-----S
1       1 1     1
1'JUNK'1  1'REL'1
1       1 1     1
+-------+ +-----+
```

Traps:

NOT FILE NAME

CHANNELS FULL

IS OPEN

BASIC I/O ERROR

INCONSISTENT

Conditions for FAILure:

The file does not exist.

Description:

The given file is deleted or it is marked for later deletion after all read references. by other jobs are completed.

<u>RENAME</u>

Use: change name of a file on disk

Arguments:

#1: a <u>new</u> physical device name for a file on disk

#2: an <u>old</u> physical device name for a file on disk

Example:

```
+----------------------------F
1                            1
1          RENAME            1
1                            1
+============================+
   1                      1
   1                      1
   1                      1
   1                      1
   V                      V
+----+     +----+       +--------S
1    1     1    1       1        1
1    1---->1    1       1'GARBAG'1
1    1     1    1       1        1
+----+     +----+       +--------+
   1          1
   1          1
   1          1
   1          1
   V          V
+------S   +-----S
1      1   1     1
1'JUNK'1   1'REL'1
1      1   1     1
+------+   +-----+
```

Traps:

NOT FILE NAME

CHANNELS FULL

IS OPEN

NAME IN USE

BASIC I/O ERROR

INCONSISTENT

Conditions for FAILure:

The file does not exist or it is read-protected.

Description:

The given old file is renamed with the given new file name. If the two names are effectively the same (employing the degenerate-list convention) renaming does not occur.

INW

Use: input one word

Arguments:

#1: a logical name associated with mode input or input-output for an open physical device

Results:

#1: an INTEGER

Example:

```
                     +------P
                     1      1
                     1VALUE1
                     1      1
                     +------+
                       H
                       H
                       H
                       H
                       V
 +-------------F     +---+
 1             1     1   1
 1     INW     1---->1   1
 1             1     1   1
 +=============+     +---+
   1
   1
   1
   1
   V
+------------M
1            1
1SYMBOL.TABLE1
1            1
+------------+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

BASIC I/O ERROR

Conditions for FAILure:

a) The physical device is a disk file, and the input word selector points to a non-existent word.

b) The physical device is the TTY, and the user has typed in ↑Z (see the section "Teletype Input/Output").

Default Results:

#1: the INTEGER 0

Description:

The next word of the input stream is delivered, and the input word selector is advanced by one. If the physical device is the TTY and all typed input has been read previously, the function waits for the user to type more (up to a break character).

Additional Description for Non-Uniform Input/Output:

Beware that if either INL or INLS was previously preformed to input a line of characters which did not occupy an integral number of words, then any characters of the partial word will be lost. Note, however, that any disk file produced via AMBIT/L input/output functions will always pad out a line with NULL characters to occupy an integral number of words.

INL

Use: input a line of characters

Arguments:

#1: a logical name associated with mode input or input-output for an open physical device.

Results:

#1: a list of BASIC SYMBOLS, terminating with either %CR, %ESC, %VT (vertical tab), or %FF (form feed)

Example:

```
+-------------F     +---+     +---+
1             1     1   1     1   1
1     INL     1---->1   1---->1   1
1             1     1   1     1   1
+=============+     +---+     +---+
  1                   1         A
  1                   1         H
  1                   1         H
  1                   1         H
  V                   V         H
+---I               +---+       H
1   1               1   1       H
1 5 1               1   1       H
1   1               1   1       H
+---+               +---+       H
                      A         H
                      H         H
               ...   .H         H
                      H         H
                      H         H
                    +-----P   +----P
                    1     1   1    1
                    1FIRST1   1REST1
                    1     1   1    1
                    +-----+   +----+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

BASIC I/O ERROR

Conditions for FAILure:

a) The physical device is a disk file, and an attempt to read a non-existent word occurred while developing an input line. The input word selector has been advanced pointing to the non-existent word.

b) The physical device is the TTY, and the user has typed in a line (possibly null) ending in ↑Z (see the section "Teletype Input/Output"). The input selector has been advanced by one plus one for every five characters typed in preceding the ↑Z.

Default Results:

#1: the NULL CELL

Description:

If a previous call upon INL or INLS caused a line to be read which did not account for an integral number of full words, then this call begins reading the very next character of the partial word (unless the SELWI function has been called meanwhile). Any sequence number appearing in the line being inputted is ignored, and the very next character (should be TAB) is ignored. A line of arbitrary length is read terminated by either CR (see the section "Input/Output of CR"), ESC (see the section "Input/Output of ESC"), VT, or FF characters. Any NULL character is ignored. The input word selector is advanced selecting the next full word following the last character of the line. If the physical device is the TTY and all typed input has been read previously, the function waits for the user to type more (up to a break character).

If INL calls an input/output trap function it presents as first argument the STRING 'INLS' rather than 'INL'. This "bump" in the design was made to optimize the time and space required for these character input functions.

INLS

Use: input a line of characters and a sequence number

Arguments:

#1: a logical name associated with mode input or input-output for an open physical device

Results:

#1: a list of BASIC SYMBOLS, terminating with either %CR, %ESC, %VT (vertical tab), or %FF (form feed)

#2: either the INTEGER -1 indicating a line with no sequence number (always for TTY), or a positive INTEGER which is the sequence number

Example:

```
+----A             +----P  +---P
1    1             1    1  1   1
1@DEV1             1LINE1  1SEQ1
1    1             1    1  1   1
+----+             +----+  +---+
  A                 H       H
  1                 H       H
  1                 H       H
  1                 H       H
  1                 V       H
+------------F     +---+    H
1            1     1   1    H
1            1---->1   1    H
1            1. .. 1.  1    H
1            1     +---+    H
1            1              H
1            1              H
1   INLS     1              H
1            1              H
1            1              V
1            1             +---+
1            1             1   i
1            1------------>1   1
1            1             1   1
+============+             +---+
```

Traps:
NOT LOG. NAME
NOT PERMITTED
BASIC I/O ERROR

Conditions for FAILure:

a) The physical device is a disk file, and an attempt to read a non-existent word occurred while developing an input line. The input word selector has been advanced pointing to the non-existent word.

b) The physical device is the TTY, and the user has typed in a line (possibly null) ending in ↑Z (see the section "Teletype Input/Output"). The input selector has been advanced by one plus one for every five characters typed in preceding the ↑Z.

Default Results:

#1:	the NULL CELL
#2:	the INTEGER -1

Description:

If a previous call upon INL or INLS caused a line to be read which did not account for an integral number of full words, then this call begins reading the very next character of the partial word (unless the SELWI function has been called meanwhile). A line of arbitrary length is read terminated by either CR (see the section "Input/Output of CR"), ESC (see the section "Input/Output of ESC"), VT, or FF characters. Any NULL character is ignored. If the line contains a sequence number, it is read and the very next character (should be TAB) is ignored. If the line has more than one sequence number, only the last one is reported. The input word selector is advanced selecting the next full word following the last character of the line. If the physical device is the TTY and all typed input has been read previously, the function waits for the user to type more (up to a break character).

OUTW

Use: output one word

Arguments:

#1: a logical name associated with mode output or input-output for an open physical device

#2: an INTEGER

Example:

```
+-------------F
1             1
1     OUTW    1
1             1
+=============+
  1         1
  1         1
  1         1
  1         1
  V         V
+----M    +---I
1    1    1   1
1DUMP1    1-1 1
1    1    1   1
+----+    +---+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

CANNOT OUTPUT

BASIC I/O ERROR

Description:

The given word is outputted, and the output word selector is advanced by one. If the physical device is the TTY, five ASCII characters (except NULL is ignored) are placed into the Teletype output buffer, which is flushed by a call on OUTW only when the buffer is filled to its 16-word capacity. The user should beware of outputting ↑D and other troublesome characters to the TTY.

OUTS

Use: output a string of characters

Arguments:

#1: a logical name associated with mode output or input-output for an open physical device

#2: a list o: possibly varied content: each element may be either a BASIC SYMBOL, STRING, INTEGER, REAL, or NULL CELL; the degenerate-list convention applies

Example:

```
+-------------F
1             1
1     OUTS    1
1             1
+=============+
  1         1
  1         1
  1         1
  1         1
  V         V
+---M     +---B
1   1     1   1
1TTY1     1%* 1
1   1     1   1
+---+     +---+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

CANNOT OUTPUT

BASIC I/O ERROR

Description:

Each element of the second argument is replaced by a uniform list of BASIC SYMBOLS by applying the AMBIT/L built-in function TRD to each STRING, INTEGER, and REAL, and each NULL element is eliminated.

The resulting list is outputted using an integral number of words by padding out unused bytes with NULL characters. The output word selector is advanced accordingly. If the physical device is the TTY, the Teletype output buffer is flushed so that all characters are typed (even if this call on OUTS did not contribute any characters for output).

<u>OUTL</u>

Use: output a list of characters

NOTE: The characteristics of OUTL are the same as those of OUTS, except effectively a BASIC SYMBOL %CR is always appended to the end of the list of elements of the second argument.

OUTLS

Use: output a line of characters and a sequence number

Arguments:

#1: a logical name associated with mode output or input-output for an open physical device

#2: a list of possibly varied content: each element may be either a BASIC SYMBOL, STRING, INTEGER, REAL, or NULL CELL; the degenerate-list convention applies

#3: either the INTEGER -1 indicating a line with no sequence number, or a positive INTEGER which is the sequence number to be outputted.

Example:

```
+---------------------F
1                     1
1        OUTLS        1
1                     1
+=====================+
  1       1         1
  1       1         1
  1       1         1
  1       1         1
  V       1         V
+------M  1       +---I
1      1  1       1   1
1OUTPUT1  1       11001
1      1  1       1   1
+------+  1       +---+
          1
          1
          1
          1
          1
          V
        +---+     +---+     +---+
        1   1     1   1     1   1
        1   1---->1   1---->1   1
        1   1     1   1     1   1
        +---+     +---+     +---+
          1         1         1
          1         1         1
          1         1         1
          1         1         1
          V         V         V
        +-------S +---B     +----I
        1       1 1   1     1    1
        1'HELLO'1 1%  1     119701
        1       1 1   1     1    1
        +-------+ +---+     +----+
```

Traps:
NOT LOG. NAME
NOT PERMITTED
CANNOT OUTPUT
NOT SEQ. NO.
BASIC I/O ERROR

Description:

The characteristics of OUTLS are the same as those of OUTL when the given third argument is the INTEGER -1. When the given third argument is a positive INTEGER the characteristics of OUTLS are the same as those of OUTL, except the line being outputted begins with a sequence number and TAB.

SELWI

Use: select word for input, i.e. set the input word selector

Arguments:

#1: a logical name associated with mode input or input-output for an open physical device

#2: an INTEGER greater than 0

Example:

```
+-------------F
1             1
1    SELWI    1
1             1
+=============+
  1         1
  1         1
  1         1
  1         1
  V         V
+------M  +---I
1      1  1   1
1SYMTBL1  1 1 1
1      1  1   1
+------+  +---+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

NOT WORD NO.

Description:

The input word selector is set to the value of the given second argument.

Additional Description for Non-Uniform Input/Output:

Beware that if either INL or INLS was previously performed to input a line of characters which did not occupy an integral number of words, then any characters of the partial word will be lost. Note, however, that any disk file produced via AMBIT/L input/output functions will always pad out a line with NULL characters to occupy an integral number of words.

SELWO

Use: select word for output, i.e. set the output word selector

Arguments:

#1: a logical name associated with mode input-output for an open physical device.

#2: an INTEGER greater than 0

Example:

```
+-------------F
1            1
1    SELWO   1
1            1
+=============+
  1         1
  1         1
  1         1
  1         1
  V         V
+------M  +------A
1      1  1      1
1SYMTBL1  1@VALUE1
1      1  1      1
+------+  +------+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

NOT WORD NO.

Description:

The output word selector is set to the value of the given second argument.

RDSELI

Use: read the input word selector

Arguments:

#1: a logical name associated with mode input or input-output for an open physical device.

Results:

#1: an INTEGER greater than 0

Example:

```
                 +---P
                 1   1
                 1NOW1
                 1   1
                 +---+
                   H
                   H
                   H
                   H
                   V
+-----------F    +---+
1           1    1   1
1   RDSELI  1---->1   1
1           1    1   1
+===========+    +---+
  1
  1
  1
  1
  V
+------M
1      1
1SYMTBL1
1      1
+------+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

Default Results:

#1: the INTEGER 0

Description:

The result is set to the current value of the input word selector.

RDSELO

Use: read the output word selector

Arguments:

#1: a logical name associated with mode output or input-output for an open physical device

Results:

#1: an INTEGER greater than 0

Example:

```
+-------------F     +---I
1             1     1   1
1   RDSELO    1---->1 1 1
1             1     1   1
+=============+     +---+
  1
  1
  1
  1
  V
+-----M
1     1
1FILE11
1     1
+-----+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

Default Results:

#1: the INTEGER 0

Description:

The result is set to the current value of the output word selector.

RDLNGTH

Use: read the length

Arguments:

#1: a logical name for an open physical device

Results:

#1: a positive INTEGER

Example:

```
+-------------F      +---+
1             1      1   1
1   RDLNGTH   1----->1   1
1             1      1   1
+=============+      +---+
  1                    A
  :                    1
  1                    1
  1                    1
  V                    1
+--------A           +----F    +---+
1        1           1    1    1   1
1@TEMFILE1           1ADD11--->1   1
1        1           1    1    1   1
+--------+           +====+    +---+
  A                              A
  1                              1
  1                              1
  1                              1
  1                              1
+--------------------------------F
1                                1
1              SELWO             1
1                                1
+================================+
```

Traps:

NOT LOG. NAME

NOT PERMITTED

Default Results:

#1: the INTEGER 0

Description:

The result is set to the current length of the physical device. Note this is also defined for the TTY (see the sections "Files on Disk" and "Teletype Input/Output").

RDLNMS

Use: read all logical names

Results:

#1: a list of all logical names

Example:

```
                              +----P
                              1    1
                              1SIZE1
                              1    1
                              +----+
                                H
                                H
                                H
                                H
                                V
                    +------F  +---+
                    1      1  1   1
                    1LENGTH1->1   1
                    1      1  1   1
                    +======+  +---+
                      1
                      1
                      1
                      1
                      V
+-------------F     +---+
1             1     1   1
1   RDLNMS    1---->1   1
1             1     1   1
+=============+     +---+
```

Description:

The result is set to a created list of all logical names of currently open devices. The list is ordered, with the most recently opened name last. Note that the degenerate list convention is not used for the result.

RDINFO

Use: read information associated with a logical name

Arguments:

#1: a logical name for an open physical device

Results:

#1: a mode name in INTEGER form, i.e. 1, 2, or 3

#2: a physical device name, i.e. a list

#3: a channel number, i.e. an INTEGER 0, 1, 2, ..., or 13

#4: a list of all logical names associated with the physical device associated with the given argument.

Example:

```
+-------------F     +---+
1             1     1   1
1             1---->1   1
1             1     1   1
1             1     +---+
1             1
1             1
1             1
1             1
1             ;
1             1     +---+     +----P
1             1     1   1     1    1
1   RDINFO    1---->1   1     1CHAN1
1             1     1   1     1    1
1             1     +---+     +----+
1             1                H
1             1                H
1             1        /=======/
1             1        H
1             1        V
1             1     +---+
1             1     1   1
1             1---->1   1
1             1--\  1   1
+=============+  1  +---+
  1              1
  1              1
  1              1
  1              1
  V              1
+----M           1  +---+
1    1           \->1   1
1LSTG1              1   1
1    1              1   1
+----+              +---+
```

166

Traps:

NOT LOG. NAME
NOT PERMITTED

Default Results:

#1:	the INTEGER 0
#2:	the NULL CELL
#3:	the INTEGER 0
#4:	the NULL CELL

Description:

The four results are set as indicated above. Note that the degenerate list convention is not used for the second and fourth results. Also note the second result is not a created list and thus should not be altered by the programmer.

(END)

Section G

Using DIAGEN: the AMBIT/L Diagram Generator

January 10, 1972

This section describes how to use the AMBIT/L Diagram Generator.

The Diagram Generator (DIAGEN) is a translator in the AMBIT/L Programming System which reads as input an encodement of one insertion (or block) of an AMBIT/L program and produces as output a listing of that insertion. This kind of translation is often performed by the compiler of a programming system, and such an organization would make sense in the case of AMBIT/L. Historically, however, DIAGEN has been written in FORTRAN and MACRO - 10 assembly language, and merging DIAGEN and the AMBIT/L Compiler (which is written in AMBIT/L) would require the establishment of the desire or need on the part of the AMBIT/L user community. The current separation of these two translators decreases their combined effectiveness in reporting errors; this will be elaborated upon later.

Since the format of the listing produced is described in Section C, "The Drawing of AMBIT/L Programs and Their Encodement", this section is concerned more with the actual use of DIAGEN.

The Diagram Generator is invoked by a Monitor command of the form:

```
RUN DIAGEN [ proj , prog ]
```

where proj and prog are the project and programmer numbers of the directory where the AMBIT/L Programming System is residing. This calling sequence may be somewhat different, depending upon the method used to install AMBIT/L on a particular PDP - 10.

When DIAGEN is invoked it first prompts the user by typing an asterisk on a new line. The user is then expected to type the file name of the source file he wishes to list. The name is accepted in standard form: a primary name of one to six alphanumeric characters followed optionally by an extension of zero to three alphanumeric characters with a period as separator. The file name must be immediately terminated by a carriage return. If no period and no extension is provided DIAGEN assumes a default extension of "AL" for AMBIT/L. A null extension name is specified by following the primary name with just a period. Note that there is no option for specifying a project-programmer number; thus the source file must exist in the disk directory in which the user is currently logged in. If there is a syntax error in the name or if the specified file does not exist, DIAGEN informs the user by typing a question mark followed by another prompting asterisk on a new line. Otherwise, a listing file is created in the user's current disk directory with the same primary name as the source file and an extension name of "LST".

DIAGEN then reads the source file and produces the listing file. When it reads a source file with sequence numbers (as produced by some text editors and PIP), it notes the number of the starting line of each rule and includes that number in the listing of the rule. DIAGEN treats a source file without sequence numbers as if it were sequenced by one (starting with 00001 as the first line). A source file may even be mixed in its use and omission of sequence numbers since each line is individually examined for the presence of a sequence number.

All textual material except for blank lines is simply copied from the source file to the listing file. To separate parts of a program listing use comment lines (beginning with a '$'). DIAGEN expects the very first non-blank line of a source file is not the beginning of a rule. It also expects that the last line of a source file includes the 'END' statement not followed immediately by a semicolon. When reading a source file which does not end in this way DIAGEN detects an error condition which is reported as:

ERROR READING SOURCE FILE

and it returns control to the PDP - 10 Monitor; however, the listing file is not lost.

Other than seeking the words 'END' or 'RULE', DIAGEN does not analyze any of the textual portion of the program outside of rules and, therefore, it does not detect or report on any errors in that text. For example, the lack of matching parentheses or BEGIN - END pairs goes unnoticed.

When it finds 'RULE' DIAGEN reads the encodement of the rule up to the terminating semicolon and performs some syntactic analysis on the encodement. As it begins to analyze each rule DIAGEN types one decimal digit on the terminal to inform the user of its progress. As each node specification of the encoded rule is read DIAGEN places that node and its links onto the representation of the listing page it is building. After assimilating the entire rule, the representation is sent to the listing file. If a syntax error is detected within a rule DIAGEN types 'SYNTAX ERROR' followed by the current source line (with sequence or line number) and then a line with an arrow pointing to the character of the source line which triggered the detection of the error. In the case of such an error DIAGEN abandons further analysis of the current rule; the listing includes the initial part of the diagram and then the source text of the encodement from the line of the error to the terminating semicolon.

The syntax accepted by DIAGEN is very permissive. For example, it does not check that the type-set is one of the allowed forms, and most node names are not analyzed. Nodes in a rule which are either inaccessible or missing do not trigger any error condition. Links are drawn only on the basis of their origins and routes (explicit or default); the specified destination of a link is not checked for consistency. This can lead to a hard-to-find bug when a diagram may look good, and the specified destination may not be the correct one; the AMBIT/L Compiler ignores specified routes (except for the initial perburbation) and bases its analysis on the specif.ed destination. USERS BEWARE!

Since DIAGEN diagnoses so few errors, users are urged to examine the listing to notice incorrect or undesirable rule encodement. Such a review of the program prior to its submission to the compiler can be used to catch other syntactic errors or even semantic ones. As long as there are no ambiguities in the specification of a rule, the Compiler does not complain about poor layout.

When DIAGEN completes its translation of a source file, it restarts by again prompting the user with an asterisk on a new line. The user may then specify another source file which he wishes to be translated. If he is finished using DIAGEN he types ↑C (i.e., CTRL C) to return control to the PDP-10 Monitor. To stop the operation of DIAGEN at any time the user may type two ↑C's (CTRL C) on the terminal to abort the current translation and return control to the Monitor.

Then to obtain a listing the user may use the 'TYPE' command to see it immediately on his terminal or he may request a line printer copy by standard methods. In some cases the user may wish to use a text editor to help look at some of the listing. The listing file is unsequenced, i.e. no sequence numbers are attached to lines. The listing includes TAB characters whenever possible corresponding to tab settings at every eight typing positions. Thus if a terminal with hardware tab capability is used as the listing device the user should be sure to take advantage of the possibility for a much quicker listing.

DIAGEN always translates the entire source file it is given. Occassionally the need arises to produce diagrams for only a few rules of a large insertion. The user can use a text editor to produce a temporary file containing only those rules he wishes to diagram. When doing so, it should be recalled that the file must end with an 'END' statement and it must begin with a non-blank line other than a rule's beginning. It is suggested tha an initial line of 'BEGIN' be used.

(END)

Section H

Using COMPIL: the AMBIT/L Compiler

January 10, 1972

This section describes how to use the AMBIT/L Compiler and how to interpret its informative typeout. Included is a collection of all possible error diagnostics along with associated explanations and error recovery transformations made on the source text.

The AMBIT/L Compiler, which is itself an AMBIT/L program, is used to translate one insertion of an AMBIT/L program from a source file of (ASCII) characters into an intermediate binary encodement file. The source file represents both textual and diagrammatic portions of the program in an encodement language whose syntax is presented in Section D, "The Syntax of the Encodement of AMBIT/L Programs". It is prepared usually by a text editing program available on the host PDP - 10 Time-Sharing System, such as EDITOR, TECO, SOS, etc. The primary name of the source file must match the name of the insertion which it contains, except the primary name does not include any periods which might be in the insertion name, and only up to six characters can be used as a primary file name.* The programmer is free to choose any file name extension of zero to three alphanumeric characters, but the Compiler (and also Diagram Generator) expect the default extension "AL" (for AMBIT/L).

The AMBIT/L Compiler is invoked by a Monitor command of the form:

RUN COMPIL [proj , prog]

where proj and prog are the project and programmer numbers of the directory where the AMBIT/L Programming System is residing. This calling sequence may be somewhat different, depending upon the method used to install AMBIT/L on a particular PDP - 10.

When the Compiler is first invoked it prompts the user with the following request:

```
*SOURCE=
```

*This rule need not be strictly followed at compilation time or diagram generation time, but it is a requirement that the primary name of the REL file conforms to this rule when the insertion is linked by the AMBIT/L Link Editor.

The user is then expected to type the file name of the insertion he wishes to translate. The name is accepted in standard form: a primary name of one to six alphanumeric characters followed optionally by an extension of zero to three alphanumeric characters with a period as separator. The file name must be immediately terminated by a carriage return. If no period and no extension is provided, the Compiler assumes a default extension of "AL". A null extension name is specified by following the primary name with just a period. Note there is no option for specifying a project-programmer number; thus the source file must exist in the disk directory in which the user is currently logged in. If there is a syntax error in the name or if the specified file does not exist, the Compiler appropriately informs the user and then repeats the prompting message on a new line.

The source file may optionally include sequence numbers on any or all lines. The Compiler treats each unsequenced line as if it had a sequence number one higher than the previous line; a thoroughly unsequenced source file is treated as if it were sequenced by one, starting with 00001 on the first line. The compiler uses these sequence numbers only for informative typeout on the terminal. Such typeout is done for correct programs as well as for reporting error diagnostics. The Diagram Generator uses the same method for providing default sequence numbers when necessary and thus the user can make the correspondence between the diagrammatic listing and the informative typeout of the compiler.

Except for the typeout on the terminal, the only output of the compiler is a binary file in the user's current disk directory called a "REL" file to conform to the name of intermediate code files produced by other translators on the PDP - 10 (e.g., FORTRAN, MACRO - 10, etc.). Although "REL" is for "relocatable", AMBIT/L intermediate code (or even final interpreter code) really doesn't have anything to do with relocation. Furthermore, the format of an AMBIT/L REL file bears no relation to that of other programming subsystems of the PDP - 10. The REL file produced by the Compiler is given the same primary name as the source file of which it is a translation and an extension name of "REL".

An AMBIT/L REL file is always produced even if several error conditions are detected during compilation. It is also correct or meaningful code since the Compiler performs well-documented error recovery transformations on the given source program specific to each type of error condition. However, in the rare case that the Compiler itself "bombs out" the REL file is lost; there are no known bugs of this sort for either correct or incorrect AMBIT/L programs. Any user encountering such a problem should inform the system maintainer of his trouble, hopefully accompanied by the typeout and by a copy of the source file.

If the user wishes to stop a compilation prematurely, he may do so at any time by typing two ↑C's (CTRL C) on the terminal to return control to the PDP - 10 Monitor.

The Compiler's typeout is useful to the user as a continual indication of its successful progress as it begins to translate each rule of the insertion. It can also be of help during debugging of the program and thus should always be saved. For each rule, one typed line is issued which contains two or more columns. The first column has a decimal number which is the sequence number of the rule in the source file. The second column has a decimal number which indicates the word number in the binary object code being produced where the encodement of that rule begins. Remaining columns are used to indicate any identifiers (if any) which are declared as labels of that rule. A similar typeout is issued for each END statement in the source program. The automatically declared identifiers 'RET' and/or 'EXIT' are listed when appropriate. Each INSERT command in the source program which inserts a function body and each one-rule function body causes one line to be typed indicating that 'RET' is automatically declared. As errors are detected throughout the compilation, appropriate error diagnostics are typed. Also, the bell may ring as compilation proceeds; each ring indicates the

Garbage Collector has been invoked automatically to regain free storage. Such activity is normal and expected.

After typing the line which corresponds to the final END statement, the Compiler types a list of all undeclared identifiers (if any). The user should always look through the list to see if it contains just what he expects. An undeclared identifier should be one which is declared within an enclosing block in some other insertion or is built-in to the user's environment. The resolving of these identifiers is done by the Link Editor. The user should not be surprised to find certain built-in function names in the typed list which are employed as a result of certain macro expansions: MEMBER, EQ, TRT, TRS.

After typing the undeclared identifiers the bell on the terminal rings indicating a forced invocation of the Garbage Collector. Then there is a pause for several seconds while the Compiler appends the symbol table to the REL file. Finally, the Connect Time (in seconds) and the number of Kilo-Core-Seconds of computing performed by the Compiler are typed and control returns to the PDP - 10 Monitor.

Below is a sample program in both diagrammatic and encoded form followed by a typeout of its compilation.

```
INSERTION EXAMPLE;
BEGIN
B(X) Y :
  INSERT B;
C(X)Y:
BEGIN
RULE
(00070)

     +---P     +---P
     I   I     I   I
     I X I     I Y I
     I   I     I   I
     +---+     +---+
       I         H
       I         H
       I         H
       I         H
       V         V
     +---+     +---+
     I   I     I   I
     I   I---->I   I
     I   I     I   I
     +---+     +---+

 SF/RET;
END;
START: RULE
(00140)

     +---T     +---P
     I   I     I   I
     I(Q)I     I R I
     I   I     I   I
     +---+     +---+
       H         H
       H         H
       H         H
       H         H
       V         H
     +---A       H
     I   I       H
     I@P I       H
     I   I       H
     +---+       H
       A         H
       I         H
       I         H
       I         H
       I         V
     +---F     +---+
     I   I     I   I
     I C I---->I   I
     I   I     I   I
     +===+     +---+

 F/ERROR;
END
```

```
00010   INSERTION EXAMPLE;
00020   BEGIN
00030   B(X) Y :
00040     INSERT B;
00050   C(X)Y:
00060   BEGIN
00070   RULE
00080   A1/P/X D/B1,
00090   A2/P/Y BD/B2,
00100   B1 R/B2,
00110   B2//
00120   SF/RET;
00130   END;
00140   START: RULE
00150   A1/TRT(Q) BD/B1,
00160   A2/P/R BD/C2,
00170   B1/A/@P,
00180   C1/=F/C U/B1 R/C2,
00190   C2//
00200   F/ERROR;
00210   END

.RUN COMPIL[72,213]

*SOURCE=EXAMPL
40      4       RET
70      6
130     10      RET     EXIT
140     11
210     20      EXIT.
UNDECLARED IDENTIFIERS:
R
P
TRT
Q
ERROR

123 KCS
CT 71 SEC
EXIT

.
```

This memo concludes with the following collection of all error diagnostics of the Compiler arranged in numeric order. With each message is given further explanatory material and the transformation performed on the source text to recover from the particular error. The meta-variable n used in the diagnostic messages represents a line number which is a sequence number of the source file. When an error condition is reported as being "IN" a particular line, the user should expect to find the error right there. Several diagnostics, however, indicate an error occurs "NEAR" a particular line. In this case the user should look on that line and then look backward from there (towards the beginning of the source file) to locate the error being diagnosed. Usually, the error is, in fact, on the very line mentioned.

The Compiler is usually very thorough in its detection of syntactic and semantics errors. The one exception to this is in the way it (currently) ignores link routes except for at most the first two characters. Syntactically, the compiler requires that a route begin optionally with a perturbation followed by a direction, but then the remainder of the route is ignored as long as it consists of letters and/or digits. Compilation is based only on the destination node specified. Users must be rather careful in this area since the Diagram Generator draws links based only on the route, and it ignores the destination. Thus, total reliance on a diagrammatic listing is not sufficient.

When a diagnostic message appears during a compilation, the user should not assume that re-compilation is necessary. Instead, the recovery procedure employed may be adequate, if not exactly appropriate. Since there is a recovery for every error condition the user should usually allow a compilation to continue to completion.

Several of the recoveries are represented by a transformation on the source to the compiler. A right-arrow is used to separate the before and after forms of the transformation. A delta (Δ) is used to show where the input scanner is located before and after the transformation is applied. As a shorthand notation to show its context, a transformation may be enclosed within curly braces, and the contextual form appears outside of the braces.

1001: ILLEGAL CHARACTER IN n

error	We find a character x which the operating system permits in an input file but which is not permitted in an AMBIT/L program.
recovery	{ Δ x → Δ }
note	The character may be a non-printing character, such as a control character.

1301: PROLOG IS 's1 s2' NEAR n

error	We are beginning compilation of an insertion and expect to find a prolog (namely INSERTION followed by an identifier). Instead, we find the segments s1 s2.
recovery	{ Δ → Δ INSERTION X } s1 s2

1302: PROLOG IS FOLLOWED BY 's1' NEAR n

error	We have read a prolog and expect to find a semicolon. Instead, we find the segment s1 .
recovery	{ Δ → Δ ; } s1

1303: PROGRAM SECTION BEGINS WITH 's1 s2' NEAR n

error	We have read a prolog followed by a semicolon. We expect to find the beginning of a block. Instead we find the segments s1 s2 .
recovery	{ Δ → Δ BEGIN } s1 s2

1401: ARGUMENT LIST ENDS WITH '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We have read a (possibly empty) list of arguments in a function heading and expect to find another argument (an identifier) or a right parenthesis. Instead, we find the segment <u>sl</u> .

<u>recovery</u> { Δ → Δ) } <u>sl</u>

1402: RESULT LIST ENDS WITH '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We have read a (possibly empty) list of results in a function heading and expect to find another result (an identifier) or a colon. Instead, we find the segment <u>sl</u> .

<u>recovery</u> { Δ → Δ : } <u>sl</u>

1403: FUNCTION BODY BEGINS WITH '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We have read a function heading and expect to find the beginning of a block, rule, or insert. Instead, we find the segment <u>sl</u> .

<u>recovery</u> { Δ → Δ BEGIN END } <u>sl</u>

162

1404: PROGRAM SECTION BEGINS WITH '<u>s1</u> <u>s2</u>' NEAR <u>n</u>

<u>error</u> We have read a program section (see note) and expect to find the beginning of a following program section. Instead, we find the segments <u>s1</u> <u>s2</u> .

<u>recovery</u> { Δ s1 s2 ... s[i-1] → Δ s[i-1] } s[i]
where

s1, s2, ..., s[i-1], and s[i]

are segments and <u>i</u> is the smallest integer such that

s[i-1]	is an identifier which can begin a program section, and
s[i]	is any segment except colon or a left parenthesis.

<u>note</u> A program may be viewed as a prolog followed by a sequence of sections, where

```
section  →   BEGIN |
             declarative ; |
             function-heading : |
             identifier : |
             rule ; |
             insert ; |
             END ; |
             END  end-of-file
```

1405: DECLARATIVE FOLLOWED BY '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We have read a declarator followed by a sequence of identifiers, and we expect to find another identifier or a semicolon. Instead, we find the segment <u>sl</u> .

<u>recovery</u> $\{ \Delta \rightarrow \Delta \ ; \}$ <u>sl</u>

1406: MISPLACED DECLARATIVE BEGINS WITH '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We have read one or more function definitions, attached labels, or imperatives in the current block, and we do not expect to find a declarative. Instead, we find a declarative beginning with <u>sl</u> .

<u>recovery</u> (Same as 1404; that is, we skip over the next section of the program.)

1407: FUNCTION DEF FOLLOWED BY '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We have read a function definition and expect to find a semicolon. Instead, we find the segment <u>sl</u> .

<u>recovery</u> $\{ \Delta \rightarrow \Delta \ ; \}$ <u>sl</u>

1408: MISPLACED FUNCTION DEF BEGINS WITH '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We have read one or more attached labels or imperatives in the current block, and we do not expect to find a function definition. Instead, we find a function definition beginning with <u>sl</u> .

<u>recovery</u> (Same as 1404; that is, we skip over the next section of the program.)

1409: BLOCK FOLLOWED BY '<u>s1</u>' NEAR <u>n</u>

<u>error</u> We have just read a block which may have a transfer-list. We expect to find either a transfer-list or a semicolon. Instead, we find <u>s1</u>.

<u>recovery</u> { ∆ → ∆ ; } <u>s1</u>

1410: SECTION BEGINS WITH END-OF-FILE

<u>error</u> We tried to find the beginning of a section (see 1404); instead, we find the end-of-file which terminates the insertion being compiled.

<u>recovery</u> { ∆ → ∆ END }

1601: REDECLARATION OF '<u>s1</u>' NEAR <u>n</u>

<u>error</u> We have read one declaration of the identifier <u>s1</u> in the current block and do not expect to find another declaration of this identifier. Instead, we find another declaration of <u>s1</u>.

<u>recovery</u> { ∆ <u>s1</u> → ∆ }

<u>note</u> Thus, we skip the new declaration of <u>s1</u>.

1602: MISDECLARATION OF '<u>s1</u>' NEAR <u>n</u>

<u>error</u> We have read a reference instance of implicit type L or F of the identifier <u>s1</u> in the current block. We expect to find a declaration instance of implicit type L (in an attached label) or F (in a function definition). Instead, we find some other declaration of <u>s1</u>.

<u>recovery</u> The declaration of the identifier <u>s1</u> we have just read is ignored, and the previous declaration holds.

<u>note</u> This may lead to later 1801 diagnostics.

2501: TOKEN NAME ENDS WITH '<u>s1</u>' NEAR <u>n</u>

<u>error</u> We have begun to read a name which is a token. Such a name should have the form

$(\{ \underline{\text{literal}} \}_0^\infty)$

So far, we have read

$(\{ \underline{\text{literal}} \}_0^\infty \; (\{ \underline{\text{literal}} \}_0^\infty \ldots (\{ \underline{\text{literal}} \}_0^\infty$

and we expect to find a <u>literal</u> or a right parenthesis. Instead, we find <u>s1</u> .

<u>recovery</u> $\{ \underline{x} \; \Delta \rightarrow \Delta \; (0) \} \; \underline{s1}$

where

<u>x</u> is the portion of the name already read.

Thus, this error in a name which is a token causes the entire name to be replaced by the token (0).

2701: NODE BEGINS WITH '<u>s1</u>' NEAR <u>n</u>

<u>error</u> We have just read either

RULE $\{\text{node}, \}_0^\infty$

or

$\{\text{node}, \}_0^\infty$

as the beginning of a rule and we expect to find a position which will begin a new node. Instead, we find the segment <u>s1</u>.

<u>recovery</u> $\{ \Delta \; s1 \ldots s[i-2] \rightarrow \Delta \; s[i-2] \} \; s[i-1] \; s[i]$

where the s[i] are segments and <u>i</u> is the smallest integer such that

a. s [i-1] is a semicolon, or
b. s [i-1] and s [i] are both slashes, or
c. s [i-2] is a comma, or
d. s [i-1] is an end-of-file.

186

2702: RE-USE OF POSITION '<u>s1</u>' IN <u>n</u>

<u>error</u> We have read one or more nodes of a rule and found <u>s1</u> as the position of one of these nodes. Now we find <u>s1</u> as the position of another node in the same rule.

<u>recovery</u> { Δ <u>s1</u> → Δ <u>g1</u> }
where <u>g1</u> is an arbitrarily selected position which has not been used and (we correctly predict) will not be used in this rule.

2703: ILLEGAL TYPE-SET '<u>s1</u>' IN DATA-NODE NEAR <u>n</u>

<u>error</u> We are reading a data-node and have seen a position and a slash, and we expect to find a type-set. Instead, we find <u>s1</u>.

<u>recovery</u> { Δ s[1] ... s[i] → Δ C / ** } s[i+1] s[i+2]
where the s[i] are segments and <u>i</u> is the smallest integer such that

a. s[i+1] is a comma, or
b. s[i+1] is a semicolon, or
c. s[i+1] and s[i+2] are both slashes, or
d. s[i+1] is an end-of-file.

2704: ILLEGAL TEST-NAME '<u>s1</u>' IN DATA-NODE NEAR <u>n</u>

<u>error</u> We are reading a name-test in a data-node. We have seen zero or more occurrences of a name followed by a slash and expect to find a name. Instead, we find <u>s1</u>.

<u>recovery</u> (same as 2703; that is, the current node is replaced by a null cell.)

2705: ILLEGAL MAIN NAME '<u>s1</u>' IN DATA-NODE NEAR <u>n</u>

<u>error</u> We are reading a data-node and have seen a position, a slash, a type-set, and a slash. We expect to find a name or the beginning ('=' or '#') of a name-test. Instead, we find <u>s1</u>.

<u>recovery</u> (Same as 2703; that is, the current node is replaced by a null cell.)

2706: NODE FOLLOWED BY '<u>s1</u>' NEAR <u>n</u>

<u>error</u> We are reading the links of a node and we expect to find another link or the termination of the node by comma, semicolon, or double-slash. Instead, we find <u>s1</u>.

<u>recovery</u> $\{\Delta\ s[1] \ldots s[i] \rightarrow \Delta\}\ s[i+1]\ \ s[i+2]$ where the $s[i]$ are segments and <u>i</u> is the smallest integer such that

a. $s[i+1]$ is a comma, or
b. $s[i+1]$ is a semicolon, or
c. $s[i+1]$ and $s[i+2]$ are both slashes, or
d. $s[i+1]$ is an end-of-file.

2707: ILLEGAL TYPE-SET '<u>s1</u>' IN CALL-NODE NEAR <u>n</u>

<u>error</u> We are reading a call-node and have seen a boundary, a slash, and an '='. We expect to find a type-set. Instead, we find <u>s1</u>.

<u>recovery</u> (Same as 2703; that is, the current node is replaced by a null cell.)

2708: ILLEGAL TYPE-SET 's1' FOR CALL-NODE NEAR n

error — We are reading a call-node and have seen a boundary, a slash, a type-set s1, and either nothing more or a slash followed by a name. The type-set s1 should have been F.

recovery — { s1 → F }

2709: ILLEGAL NAME OR VALUE-CALL 's1' IN CALL-NODE NEAR n

error — We are reading a call-node and have seen a boundary, a slash, an '=', a type-set, and a slash. We tried to read a name and didn't find one and then read a '#' if there was one. We now expect to find a value-call. Instead, we find s1.

recovery — (Same as 2703; that is, the current node is replaced by a null cell.)

2801: RE-USE OF POSITION 's1' IN n

error — We have read one or more nodes of a rule and found s1 as the position of one of these nodes. Now we find a node which has an extended boundary (covering two or more positions) which contains s1 and which is in the same rule.

recovery — The position s1 is deleted from the set of positions specified by the extended boundary.

2802: ILLEGAL EXTENDED BOUNDARY 's1-s2' IN n

error — We find an extended boundary, s1 - s2 at the beginning of a node such that s1 is to the right of and/or below s2.

recovery — { Δ s1 - s2 → Δ s1 }

2901: TYPE-SET '<u>s1</u>' CONTAINS '<u>s2</u>' IN <u>n</u>

<u>error</u> We are reading a type-set <u>s1</u> and expect to find a type-code or a termination of the type-set. Instead, we find <u>s2</u>, which is not a legal type code.

<u>recovery</u> $\{ \Delta \ \underline{s2} \rightarrow \Delta \}$

2902: TYPE-SET '<u>s1</u>' CONTAINS DUPLICATE '<u>s2</u>' IN <u>n</u>

<u>error</u> We are reading a type-set <u>s1</u> and have seen the type-code <u>s2</u>. Now we find a second instance of <u>s2</u>.

<u>recovery</u> $\{ \Delta \ \underline{s2} \rightarrow \Delta \}$

3001: ILLEGAL WALK '<u>s1</u>' IN VALUE-CALL IN <u>n</u>

<u>error</u> We are reading a value-call in a call-node and have seen the 'V' with which it begins. We expect to find a walk. Instead, we find <u>s1</u>.

<u>recovery</u> $\{ \Delta \ \underline{s1} \rightarrow \Delta \}$ @ <u>identifier</u>

3301: ILLEGAL LINK ROUTE '<u>s1</u>' IN <u>n</u>

<u>error</u> We are reading a data-node and expect to find a link or a termination of the data-node. Instead, we find an identifier <u>s1</u>. This should be (at this point) a route, but it is not.

<u>recovery</u> $\{ \Delta \ \underline{s1} \rightarrow \Delta \}$
and assume the termination of the node is next.

<u>note</u> This diagnostic will be followed by diagnostic 2706 (indicating improper termination of a node) when the assumption made in the recovery, above, is incorrect. A termination for a node is a comma, a semicolon, or a double slash.

3302: LINK ROUTE 'sl' NOT FOLLOWED BY DESTINATION IN n

error — We are reading a data-node and expect to find a link or a termination of the data-node. Instead, we find a route sl which is legal but is not followed by a slash and a destination.

recovery — $\{ \Delta \ \underline{sl} \rightarrow \Delta \}$
and assume the termination of the node is next. (See note to 3301.)

3303: LINK WITH ROUTE 'sl' IS DUPLICATE IN n

error — We are reading links in a data-node and have already seen a link of some particular type (solid or broken) and name (horizontal or vertical). Now we find a link whose route, sl, has the same type and name as that already seen.

recovery — $\{ \Delta \ \underline{sl} \ / \ \underline{dest} \rightarrow \Delta \}$

3401: LINK ORIGIN 'sl' OUT OF NODE IN n

error — We are reading the links of a call-node. We expect to find a link which either has no origin or an origin within the (possibly extended) call boundary. Instead, we find the origin sl.

recovery — Accept the origin sl as the origin of the link.

3402: ILLEGAL LINK ROUTE 'sl' NEAR n

error — We are reading a call-node and have read an origin and a slash. We expect to find the route of a link. Instead, we find sl.

recovery — $\{ \underline{origin} \ / \ \Delta \ \underline{sl} \rightarrow \Delta \}$
and assume the termination of the node is next. (See note to 3301.)

3403: BROKEN LINK WITH ROUTE 's1' ON CALL-NODE IN n

error — We are reading the links of a call-node. We expect to find only solid or flow links. Instead, we find a link with route s1 indicating a broken link.

recovery — Replace the 'B' in s1 with an 'S', thus converting it to a solid link.

3404: ILLEGAL LINK ROUTE 's1' IN n

error — We are reading a call-node and expect to find a link or a termination of the call-node. Instead, we find an identifier s1. This should be (at this point) a route, but it is not.

recovery — $\{\Delta\ s1 \rightarrow \Delta\}$
and assume the termination of the call-node is next. (See note to 3301.)

3405: (Same as 3404)

note — 3405 and 3404 are for routes which have and do not have perturbations, respectively.

3406: LINK ROUTE 's1' NOT FOLLOWED BY DESTINATION IN n

error — We are reading a call-node and expect to find a link or termination of the call-node. Instead, we find a route s1 of a solid or broken link which is legal but not followed by a slash and a destination.

recovery — $\{\Delta\ s1 \rightarrow \Delta\}$
and assume the termination of the node is next. (See note to 3301.)

3407: LINK ROUTE '<u>sl</u>' IS DUPLICATE IN <u>n</u>

<u>error</u> We are reading links in a call-node and have already seen a solid link of some particular name (see note). Now we find a solid link whose route is <u>sl</u> and whose name is that already seen.

<u>recovery</u> The link in question is ordered immediately before the previous link which had the same name.

<u>note</u> The argument (result) links of a call-node have names specified by, primarily, the digit (letter) of their names and, secondarily, the perturbation (implicit or explicit) of their routes. In diagrammatic form, an argument (result) link is <u>before</u> another argument (result) link if its point of origin is to the left of (above) the other argument (result) link.

3408: LINK ROUTE '<u>sl</u>' NOT FOLLOWED BY DESTINATION IN <u>n</u>

<u>error</u> We are reading a call-node and expect to find a link or termination of the call-node. Instead, we find a route <u>sl</u> of a flow link which is legal but not followed by a slash and a destination.

<u>recovery</u> $\{ \Delta\ \underline{sl} \rightarrow \Delta \}$
and assume the termination of the node is next. (See note to 3301.)

3901: VALUE-CALL NODE HAS RESULTS IN n

error	A call node with a negated value-call is on line n. No results can be used on such a node, but results appear on this one.
recovery	Delete the result links from the node.

3902: (Same as 3901)

note	3901 and 3902 for negated and non-negated value-calls, respectively.

4301: LINK(S) CONTRADICT TYPE-SET OF NODE IN n

error	The right link(s) of the node on line n imply a type which is not permitted by the explicit type-set of that node.
recovery	The link(s) are eliminated.

4302: (Same as 4301)

note	4301 and 4302 are for right link(s) and down link(s), respectively.

4401: RIGHT LINK FROM NODE IN n1 TO NODE IN n2

error	There is a right link from the node on line n1 to the node on line n2. This implies that the node on line n2 is a cell; but its type-set contradicts that.
recovery	The link is eliminated.

4501: TRANSFER-LIST BEGINS WITH '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We have just read a double slash and expect to find a transfer-list. Instead, we find <u>sl</u>.

<u>recovery</u> { Δ → Δ S / NEXT F / ? ; } <u>sl</u>

4502: ILLEGAL LABEL REF '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We are reading a transfer-list and have just seen the SF/ or S/ or F/ with which it begins. We expect to find a label reference but instead find <u>sl</u>.

<u>recovery</u> { <u>x</u> Δ → Δ S / NEXT F/? ; } <u>sl</u>
where <u>x</u> is SF/ or S/ or F/.

4503: ILLEGAL LABEL REF '<u>sl</u>' NEAR <u>n</u>

<u>error</u> We are reading a transfer-list and have seen the first transfer followed by S/ or F/ as the beginning of the second transfer. We expect to find a label reference but instead find <u>sl</u>.

<u>recovery</u> { Δ → Δ <u>x</u> ; } <u>sl</u>
where

<u>x</u> is F/? if the first transfer was a success transfer, and

<u>x</u> is S/NEXT if the first transfer was a fail transfer.

4504: TRANSFER-LIST FOLLOWED BY 'sl' NEAR n

error — We have just read a transfer-list and expect to find a semicolon. Instead, we find sl.

recovery — $\{ \Delta \rightarrow \Delta ; \}$ sl

4701: FLOW LINK BIND FROM NODE IN nl TO NODE IN n2

error — The current rule has a flow cycle; we (for the moment) blame the cycle on the flow link from the node on line nl to the node on line n2.

recovery — Delete the flow link in question.

note — This is a guess; if it doesn't work, this diagnostic will be followed by diagnostics 4701 or 4702 or 4703.

4702: CANNOT VISIT NODE IN n

error — The current rule has a node on line n which cannot be visited because it is unnamed and is not the destination of a link from a node which has been visited.

recovery — Replace the name of the node on line n with the name @ A.

note — This is a guess; if it doesn't eliminate all other unreachables, further 4702 or 4703 diagnostics will occur.

4703: ARG/RES BIND FROM CALL-NODE IN <u>n1</u> TO NODE IN <u>n2</u>

<u>error</u> The current rule has a cycle where the argument(s) of function call(s) cannot be determined until the function(s) are called. We blame the problem on the argument link from the call-node on line <u>n1</u> to its argument on line <u>n2</u>, since the binding of the argument node depends on the function of which it is an argument to be called.

<u>recovery</u> Change the argument link being blamed to point to the NULL CELL.

5901: ILLEGAL LABEL REF NEAR <u>n</u>

<u>error</u> We find a label-reference, <u>s1</u>, which is a legal name but is not an identifier, negated identifier, or indirect, as required.

<u>recovery</u> $\{ \Delta\ \underline{s1} \rightarrow \Delta\ ?\ \}$

<u>note</u> Diagnostic 5901 applies to the fail transfer of a rule; 6001, to any success transfer; and 6101, to the fail transfer of a block.

6001: (Same as 5901)

6101: (Same as 5901)

6701: DESTINATION '<u>s1</u>' IS MISSING IN <u>n</u>

<u>error</u> We have read a link whose destination node has been referenced on line <u>n</u>, but has not been specified in this rule.

<u>recovery</u> Place a type-less, name-less node at <u>s1</u>.

(END)

Section I

U [illegible] LINK: the AMBIT/L Link Editor

January 11, 1972

This section describes how to use the AMBIT/L Link Editor. First its simplified use by a novice user is described. Next, its normal use is given, and finally the advanced use of partial dumping is described.

The AMBIT/L Link Editor (LINK) is used to prepare an executable AMBIT/L program from one or more AMBIT/L REL files (which are the compiled output of the AMBIT/L Compiler). This process is done in two steps: first, each insertion of the program being prepared is "linked", which means that it is transformed into a final binary representation which is embodied as an "ABS" file (for "absolute"). Linking of a particular insertion must be done in the context of all insertions which include the blocks enclosing it. Linking of a particular insertion amounts to resolving all of its references to undeclared identifiers as detected by the Compiler. The intermediate binary code of the REL file is created with holes or space which can be filled in by LINK to create the ABS file. Since a REL file must contain the information on what is unresolved and also a complete symbol table, its size is usually considerably greater than the corresponding ABS file. In fact, each ABS file merely contains one word of overhead plus the number of words of interpreter object code which was indicated by the Compiler as it translated the corresponding insertion.

The second step of the program preparation process performed by LINK is the collection or dumping of all ABS files of a program into one large "DMP" file. A DMP file is the entire final representation of a complete AMBIT/L program in a form which can be executed interpretively by the AMBIT/L interpreter.

If the program the user is preparing consists of just one insertion in which no PERM pointers are declared, the method of using LINK is greatly simplified. This is an advantage for the novice AMBIT/L user since link editing has proven to be the most "mysterious" part of using the AMBIT/L Programming System. Thus the simplified use is first described, and the novice user need not confuse himself by reading further. However, as soon as a program exceeds one insertion the "normal" method of link editing must be used. It is not intended that a program should be very large (say, over 50 rules) and still consist of only one insertion. Instead, it is expected that when that large a program is being developed the user is no longer a novice, and he should not hesitate to split his program into several insertions.

Note that for LINK to operate in the simplified mode the directory where the REL file is located and the directory in which the user is currently logged in must not contain a file of the same primary name as that of the REL file being processed and with an extension of "LNK".

Novice Use of LINK

The AMBIT/L Link Editor is invoked by a Monitor command of the form:

RUN LINK [proj , prog]

where proj and prog are the project and programmer numbers of the directory where the AMBIT/L Programming System is residing. This calling sequence may be somewhat different, depending upon the method used to install AMBIT/L on a particular PDP - 10.

When LINK is invoked it first prompts the user by typing an asterisk on a new line. The user is then expected to type the primary name of the REL file to be processed; normally this is also the primary name of the source file of his insertion. The name must consist of one to six alphanumeric characters, beginning with an alphabetic character, and it must match the name of the insertion (up to the first six characters excluding periods). The name may be optionally followed by a project-programmer number within square brackets to specify that the REL file is in that disk directory, rather than in the one in which the user is currently logged in. The file specification is immediately followed by a carriage return.

Then LINK goes through the two steps of processing: linking and dumping. Unless there are severe errors, LINK produces three files in the user's current disk directory, each with the same primary name as the REL file being processed:

1. an ABS file with an extension name of "ABS" which is the intermediate representation of the program between the two steps of link editing; and

2. a DMP file with an extension name of "DMP" which is the final representation of the program in the form ready to be executed interpretively by the AMBIT/L interpreter; and

3. a listing file with an extension name of "MAP" which indicates all identifiers defined in the program along with a numeric definition of each one. This map is used primarily in conjunction with the DAMBIT/L debugging system and thus its format is described in the section, "Using DAMBIT/L: the AMBIT/L Debugging System".

LINK normally will produce one typed line on the terminal to inform the user that it has linked the REL file. It consists of the primary name of the REL file (which is the same as the "command" given to LINK) followed by a decimal number which is the page number assigned to the insertion by LINK. At present, the built-in environment to every AMBIT/L user program consists of 26 pages; thus page number 27 is assigned as the page number of the insertion being linked.

During the linking process LINK may detect that an identifier which was undeclared within the program is not defined in the built-in environment. In this case it will inform the user of this by typing an informative error diagnostic on the terminal and then proceed with the linking process. Even with one or more errors of this type a potentially executable program is produced. If during execution, however, the interpreter encounters a reference to an undeclared identifier it will detect an error condition and cause an error trap to occur.

At the conclusion of a successful application of LINK or after a fatal error is detected and reported to the user, control is returned to the PDP - 10 Monitor. At any time during an application of LINK, the user may abort the run by typing two ↑C's (CTRL C) on the terminal; control will then be returned to the PDP - 10 Monitor.

Normal Use of LINK

The previous description was appropriate only for the link editing of programs consisting of one insertion in which no PERM pointers are declared. The following text describes use of LINK in its complete generality. The general method of use can also be used to link single-insertion programs.

Before invoking LINK, the user must prepare two text files in addition to having compiled any insertions to be linked. One file must be prepared once as a description of the structure of the program being prepared. It is called a "BLK" file for "<u>bl</u>oc<u>k</u> structure". This file must be created with one line for each insertion in the program; each line contains any number of SPACEs and/or TABs followed by the primary name of a REL file corresponding to one of the insertions of the program. This name consists of one to six alphanumeric characters, beginning with an alphabetic character. If there are any PERM pointers declared in the corresponding insertion the name must be followed by the unsigned decimal number which indicates the total number of PERM pointers explicitly declared within the text of the insertion. If such a number is given it is separated from the name by one or more SPACEs and/or TABs.

It is recommended that the user prepare a BLK file using relative indentations of the file names to show the block structure of his program. Although such arrangement is optional, it is strongly urged for it aids in preparing LINK command files, and it helps document the program as a whole. If this arrangement is used the ordering of names within the BLK file is constrained. If the user chooses not to adhere to the recommended arrangement, he must at least include the primary file name of the outermost or main block of his program on the very first line of the BLK file.

A BLK file may optionally have sequence numbers as provided by some text editing programs available on the PDP - 10. It is customary and helpful for the file to be sequenced by ones and to have each sequence number conform to the page number assigned to the insertion by LINK. This page numbering will be explained later.

Next is presented an example of a BLK file prepared as suggested. This particular BLK file is one for the AMBIT/L Compiler, which is itself an AMBIT/L program. Note that some insertions of the Compiler include PERM declarations.

```
COMPIL.BLK  12-22-71.1520    [72,270]

00027    COMPIL
00028      ERR
00029        ERR1
00030        ERR2
00031        ERR3
00032      GETSEM
00033        CONTRO
00034      TYPECO 1
00035      OPENCO
00036      GETPRO
00037      GETBLO 17
00038        OPENBL
00039        DECLAR
00040        GENID 2
00041        CLOSEB
00042        DESCR
00043        GETINS 2
00044        GETDEC
00045        ISFLK
00046        GETNAM
00047        GETTRL
00048        GETRUL
00049          WRMPOS
00050          GETTYP
00051          GETVIN
00052          GETDLS
00053          GETCLS
00054          FLOWRE
00055          EXPAND
00056          GATHER
00057          TYPEAN 2
00058        GENRUL 9
00059          GENMAT 1
00060          GENLIN 1
00061          GENREF 6
00062            GENNAM 2
00063            GENTYP 5
00064            GENMLI 7
00065          GENERR
00066        GLNTR 8
00067        GENUNC 3
00068        GENFAI 3
00069        GENWAL 5
00070      CLCPL
```

2:14

The other text file which the user must prepare may vary from one application of LINK to another. It is called a "command file" since it contains the sequence of commands to control LINK. Each command is a two- or three-character mnemonic and some commands may take one argument. Each command must be on a separate line of the command file. SPACEs and TABs may be used optionally before each command. One or more SPACEs and TABs must be used to separate a command from its argument (if any).

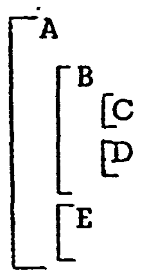
During its operation LINK must look-up several files: a BLK file, REL files and/or ABS files, and perhaps a DMP file. When performing any such look-up LINK first tries to find the file it is seeking in the disk directory in which the user is currently logged in. If the file is not there, then LINK tries to look in the current "library directory" if one was specified. The command file may contain any number of "LIB" commands to specify the current library directory. A LIB command may optionally take one argument which is a directory specification in the following standard format:

[proj , prog]

where proj and prog are the project and programmer numbers of the directory being specified. If no argument is given to a LIB command no library directory is used during succeeding file look-ups. At the beginning of an application of LINK the library directory is initially the one where the command file is.

Except for a possible LIB command, the command file must begin with a "BLK" command to specify the name of the BLK file to be used for this application of LINK. The argument to the BLK command must be a file name in standard form: a primary name of one to six alphanumeric characters followed optionally by an extension of zero to three alphanumeric characters with a period as separator. If no period and no extension is provided LINK assumes a default extension of "BLK". A null extension name is specified by following the primary name with just a period.

Next, the command file may contain commands to control the linking of any number of insertions of the program being prepared. Except for LIB commands, these commands are "LNK" (for "link"), "IN", and "OUT". If this application of LINK is used only for dumping then none of these commands need be included. The LNK and IN commands each require one argument as the primary name corresponding to the insertion to which they refer; thus it consists one to six alphanumeric characters. The OUT command has no arguments. These three commands are used to "wind around" the block structure of the program being prepared as necessary so that those insertions to be linked are appropriately linked in their proper context of enclosing blocks. Each LNK or IN command moves the current point of context deeper one level in the block structure, and each OUT command pops out one level. As an example, for the following program structure:



the following sequence of commands must be used to link all five insertions:

```
LNK   A
LNK   B
LNK   C
OUT
LNK   D
OUT
OUT
LNK   E
OUT
OUT
```

If only insertion E were to be linked the sequence of commands would be:

```
IN    A
LNK   E
OUT
OUT
```

Since insertions B, C and D do not enclose insertion E they do not have to enter into the sequence of commands.

The two examples just given include corresponding OUT commands to every LNK or IN command. Although this is conceptually correct, a lazy user is free to omit any number of final OUT commands in his sequence of commands to affect linking.

Following the commands to control linking (if any), the command file may contain a "DMP" command to specify that a DMP file be created. The DMP command may take an optional argument as a primary file name of one to six alphanumeric characters. If such a name is provided, the DMP file is created with that primary name and an extension of "DMP". If no argument is given the DMP file is created with the same extension name ("DMP"), but with the same primary name as that used for the REL file of the insertion which is the outermost or main block of the program being prepared.

Finally, a command file may terminate optionally with an "END" command which has no argument. Since LINK interprets the reading of an end-of-file as an END command the user is free to omit it.

This description of the normal use of LINK has thus far described the preparation of the BLK file and command file. The remainder of this section provides a description of its use. The AMBIT/L Link Editor is invoked by a Monitor command of the form:

RUN LINK [<u>proj</u> , <u>prog</u>]

where proj and prog are the project and programmer numbers of the directory where the AMBIT/L Programming System is residing. This calling sequence may be somewhat different, depending upon the method used to install AMBIT/L on a particular PDP - 10.

When LINK is invoked it first prompts the user by typing an asterisk on a new line. The user is then expected to type the name of the command file he has prepared for this application of LINK; the name is accepted in standard form: a primary name of one to six alphanumeric characters followed optionally by an extension of zero to three alphanumeric characters with a period as separator. If no period and no extension is provided LINK assumes a default extension of "LNK". A null extension name is specified by following the primary name with just a period. The file name may be optionally followed by a project-programmer number in square brackets to specify that the command file is in that disk directory, rather than in the one in which the user is currently logged in. The file specification is immediately followed by a carriage return .

LINK then tries to find the specified command file, and if successful it then continues automatically until proper or improper termination as controlled by the commands of the command file. If, however, it cannot find the specified command file, LINK assumes it has been invoked for novice use. This is equivalent to its finding a command file of the form:

```
BLK    pname
LNK    pname
DMP
```

and a BLK file whose name is pname .BLK, whose one-line contents are:

```
pname
```

where pname is the primary name given for the specified command file. If the user had made a typing error, he should expect an error diagnostic being produced resulting from LINK's inability to locate pname.REL .

9.3ll

After it finds the specified command file it reads that file one command at a time. A description of the LIB command and the meaning of the library directory was provided earlier. LINK expects to find a BLK command specifying a BLK file; if there is no such command or if it cannot locate the BLK file, an informative error diagnostic is typed and control returns to the PDP - 10 Monitor. Otherwise the BLK file is read and assimilated.

Then LINK goes through the two steps of processing accordingly a they are specified: linking and/or dumping. As printable output for this one application of link, a listing file is produced in the disk directory in which the user is currently logged in with an extension name of "MAP" and a primary name which matches that of the command file. Each insertion which is linked contributes a logical page to the listing which indicates all identifiers defined in that insertion along with a numeric definition of each one. This map is used primarily in conjunction with the DAMBIT/L debugging system and thus its format is described in the section, "Using DAMBIT/L: the AMBIT/L Debugging System".

LINK normally will produce one typed line on the terminal for each insertion which has been linked. It consists of the primary name of the REL file followed by a decimal number which is the page number assigned to the insertion by LINK. At present, the built-in environment to every AMBIT/L user program consists of 26 pages; thus 27 is assigned as the page number of the outermost or main block of a user program, and succeeding numbers are assigned according to the ordering of the file names in the BLK file.

For each REL file (or insertion) being linked, an ABS file is created in the disk directory in which the user is currently logged in with an extension name of "ABS" and a primary name which matches that of its corresponding REL file. Each ABS file is the intermediate representation of an insertion between the two steps of link editing.

During the linking of an insertion, LINK may detect that the definition of an identifier which was undeclared within that insertion (at compile-time) cannot be resolved since no enclosing block, including the built-in environment, defines the identifier. In this case, LINK will inform the user of this by typing an informative error diagnostic on the terminal and then proceed with the linking process. Even with one or more errors of this type potentially executable interpreter code is produced. If during execution, however, the interpreter encounters a reference to an undeclared identifier it will detect an error condition and cause an error trap to occur.

If a DMP command is included in the command file LINK creates a DMP file by collecting together all ABS files of the program as governed by the primary names included in the BLK file. All ABS files must exist for the dumping to be successful in either the current disk directory or the current library directory (as set by the most recent LIB command). A missing ABS file will cause an error condition to be detected which results in an informative diagnostic message being typed on the terminal and control returned to the PDP - 10 Monitor, as with any fatal error. At any time during an application of LINK, the user may abort the run by typing two ↑C's (CTRL C) on the terminal; control will then be returned immediately to the PDP - 10 Monitor.

Once a program's DMP file has been created, the only reason for not deleting the constituent ABS files is that a partial linking may later be done to fix just one part of the program. The user must beware, however, that such partial linking must be done carefully since errors are difficult to detect. When re-linking a particular insertion, the user should be sure that all enclosed insertions should also be re-linked since the definition of an identifier may have changed in the enclosing block. Since compilation is relatively expensive and linking is not, for long-term storage it is recommended that all ABS files be deleted. Also for long-term documentation one complete map listing of all insertions of a program is more desirable than individual ones.

Partial Dumping

For large programs it may be desirable to use partial results of a previously created DMP file so that only changes or additions need be processed. This is an advanced use of LINK which should be avoided until the user feels rather comfortable with link editing. The normal user may cease reading this memo here.

As previously described. the command file may end in a DMP command which directs LINK to collect all ABS files mentioned in the BLK file and create a DMP file. The optional argument used for specifying a primary name of the DMP file is given on the same line as the DMP command.

For specifying partial dumping, the user may provide an optional second argument (even if the first argument was not given) as a decimal integer at the beginning of the very next line of the command file. This integer is a page number indicating the HIGHEST PAGE NUMBER WHICH DOES NOT HAVE TO BE RE-COLLECTED.

If the integer argument given is less than 27, or if it is not given, or if there is no DMP file in the current disk directory whose name is the same as the DMP file being specified, then a new DMP file is created. Otherwise, the old DMP file will be used and appropriately altered; note that its creation date remains the same. Also, the length of the new DMP file will be at least as large as that of the old one, even if there is less useful information in the new one.

There is, however, one restriction on increasing the number of pages of a DMP file by partial dumping: if there are $\underline{x}$ pages in the old DMP file (this does not include the 26 built-in pages) and there are $\underline{y}$ pages in the new one, then

if $1 \leq x \leq 123$, then y must be ≤ 123,
if $124 \leq x \leq 251$, then y must be ≤ 251,
if $252 \leq x \leq 374$, then y must be ≤ 374.

At the present time DMP files are restricted to being 374 user pages long.

211

As an example, for the following BLK and command files, LINK will use the existing A.DMP as a base from which to create a new one. In particular, up to page 30 of the previous DMP file is assumed to be unchanged. Note that if, for example, the old DMP file had only up to page 28, then LINK would "know enough" to begin collecting at page 29, even though the command file has "30". Only those ABS files which are being collected need to exist for such a partial dumping.

A.BLK	A.LNK
A	BLK A
B	IN A
C	LNK E
D	DMP
E	30

212

(END)

Section J

Using the AMBIT/L Cross-Referencer

January 13, 1972

This section describes how to obtain and interpret cross-reference maps of off-page references to identifiers according to the five categories: LABELs, MARKs, FUNCTIONs, PERM POINTERs, and TEMP POINTERs. To take advantage of this facility the user must be familiar with using the AMBIT/L Link Editor.

The cross-referencing process is performed by the user in two stages. First, the user must invoke the AMBMAP program by a Monitor command of the form:

RUN AMBMAP [proj , prog]

where proj and prog are the project and programmer numbers of the directory where the AMBIT/L Programming System is residing. This calling sequence may be somewhat different, depending upon the method used to install AMBIT/L on a particular PDP - 10.

When AMBMAP is first invoked it prompts the user with an asterisk on a new line. The user is then expected to type the name of a command file he has prepared for this application of the Cross-Referencer; the name is accepted in standard form: a primary name of one to six alphanumeric characters followed optionally by an extension of zero to three alphanumeric characters with a period as separator. If no period and no extension is provided AMBMAP assumes a default extension of "LNK". A null extension name is specified by following the primary name with just a period. The file name may be optionally followed by a project-programmer number in square brackets to specify that the command file is in that disk directory, rather than in the one in which the user is currently logged in. The file specification is immediately followed by a carriage return.

AMBMAP then tries to find the specified command file, and if successful it then continues automatically with the first stage of the cross-referencing process (until proper or improper termination) as controlled by the commands of the command file. If, however, it cannot find the specified command file, it notifies the user with an informative error diagnostic typed on the terminal and control is returned to the PDP - 10 Monitor.

The form of the command file which AMBMAP accepts is exactly the same as the command file for a link edit. AMBMAP interprets a DMP command as if it were "END". The user should refer to the part, "Normal Use of LINK" in Section I, "Using LINK: the AMBIT/L Link Editor". Note that

AMBMAP does not accommodate the same simplified or novice use of LINK. Thus a BLK file must exist for the program being cross-referenced.

Although AMBMAP accepts a command file of the same form as that of LINK it interprets the LNK command as the signal to include the particular insertion in the cross-reference maps it produces rather than link edit the corresponding REL file. As for LINK, AMBMAP requires the existence of any REL files which are mentioned in either IN or LNK commands.

AMBMAP normally will produce one typed line on the terminal for each insertion being cross-referenced. It consists of the primary name of the REL file followed by a decimal number which is the page number of that insertion. Recall that 27 is the page number of the outermost or main block of a user program.

During the cross-referencing of an insertion, AMBMAP may detect that the definition of an identifier which was undeclared within that insertion cannot be resolved since no enclosing block, including the built-in environment, defines the identifier. In this case an informative diagnostic message will be typed on the terminal and cross-referencing will proceed normally. Undeclared identifiers are not included in the cross-reference listing.

The result of applying AMBMAP is the creation of five text files in the current disk directory containing the cross-referencing information in an encoded form. Each insertion which is cross-referenced may contribute information to any of these files. The primary name of each of these files is the same as that of the specified command file. Different extension names are used to indicate their contents:

Extension	Contents - Off Page References to:
LBL	LABELs
MRK	MARKs
FNC	FUNCTIONs
PRM	PERM POINTERs
TEM	TEMP POINTERs

If AMBMAP terminates properly all five files will be created even if some do not have any cross-referencing information. If improper termination occurs, entries will be made in the disk directory, but the files will have no contents; in such a case an informative diagnostic message will be typed on the terminal and control returned to the PDP - 10 Monitor. At any time during the execution of AMBMAP, the user may abort the run by typing two ↑C's (CTRL C) on the terminal; control will then be returned immediately to the PDP - 10 Monitor. This concludes the first stage of cross-referencing.

The second stage of cross-referencing consists of the user's directing the ALXREF program to create a cross-reference listing of one of the five types of references in alphabetical order by converting one of the files created during the first stage. ALXREF must be separately invoked for each of the five possible listings which is desired. For each identifier which was referenced off-page, the listing indicates on which page (by insertion name) the identifier is declared and then a list is given which indicates in which insertions such an off-page reference occurs. Multiple declarations of the same identifier on a page are merged (unfortunately). If an identifier is declared in the built-in environment the insertion name given for where it is declared is one of the following: ZENV0, ZENV1, ZENV2, ZENV3. There is no reason for the normal user to distinguish among these four insertion names.

The ALXREF program is invoked by a Monitor command of the form:

```
RUN ALXREF [ proj , prog ]
```

where proj and prog are the project and programmer numbers of the directory where the AMBIT/L Programming System is residing. This calling sequence may be somewhat different, depending upon the method used to install AMBIT/L on a particular PDP - 10.

When ALXREF is invoked it prompts the user to supply an input file name by typing

```
INPUT=
```

on a new line. The user is then expected to provide the name of one of the five files produced by the first stage of the cross-referencing process. The file name is accepted in standard form, but it may not be followed by a disk directory specification. Thus the file must exist in the disk directory in which the user is currently logged in. If the file is not found or if a syntactically incorrect file name is specified, the user is given another try.

If the file is found, ALXREF next prompts the user to supply the name of the BLK file of the program being cross-referenced by typing:

```
BLK FILE: INPUT=
```

The user is then expected to provide that file name in standard form. If no period and no extension is provided ALXREF assumes a default extension of "BLK". A null extension name is specified by following the primary name with just a period. The name of the BLK file may not be followed by a disk directory specification, and thus it must exist in the disk directory in which the user is currently logged in. If the file is not found or if a syntactically incorrect file name is specified, "INPUT=" is repeated for the BLK file, and the user is given another try.

If the BLK file is found, ALXREF next prompts the user to supply the name of an output file for the cross-reference listing by prompting the user with:

```
OUTPUT=
```

The user is then expected to provide a file name in standard form, but it may not be followed by a disk directory specification. If no period and no extension is provided ALXREF assumes a default extension of "LST". A null extension name is specified by following the primary name with just a period. If a syntactically incorrect file name is specified, the user is given another try. The specified output file is then created in the disk directory in which the user is currently logged in. After successful completion ALXREF returns control to the PDP - 10 Monitor. The listing thus produced may be typed or listed by standard means. The user should eventually delete the intermediate five files used to hold the intermediate cross-referencing information.

This memo concludes with a sample cross-reference listing of FUNCTIONs in the ALXREF program itself.

```
AMBIT/L EXTERNAL SYMBOL CROSS-REFERENCE MAP
INPUT FILE: ALXREF.FNC
BLK FILE: ALXREF.BLK

ADI  (ZENV0)
  USE:  XRPRO   FILEN   ALXREF

AFTER  (ZENV0)
  USE:  ALXREF

ALPHNUM  (ZENV0)
  USE:  XRINP   FILEN

CLOSE  (ZENV2)
  USE:  XRPRO   ALXREF

COPY.LIST  (ZENV0)
  USE:  XRINP   FILEN

DELCR  (ALXREF)
  USE:  XRINP

INL  (ZENV2)
  USE:  XRPRO   ALXREF

INLS  (ZENV2)
  USE:  XRINP

LE  (ZENV0)
  USE:  FILEN

MEMBER  (ZENV0)
  USE:  XRPRO   XRINP   FILEN   ALXREF

OPEN  (ZENV2)
  USE:  ALXREF

OUTL  (ZENV2)
  USE:  ALXREF

OUTS  (ZENV2)
  USE:  XRPRO   ALXREF

REQ.XRINPUT  (ALXREF)
  USE:  XRPRO

TRD  (ZENV0)
  USE:  ALXREF

TRI  (ZENV0)
  USE:  XRINP

TRS  (ZENV0)
  USE:  XRPRO   XRINP   FILEN   ALXREF

TRT  (ZENV0)
  USE:  XRPRO   ALXREF
```

(END)

Section K

System PERM POINTERs for the

AMBIT/L User

December 13, 1971

This section indicates the names, numbers, and uses of those System PERM POINTERs which are built-in to the AMBIT/L Interpreter and are of interest to the normal user. Other such pointers are used internally by DAMBIT/L, the AMBIT/L debugging system.

PAGER.CT - PERM 13

Each time a page is read from the DMP file into the object area of the interpreter the destination of the DOWN link of this POINTER is updated to point to an INTEGER whose value is one greater than before. If the NULL CELL is found, it is treated like the INTEGER 0. This counter operates modulo 32768.

FREE.CT - PERM 15

After each garbage collection or call on the built-in function FLTH this POINTER is made to point DOWN to the INTEGER which represents the number of words (count) of free storage available.

TRAP.GCOL - PERM 16

When garbage collection is invoked automatically (i.e., not by calling the GCOL built-in function), then after FREE.CT is updated an attempt is made to call a trap function via this POINTER. If it points DOWN to the NULL CELL no function call is made. At present, this POINTER is initialized to the NULL CELL.

GCOL.CHOKE - PERM 17

After a garbage collection is complete and FREE.CT is updated, if there are no words of free storage then an attempt is made to transfer control indirectly via this POINTER. If it points DOWN to the NULL CELL (which is how it is initialized) the GC fatal error trap occurs; otherwise an "indirect goto" is performed under the assumption that the programmer has set the DOWN link of this POINTER to point to a LABEL node corresponding to an appropriate place in his program. Since this is a "goto" rather than a function call it may pop the interpreter control stack in such a way that previously referenced structures are made available for garbage collection.

STACK.CHOKE - PERM 18

If an overflow of the interpreter control stack occurs an attempt is made to transfer control indirectly via this POINTER. If it points DOWN to the NULL CELL (which is how it is initialized) the STK fatal error trap occurs; otherwise n "indirect goto" is performed under the assumption that the programmer has set the DOWN link of this POINTER to point to a LABEL node corresponding to an appropriate place in his program such that the stack will be popped by some amount.

P.RAND - PERM 19

This POINTER is used only by the RANDOM built-in function; for a complete description of its use see the documentation of the function in "AMBIT/L Built-in Functions for the Programmer" (Section E).

P.SEED - PERM 20

This POINTER is used only by the RANDOM built-in function; for a complete description of its use see the documentation of the function in "AMBIT/L Built-in Functions for the Programmer" (Section E).

(END)

Section L

AMBIT/L Program Execution

January 14, 1972

This section describes how to invoke the AMBIT/L interpreter to run a particular AMBIT/L program which has been compiled and link edited into a single DMP file. Also included is a description of how to prepare a bootstrap for running a commonly used AMBIT/L program.

Execution

Once an AMBIT/L program has been linked and dumped by the Link Editor it is embodied as one DMP file (whose extension name is "DMP"). It is then ready to be executed as a running program by the AMBIT/L interpreter. Although a special bootstrap can be made up to cause the interpreter to run the program, the usual method of use during program development begins by invoking the interpreter directly. The creation and use of a bootstrap will be discussed later.

The AMBIT/L interpreter is invoked by a Monitor command of the form:

RUN AMBIT [proj , prog]

where proj and prog are the project and programmer numbers of the directory where the AMBIT/L Programming System is residing. This calling sequence may be somewhat different, depending upon the method used to install AMBIT/L on a particular PDP - 10.

When the interpreter is invoked it first prompts the user by typing an asterisk on a new line. The user is then expected to type a command line which must begin with the name of the program to be run, which is the primary name of the DMP file containing the executable program. The name must consist of one to six alphanumeric characters, but note that the user is free to alter the primary name of any AMBIT/L DMP file after it is created by the Link Editor or between two uses of the program. The name may be optionally followed by a project-programmer number within square brackets to specify that the DMP file is in that disk directory, rather than in the one in which the user is currently logged in. Then, the command line may be terminated by a carriage return if the user does not wish to supply any optional parameters affecting interpreter initialization. The interpreter then looks for the specified DMP file and if found uses it in read-only mode and proceeds with the execution of the program. If the DMP file is not found an indicative error message is typed on the terminal, and the interpreter restarts by prompting the user with another asterisk on a new line. If a syntactic error is detected in this or any other part of the command line only a question mark is typed and the interpreter then restarts.

Several options are available to the user to control the allocation of memory space and the employment of the AMBIT/L debugging system or the DDT debugger (useful only to AMBIT/L systems programmers). These options are controlled by "switches" (consistent with PDP - 10 monitor terminology) following the name (and perhaps directory specification) on the command line. The switches may be given in any order and the use of SPACEs as separators is entirely optional. Each switch begins with a slash followed by one or more letters. Although just the first letter is checked by the interpreter, names of switches presented here are longer for mnemonic value. Those options which require an argument accept the argument following an equal sign which follows the switch name.

The novice user is expected to ignore all but one switch, and thus it is first described so the remainder of this section may be skipped until special needs arise. The following switch may be included on the command line following the DMP file name (and perhaps directory specification):

/BRK

which is a mnemonic for "user <u>break</u>". When this switch is included on the command line the DAMBIT/L debugger will be invoked at the very first rule (just before its execution) of the user's program. This is particularly useful for setting console breaks or "breakpoints" in one's program before it begins. Details of using DAMBIT/L are covered in the section, "Using DAMBIT/L: the AMBIT/L Debugging System". (The novice user should stop reading this memo here, and skip foward to pages L-7 and L-8.)

To fully describe the impact of using the various other switches is a difficult task, and thus the following descriptions are rather sketchy. The reader is encouraged, however, to read through all of the descriptions since a variety of useful information is found within them. For those switches which require a numeric argument, the value of that argument is checked to see that it lies within an allowable range; if it does not, the error is reported in the same way that a syntactic error is indicated.

/LOW = kcore

where kcore is a decimal number between 3 and 32; a default value of 6 is normally used. This switch is used to specify in decimal K (i.e., 1024-word blocks) the size of the low segment used for the running of the program. The AMBIT/L interpreter runs on the PDP - 10 as a low and sharable high segment program. The high segment always occupies 6K words but the size of the low segment is adjustable at initialization time to conform to the particular needs of the AMBIT/L program being executed. The low segment includes all changeable memory of the AMBIT/L interpreter. Of particular concern are the three relatively large areas of this memory: the object code paging area, the control stack, and free storage area. The size of the low segment should be chosen large enough for these three areas to be sufficiently large. Using a smaller low segment is more economical if it does not lead to increased computing time; this becomes a space/time trade-off.

/OBJ = n

where n is a decimal number between 400 and 16000; a default value of 2000 is normally used. This switch is used to specify the size in memory locations (words) of the object code paging area used for the object code in the DMP file, for input/output buffers, and for the Garbage Collector bit table (when in use). The bit table's length is approximately 1/32 of the size of free storage. At a minimum, all input/output buffers which are active must remain in this area plus the page currently being executed or the bit table during a garbage collection. The Teletype buffers normally take up 85 words and each open disk channel requires 533 words. To avoid a lot of paging activity, which slows down the running time of a program (and adds some cost), this area should be sufficiently large to hold several pages at once. If it is too small the PAG fatal error trap will occur and the user will be so informed. To gain information on the activities of the paging system the user may employ the page-timing instrumentation version of the AMBIT/L interpreter called "TAMBIT". Details of its use are given in Section N, "Using TAMBIT: the AMBIT/L Interpreter with Page Timing Instrumentation". Short of that, the user may observe (possibly with the DAMBIT/L debugger) the System PERM POINTER PAGER.CT. Each time a page is read from the DMP file (either environmental or user) into the object area, the destination of the DOWN link of this POINTER is updated to point to an INTEGER whose value is one greater than before. If the NULL CELL is found, it is treated like the INTEGER 0. This counter operates modulo 32768.

/STACK = n

where n is a decimal number between 300 and 16000; a default value of 600 is normally used. If the /DDT switch is also included on the command line, a fixed stack size of 1000 is used, independently of this switch. The stack is used as an ALGOL stack for the saving of function call information and for temporary storage. Its size requirement mostly depends upon the depth of function calls or insertions entered and on the amount of recursion the program performs. It must be long enough to accommodate the needs of a particular program run. If it is too small, at the time stack overflow is detected, the interpreter will attempt a recovery procedure of transferring control indirectly via a label pointed to by the System PERM POINTER STACK.CHOKE. It is expected that the programmer has set the DOWN link of this POINTER to point to a LABEL node corresponding to an appropriate place in his program such that the stack will be popped by a sufficient amount. If there is a stack overflow and STACK.CHOKE points to the NULL CELL the STK fatal error trap occurs and the user will be so informed.

/MAXPAG = n

where n is a decimal number between 2 and 60; a default value of 50 is normally used. This parameter is the maximum number of pages which may be in core at the same time. More precisely, the parameter affects the number of entries in the page-use table in the low segment. Each entry occupies three words of memory. One entry is required for each page or input/output buffer or garbage collection bit table residing concurrently in the object code paging area. The numb[illegible] of entries allocated by the interpreter is the minimum of the value of the MA[illegible] parameter and the highest numbered page in the user's program. The MAXPAG parameter should be adjusted to correspond reasonably with the size of the object code area. For example, if this parameter were small and the object code area were large there would probably be thoroughly unused memory being wasted.

/PRINT

this switch takes no argument; its presence causes the interpreter to print on the terminal several values of parameters which describe the user's program and the allocation of memory just before execution of the AMBIT/L program begins. This information includes:

a) total core occupancy in decimal k (low segment and high segment)

b) length of the stack in words

c) size of the object code paging area in words

d) maximum number of in-core pages (i.e., the number of entries in the page-use table)

e) size of free storage in words

f) highest page number of the user's program

g) total number of PERMs of the user's program and the built-in environment

h) maximum static levels or depth of block structure

Note that there is no switch which directly controls the size of free storage. Instead, after all other allocation of the low segment is complete, the free storage area is allocated as all remaining space in the low segment. If it is smaller than the required minimum of 200 words then a fatal error condition is detected, an informative diagnostic message is typed on the terminal, and control is returned to the PDP - 10 Monitor.

NOTE: THE FOLLOWING TWO SWITCHES ARE INTENDED ONLY FOR USE BY AMBIT/L SYSTEMS PROGRAMMERS.

/ENV = name [proj , prog]

where name is a primary file name consisting of one to six alphanumeric characters, and proj and prog are the project and programmer numbers of a disk directory. This switch is used to specify an alternate environmental DMP file for the run if one is needed. The directory specification need only be given if the DMP file is not in the disk directory in which the default environmental DMP file is located. The default environmental DMP file is usually ZENV.DMP in the disk directory where the AMBIT/L Programming System is residing. This may vary somewhat, according to the method used to install AMBIT/L on a particular PDP - 10.

/DDT

this switch takes no arguments; its presence causes the interpreter to allocate memory so that the copy of DDT (the standard debugging program of the PDP - 10) which is initially in the low segment of the AMBIT/L interpreter is not overwritten as it is normally. When this switch is included, the size of the stack for the run is fixed at 1000 words The user must be sure that the low segment size is made large enough to accommodate DDT along with everything else. DDT and its associated symbol tables occupy approximately 6K of the low segment. DDT is used by AMBIT/L systems programmers to adjust default values for interpreter initialization and to help track down bugs in the interpreter or strange ones in users' programs.

After any number of the above switches are included on the command line, the line is terminated by a carriage return. There is no provision for continuation of a command across more than one line. As already indicated, any error detected in the command line will be reported by the typing of a question mark on the terminal followed by another prompting asterisk on a new line indicating the user should try again. Other types of errors cause more informative diagnostics, and most of these are fatal.

The program then begins execution. If the /BRK switch was included on the command line a DAMBIT/L user break occurs right away; otherwise the AMBIT/L user program is "off and running". Other than the output typed by the running program the user may find only a few other kinds of typing which are diagnostics of the system. Throughout execution, the AMBIT/L interpreter performs extensive checking on the correctness of the program it is interpreting and on the AMBIT/L data with which it is working; various internal consistency checks are also performed. If an error condition is detected by the interpreter, the interpreter immediately reports it to the user by typing a diagnostic message on the terminal which includes a one-to-three-character mnemonic indicating the type of error. A few error conditions are so serious that a return to the PDP - 10 Monitor follows the typing of the message. Usually, however, control is transferred to the DAMBIT/L debugging system. A complete list of all error messages of this type is given in Section M, "Error Traps".

Another type of error which may cause a diagnostic message to be typed is an input/output error. This class of errors is handled differently since a true error trap occurs and an indirect function call is made by the system via the System POINTER TRAP.IO. The function called to service an input/output error trap may be supplied by the programmer. As a default setting, however, the POINTER TRAP.IO points to a FUNCTION node corresponding to a built-in function which types a diagnostic error message on the terminal. As with interpreter-detected error traps, input/output error traps are followed by a transfer of control to the DAMBIT/L debugger at the beginning of the next rule in the user program. Further details on this type of error are described in Section F, "AMBIT/L Input/Output".

While a program is running, the bell on the terminal may ring from time to time. Each ring indicates that a garbage collection is taking place. Excessive ringing of the bell indicates that either there should be more free storage allocated or perhaps the user program has strange characteristics relative to the use of free storage. Further details on this are given in the section on Free Storage Management in Section E, "AMBIT/L Built-in Functions for the Programmer".

When an AMBIT/L program terminates normally or after a fatal error trap occurs, the system types on the terminal the number of Kilo-Core-Seconds (KCS) used and the number of seconds of connect time (CT) or real time used since the program execution began. A KCS is the basic unit of cost in a PDP - 10/50 Time-Sharing System which represents one second of CPU usage per 1K (1024 words) of core memory occupancy.

Under no circumstances should the PDP - 10 Monitor issue an error diagnostic during the running of an AMBIT/L program, such as "ILLEGAL MEMORY REFERENCE". If a user encounters such an error he should report it to an AMBIT/L systems programmer, preferably with sample terminal listings.

Bootstrap Preparation

This memo has already described how the AMBIT/L interpreter must be invoked to run an AMBIT/L program and then the user must type a command line. For a smoother invocation of a commonly used AMBIT/L program, a bootstrap MACRO-10 program may be prepared which is simply invoked by a Monitor command of the form:

```
RUN MYPROG
```

which may optionally be followed by a project-programmer specification in square brackets. Such a command causes MYPROG.SAV to be run which is a one-block bootstrap which directly calls upon the AMBIT/L interpreter at an alternate entry point. The bootstrap program first creates in the disk directory in which the user is currently logged in a one-line text file whose name is of t!ᴈ form <u>xxx</u>AMB.TMP, where <u>xxx</u> is the user's job number. When AMF⁻T is started at the alternate entry point it looks for such a file and expects it to contain a one-line command in the same format as the command line which a user may normally type. After reading the one-line temporary file AMBIT deletes it. Since a bootstrap proᶢ·ᴊm is presumably carefully prepared and tested at least once, it is not expected that aı. initialization error condition will arise. If one does, however, the interpreter proceeds to act as if the command had been typed in, and it either prompts the user with an asterisk or fatally terminates. Not ·:ə that the naive user of a bcotstrap cannot detect that he is running an AMBIT/L program. As an example, the AMBIT/L Compiler is an AMBIT/L program which is run by the normal interpreter; it is invoked by a command of the form:

RUN CO⁾·IPIL [<u>proj</u> , <u>prog</u>]

wherե <u>proj</u> and <u>prog</u> are the project and programmer numbers where the COMIIL.SAV bootstrap is kept.

?:2

For the user or systems programmer who wishes to create a new bootstrap the general bootstrap program is kept as a MACRO-10 source file in the directory where the AMBIT/L Programming System is residing. It is named AMBOOT.MAC. The user may copy that file into his own directory and then edit the one command line according to his needs. The comments in the source program should serve as sufficient guidelines for where the command line is. Under special circumstances a user may wish to alter the name of the interpreter being invoked (perhaps TAMBIT) or the disk directory where the interpreter is being sought. After editing AMBOOT.MAC, the user should type the Monitor command:

```
LOAD AMBOOT.MAC
```

Then after compilation and loading is complete, the user should save the bootstrap under any name he chooses by typing a Monitor command of the form:

```
SAVE MYPROG
```

and thus a bootstrap has been prepared.

(END)

Section M

Error Traps

December 14, 1971

This section provides the AMBIT/L user with a complete list of all error traps which may occur during the running of an AMBIT/L program. Errors are listed alphabetically based on the short mnemonic which is typed on the terminal as part of a diagnostic message.

When an error condition is detected by the AMBIT/L interpreter, an error trap occurs. This causes the AMBIT/L System to type a message on the terminal which includes a one-to-three-character mnemonic indicating the type of error. A few error conditions are so serious that a return to the PDP-10 Monitor follows the typing of the message. Usually, however, control is transferred to the DAMBIT/L debugging system. In the future the non-fatal traps will be implemented as actual traps where a function call is performed so the programmer may substitute his own recovery procedures. Such a trap facility now exists only for the input/output built-in functions.

This section consists of an alphabetic listing of the mnemonics; associated with each one is a one-sentence explanation of the cause or condition of the error. An error condition which causes a fatal error trap is so indicated. There are some error diagnostics which may be printed as a result of improper interpreter initialization; these errors are considered to be in a different category and are therefore not included in this memo.

Most error conditions are caused by an erroneous program. Several error conditions, however, may arise from an internal inconsistency due to a bug in the interpreter itself. The mnemonics for the unexpected error conditions each start with letter 'Z'.

Following the explanation of each error is a letter in square brackets which indicates the interpreter switch (except A) which must be ON for the error condition to be detected. Normally all switches are ON except for G and I. An interpreter with alternate switch settings can only be created by an AMBIT/L systems programmer.

Errors

Letter	When Error is Detected
A	always
C	detection of cycles
D	general debugging mode
G	consistency checks in Garbage Collector
I	internal consistency in interpreter
P	consistency checks in paging system
T	check STRING and TOKEN display list format in TRS and TRT

ACR:	An arithmetic computation involving REALS has produced a result whose magnitude is larger than can be represented by a REAL. [A]
B**:	An attempt is being made to write both the DOWN link and the RIGHT link from the NULL CELL (**). [D]
BS:	An argument to one of the following built-in functions is not a BASIC SYMBOL: AFTER, BEFORE, NEXTB, PREVB. [D]
CAL:	The first argument to the built-in function CAT or LAST is a cyclic list. [C]
CAT:	An argument to the built-in function CAT is not a CELL. [D]
CL1:	The 1st argument to the built-in function COMPARE.LIST is a cyclic list. [C]

CL2:	The 2nd argument to the built-in function COMPARE.LIST is a cyclic list. [C]
CS1:	The 1st argument to the built-in function COMPARE.STRUCT is a cyclic structure. [C]
CS2:	The 2nd argument to the built-in function COMPARE.STRUCT is a cyclic structure. [C]
D**:	An attempt is being made to write the DOWN link from the NULL CELL (**). [A]
DZ1:	An attempt is being made to divide by the INTEGER zero with the DVQ built-in function. [A]
DZ2:	An attempt is being made to divide by the REAL zero with the DVQ built-in function. [A]
DZ3:	An attempt is being made to divide by the INTEGER zero with the DVR built-in function. [A]
DZ4:	An attempt is being made to divide by the INTEGER zero with the DVQR built-in function. [A]
EWK:	An attempt is being made to end a walk by following the RIGHT link from a non-CELL. [D]
F#A:	In a function call the number (#) of arguments given is not the number of arguments expected. [D]
F#R:	In a function call the number (#) of results given is not the number of results expected. [D]

F/?:	A rule has failed where a failure exit label was not provided. [A]
GC:	A garbage collection has occurred which yielded no free CELLs. An attempt was made to transfer control via the system PERM POINTER GCOL.CHOKE, but that POINTER points DOWN to the NULL CELL. This is a fatal trap. [D]
I2:	The 2 arguments to one of the following built-in functions are both REALS, and they must be INTEGERS: AND, DVQR, DVR, OR, XOR. [A]
IF:	An attempt is being made to make an indirect function call via a node which is not a FUNCTION. [D]
IFT:	An attempt is being made to make an indirect function call as a trap via a node which is not a FUNCTION. [D]
ILF:	A reference is being made to an inactive LABEL or FUNCTION. [A]
IR1:	The 1 argument of one of the following built-in functions is neither an INTEGER nor a REAL: ABS, ADD1, EQ0, GE0, GT0, LE0, LSHIFT (first argument), LT0, NE0, NEG, NOT, SQ, SUB1. [D]
IR2:	The 2 arguments of one of the following built-in functions are neither both INTEGERS nor both REALS: ADD, AND, DVQ, DVQR, DVR. GE, GT, LE, LT, MAX, MIN, MUL, OR,SUB, XOR. [D]
ITC:	An indirect transfer of control is being attempted where the walk ends by a RIGHT link pointing to a CELL. [D]
ITN:	An indirect transfer of control is being attempted via a non-LABEL. [D]
ITU:	An attempt is being made to indirectly transfer control to a LABEL which is undefined (by the Link Editor). [A]

LAS:	The argument to the built-in function LAST is not a CELL. [D]
LCL:	The argument to the built-in function LENGTH is a cyclic list. [C]
LNR:	The argument to the built-in function LENGTH is a REAL. [A]
LSH:	The first argument to the built-in function LSHIFT is a REAL, and it must be an INTEGER. [A]
NOT:	The argument to the built-in function NOT is a REAL, and it must be an INTEGER. [A]
PAG:	The paging system cannot load a requested page because it is too large. This is a fatal trap. [A]
R**:	An attempt is being made to write the RIGHT link from the NULL CELL (**). [D]
RAR:	During the execution of the built-in function RANDOM, P.RAND was found to be pointing to neither an INTEGER whose magnitude is less than 2^{35} nor the NULL CELL. [D]
RAS:	During the execution of the built-in function RANDOM, P.SEED was found to be not pointing to an INTEGER whose magnitude is less than 2^{35}. [D]
RDU:	An attempt is being made to read the DOWN link of an undefined (by the Link Editor) POINTER. [A]
RR:	An attempt is being made to read the RIGHT link from a non-CELL. [D]
RWD:	An attempt is being made to read or write a DOWN link from a node which doesn't have one; this may be part of a walk. [A]

STK:	An overflow of the interpreter control stack has occurred. An attempt was made to transfer control via the system PERM POINTER STACK.CHOKE, but that POINTER points DOWN to the NULL CELL. This is a fatal trap. [D]
SWK:	An attempt is being made to take one step of a walk by following the RIGHT link from a non-CELL. [D]
TIC:	The argument to the built-in function TRI or TRR is a list which includes an element which is a BASIC SYMBOL representing an illegal character. [A]
TIS:	The argument to the built-in function TRI or TRR is a list of BASIC SYMBOLS which attempts to represent a number, but with illegal syntax. [A]
TLB:	The argument to a type transfer built-in function or the built-in function LENGTH is a BASIC SYMBOL. [A]
TLF:	The argument to a type transfer built-in function or the built-in function LENGTH is a FUNCTION. [A]
TLL:	The argument to a type transfer built-in function or the built-in function LENGTH is a LABEL. [A]
TLM:	The argument to a type transfer built-in function or the built-in function LENGTH is a MARK. [A]
TNB:	The argument to the built-in function TRI or TRR is a list which includes an element which is not a BASIC SYMBOL. [A]
TRR:	The argument to the built-in function TRR is an INTEGER whose magnitude is larger than can be represented by a REAL. [A]
TRS:	The argument to the built-in function TRS is a list which includes an element which is not a BASIC SYMBOL. [T]
TRT:	The argument to the built-in function TRT is a list which includes an element which is a CELL other than the NULL CELL. [T]

TUL:	There is an attempt to transfer control to an undefined (by the Link Editor) LABEL. This may be due to a S/? exit. [A]
TYP:	A debug-mode type test has failed (corresponding to a use of !). [D]
WB:	An attempt is being made to write both the DOWN link and the RIGHT link from a non-CELL. [D]
WDU:	An attempt is being made to write the DOWN link of an undefined (by the Link Editor) POINTER. [A]
WR:	An attempt is being made to write the RIGHT link from a non-CELL. [D]
WRN:	An attempt is being made to write the RIGHT link of some CELL to a non-CELL. [D]
ZBI:	An attempt is being made to call a non-existent primitive built-in function. [D]
ZBL:	The second argument to the "special" built-in function BLANKS is an INTEGER whose value is greater than 100. [D]
ZD1:	The first argument to the "private" built-in function DECODE is not an INTEGER whose value is 9,11,12,13, 14, 15, 16, or 17. [D]
ZD2:	The second argument to the "private" built-in function DECODE is not an INTEGER whose value is in the proper range, but the first argument is an INTEGER whose value is between 11 and 17. [D]
ZD3:	The second argument to the "private" built-in function DECODE is not a CELL, but the first argument is the INTEGER 9. [D]

ZF1:	The argument to the "private" built-in function I.FL is not an INTEGER whose value is between 0 and 7. [D]
ZF2:	During the execution of the "private" built-in function I.FL more space cannot be allocated for flag links. [A]
ZF3:	The second argument to the "private" built-in function W.FL is not an INTEGER whose value is between 0 and 7. [D]
ZF4:	The first argument to the "private" built-in function R.FL or W.FL is not a CELL. [D]
ZF5:	A call has been made on the "private" built-in function R.FL or W.FL when flag links are not allocated. [D]
ZF6:	During the execution of the "private" built-in function T.FL an error has been detected when attempting to de-allocate space occupied by flag links. [A]
ZFL:	During the execution of the built-in function FLTH a CELL has been found on the free storage list which is not marked as being free. [D]
ZFN:	An attempt is being made to free a STRING or TOKEN name headed by a non-CELL. [D]
ZFX:	A function-exit (FX) operation code has been encountered when not in a function. [D]

ZG1:	An unexpected internal inconsistency has been detected by the Garbage Collector: the root of a tree is not a CELL. This is a fatal trap. [G]
ZG2:	An unexpected internal inconsistency has been detected by the Garbage collector: an improper tree walk has occurred. This is a fatal trap. [A]
ZG3:	An unexpected internal inconsistency has been detected by the Garbage Collector: an attempt has been made to mark a non-CELL. This is a fatal trap. [G]
ZG4:	An unexpected internal inconsistency has been detected by the Garbage Collector: an attempt has been made to test the marking of a non-CELL. This is a fatal trap. [G]
ZG5:	An unexpected internal inconsistency has been detected by the Garbage Collector: an attempt has been made both to test and to mark a non-CELL. This is a fatal trap. [G]
ZGT:	An attempt has been made to get a word from free storage and an unmarked word has been found which probably represents data in use. [D]
ZI:	An argument to one of the following "private" built-in functions is not an INTEGER whose magnitude is less than 2^{35}: DECODE, DVLD, MLLD, TRF, TRL, TRM, TRPD, TRTD, ZSET.INDIC. [D]
ZIN:	An INSERTION is not inserting a block since an INSERT chain does not end in a BE operation. [D]

ZIS:	After allocating some variables on the internal stack there is not enough room left for further use. This is a fatal trap. [A]
ZLE:	The argument to the built-in function LENGTH is illegal data. [D]
ZLI:	An attempt is being made to call a long integer function as a trap, and the system PERM POINTER P.LIFL does not point DOWN to a CELL. [D]
ZND:	The argument to the "special" built-in function NUM.DIGITS is a REAL. [A]
ZO1:	There is insufficient free storage for complete execution of the "private" built-in function OUTSTB. This is a fatal trap. [A]
ZO2:	The "private" built-in function OUTSTB has been given a structure which includes either a LABEL, FUNCTION, REAL, or long INTEGER. This is a fatal trap. [A]
ZO3:	The "private" built-in function OUTSTB has been given a structure which includes a TOKEN. This is a fatal trap. [A]
ZO4:	A basic input/output error has occurred during the execution of the "private" built-in function OUTSTB. This is a fatal trap. [A]
ZOP:	An illegal interpreter operation code has been encountered (such as 0). [A]
ZP1:	An input error has been detected by the paging system while reading an environmental page. This is a fatal trap. [A]
ZP2:	A premature end-of-file has been detected by the paging system while reading an environmental page. This is a fatal trap. [A]

ZP3:	An input error has been detected by the paging system while reading a page of the user program. This is a fatal trap. [A]
ZP4:	A premature end-of-file has been detected by the paging system while reading a page of the user program. This is a fatal trap. [A]
ZP5:	An unexpected internal inconsistency has been detected by the paging system. This is a fatal trap. [P]
ZPC:	In a pop of the control stack an attempt is being made to go to a level higher than the current one. [D]
ZRD:	An attempt is being made to read DOWN to a given long integer, and that operation is not implemented. [A]
ZRI:	A REAL argument has been given to an input/output primitive built-in function. [A]
ZST:	A STRING or TOKEN ring has been found which is not cyclic. [I]
ZT0:	The argument to the "private" built-in function TRCODE is the BIT0 pattern. [D]
ZTC:	The argument to the "private" built-in function TRCODE is a CELL, STRING, or TOKEN. [D]
ZTI:	The argument to the "private" built-in function TRCODE is illegal data. [D]
ZTL:	The argument to a type transfer built-in function or the built-in function LENGTH is illegal data. [A]

ZUT:	An attempt is being made to read DOWN to a node using an unimplemented type test. [A]
ZWD:	An attempt is being made to write DOWN to a given long integer, and that operation is not implemented. [A]

Section N

Using TAMBIT: the AMBIT/L Interpreter with Page Timing Instrumentation

January 13, 1972

This section describes how to use the alternate version of the AMBIT/L interpreter for the instrumentation of timing and paging characteristics of a program. An example is included.

TAMBIT

Use of the normal AMBIT/L interpreter is covered in Section L, "AMBIT/L Program Execution". An alternate version of the interpreter is available for the instrumentation of timing and paging characteristics of a program. This alternate interpreter is called TAMBIT (for timing AMBIT) and is used in exactly the same way as the normal interpreter. Its operation is somewhat less efficient for the interpretation of the INSERT commands and page-changes of any sort. The low segment portion of TAMBIT includes an extra 700_{10} words of tables for keeping track of the instrumentation data. TAMBIT may be used to instrument programs which consist of less than 350 pages (including the 26 built-in environmental pages).

TAMBIT is invoked in the same way as the normal interpreter and at any time after execution of the AMBIT/L program begins the user may obtain a summary of instrumentation data. The user may interrupt execution by typing one or two ↑C's (CTRL C), or he may wait until the interpreter returns control to the Monitor after a termination of execution. Then the user may type one of the following commands to the Monitor:

```
START 140
   or
START 141
```

Using either of these commands will invoke the typing of the instrumentation data collected thus far. If the first form of command is used the data thus far collected is retained; however, use of the second form of command clears out the accumulated data after it has been typed. In either case, after the typing is done control returns to the point from which it was interrupted. Thus if it was in the midst of execution, then execution continues. If it was at Monitor command level, then control returns there.

For each page and for the Garbage Collector (which is called page 0) three statistics are kept and reported:

- the number of milliseconds spent on the page (or in the Garbage Collector) based on the number of clock ticks which occur while on the page; a clock tick occurs once every 16 2/3 milliseconds or 60 times per second.
- the number of times control transferred to the page by an INSERT command, or for page 0 the number of times garbage collection was invoked.
- the number of times the page had to be read from the DMP file since it was not already in memory when needed. This is always 0 for page 0.

The statistics typed are only for those pages where there has been some activity. Also a total is given for each of the three statistics.

Note that a page must be used when a function declared on it is called. This can contribute significantly to the time statistic if the clock ticks happen to occur at function entry or exit.

Since statistics for the 27 environmental pages is also reported, the following key to their use is given.

Page #	Name	Use
1	ZENV1	general long integer arithmetic
2	ZLICM	long integer comparison
3	ZMLLL	long integer multiply
4	ZQRLL	long integer division
5	ZASLL	long integer addition and subtraction
6	ZENV2	general input/output except for what is elsewhere
7	ZIOP	OPEN
8	ZICL	CLOSE
9	ZIDEL	DELETE
10	ZIREN	RENAME
11	ZIRDI	RDINFO
12	ZIZPH	used by OPEN, DELETE, RENAME
13	ZIZOU	OUTS, OUTL, OUTLS
14	ZIZSE	SELWI, SELWO, RDSELI, RDSELO, RDLNGTH
15	ZIMES	input/output error trap messages
16	ZOSTR	OUT.STRUCT
17	ZISTR	IN.STRUCT
18-25	ZENV3, etc.	DAMBIT/L
26	ZINIT	initialization (once only)

This section ends with a sample listing of the statistics from instrumenting an AMBIT/L compilation of a typical insertion. For demonstration purposes it is being run using a somewhat smaller object code paging area than the one normally employed.

```
                                TAMBIT

PAGE    MILLISEC    INSERTS   DISK READS
   0        3700          5          0
   1          17          0          1

   6       18637          1          1
   7         101          5          2
   8           0          3          1

  11          17          1          1
  12           0          5          2
  13        2673         91          9

  18           0          1          1

  26          67          1          1
  27       16058          1          1

  32       13614         65          3
  33         184         32         12
  34        1168        340          1
  35          17          1          1
  36          16          1          1
  37        6957          8          1
  38          33          4          3
  39         978        119          7
  40        3671        338          2
  41         699          4          4
  42         968        403          2
  43           0          4          4
  44          50         10          3
  45          33          8          4
  46        3241        368          1
  47         469         54          3
  48        9932         58          1
  49         417         34         11
  50        2063        287          3
  51          84          5          3
  52        2475        318          6
  53        4432         82         10
  54         771         54          9
  55        3331         54          9
  56        5236         54          8
  57        1884         54          8
  58       14694         54          1
  59        1385        152          7
  60         927         69         11
  61        5729        936          1
  62        2084        453          1
  63         617         82         12
  64          33          3          2

  66         505         54          7
  67         117         54          7

  70        1350          :          1

TOTALS    131434       4731        190

EXIT
```

Section O

FILUT: File Utility

January 12, 1972

This section describes how to use FILUT, a disk file utility program written in AMBIT/L. It permits a user to create, examine, or alter a PDP-10 disk file on a word-by-word basis. The user of FILUT deals in octal numbers only, both for word numbers and contents.

FILUT is a disk FILe UTility program written in AMBIT/L. It permits a user to create, examine, or alter a PDP-10 disk file on a word-by-word basis. The user of FILUT deals in octal numbers only, both for word numbers and contents.

As in AMBIT/L input/output, a disk file is viewed as being composed of a one-way potentially infinite number of 36-bit words, numbered 1, 2, 3, etc. At any time, an initial set of these words (possibly none) are considered to exist with meaningful values, and the remaining are considered non-existent.

FILUT is invoked by a Monitor command of the following form:

RUN FILUT [<u>proj</u> , <u>prog</u>]

The user is prompted for typed input by the program's issuing an initial request of:

```
*FILE=
```

The user is expected to type a file name with or without an extension. The file name must consist of one to six alphanumeric characters, and the extension may consist of zero to three alphanumeric characters. A period (.) is used to separate the name from the extension. If incorrect syntax is used, the program types a question mark and recycles with another "*FILE= " request.

If a syntactically correct file name is given, that file is opened by FILUT for input-output. This implies that if such a file did not previously exist, it is created. The program then types out "OCTAL LENGTH=" followed by the number of words in the file. If the file has just been created, that number will be zero.

FILUT now types an asterisk (*) on a new line to indicate it is waiting for the user to type a command.

Note: Throughout the remainder of this memo a dollar sign ($) indicates a user has typed ALT (or ESC). The PDP-10 Teletype service routine always echos a dollar sign when the user types ALT (or ESC).

The following command forms are accepted by FILUT. Lower case characters are used to represent any octal number of one to twelve digits. Any illegal command causes FILUT to type a "?" followed by a "*" on a new line. "Short form" refers to printing values of words with leading zeros suppressed. "Long form" refers to printing values of words as 12-digit numbers. FILUT normally prints values in their long form.

Form	Interpretation
x$	EXAMINE WORD x.

If x is 0, this is an error; otherwise EXAMINATION mode is entered. If word x does not exist, "NO-WORD" is typed followed by a "*" on a new line. If word x exists, the word is typed (in either short or long form), and word x is considered to be opened as in DDT. The user may then type one of the following forms:

<CR>	Word x is closed and a "*" is typed on a new line.
$	Word x is closed and an attempt is made to examine word x + 1.
y<CR>	Word x is changed to have the value y, and "*" is typed on a new line.
y$	Word x is changed to have the value y, and then an attempt is made to examine word x + 1.

x-y$ or x-y<CR>	EXAMINE WORDS _x_ THROUGH _y_.
	If _x_ is 0, or if _x_ is greater than _y_, this is an error; otherwise EXAMINATION mode is entered. If word _x_ does not exist, "NO-WORD" is typed followed by "*" on a new line. If word _x_ exists, _x_ is typed on a new line followed by "$" followed by the value of word _x_ (in either short or long form). This format continues on successive lines with word _x_ + 1, word _x_ + 2, etc. until either word _y_ has been typed or the last existent word has been typed. Finally, a "*" is typed on a new line.
x←z<CR>	SET WORD _x_ TO _z_.
	If _x_ is 0, this is an error; otherwise SETTING mode is entered. Word _x_ is set to _z_. If it does not exist, it is created. Then a "*" is typed on a new line.
x-y←z<CR>	SET WORDS _x_ THROUGH _y_ TO _z_.
	If _x_ is 0, or if _x_ is greater than _y_, this is an error; otherwise, SETTING mode is entered. Words _x_ through _y_ are set to _z_. Any non-existent words are created. Then a "*" is typed on a new line.

x←z$	SET WORD x TO z AND PREPARE FOR THE NEXT.

If x is 0, this is an error; otherwise SETTING mode is entered. Word x is set to z. If it does not exist, it is created. Then on a new line x + 1 is typed followed by "←". The user may then type one of the following forms:

<CR>	A "*" is typed on a new line.
w<CR>	Word x + 1 is set to w. If it does not exist, it is created. Then a "*" is typed on a new line.
$	Word x + 1 is not affected, but then on a new line x + 2 is typed followed by "←", etc.
w$	Word x + 1 is set to w. If it does not exist it is created. Then on a new line x + 2 is typed followed by "←", etc.

x-y←z$	SET WORDS x THROUGH y TO z AND PREPARE FOR THE NEXT.

If x is 0, or if x is greater than y, this is an error; otherwise, SETTING mode is entered. Words x through y are set to z . Any non-existent words are created. Then on a new line y + 1 is typed followed by "←". The user may then type one of the following forms:

<CR>	A "*" is typed on a new line.
w<CR>	Word y + 1 is set to w. If it does not exist, it is created. Then a "*" is typed on a new line.
$	Word y + 1 is not affected, but then on a new ine y + 2 is typed followed by "←", etc.
w$	Word y + 1 is set to w. If it does not exist, it is created. Then on a new line y + 2 is typed followed by "←", etc.

<CR> NULL COMMAND.

A "*" is typed on a new line.

$ EXAMINE NEXT WORD OR PREPARE TO SET NEXT WORD.

If FILUT is in EXAMINATION mode, word x + 1 is examined on a new line, where x is assumed to be the most recently examined word. If FILUT is in SETTING mode, it types out x + 1 followed by "←" on a new line, where x is the most recently set word. The user may then type one of the forms permitted in the "x←z$" command.

S<CR> PRINT VALUES IN SHORT FORM.

Leading zeros are suppressed on further typing out of values of words.

L<CR> PRINT VALUES IN LONG FORM.

Further typing out of values of words is done as 12-digit numbers.

E<CR> END THE SESSION.

This command should always be used to terminate a session with FILUT in order to guarantee that all changes the user has made to a file are properly completed.

(END)