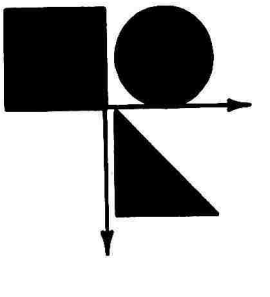
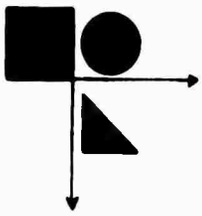
AN INTRODUCTION TO AMBIT/L, A DIAGRAMMATIC LANGUAGE FOR LIST PROCESSING

by

Carlos Christensen

Massachusetts
COMPUTER ASSOCIATES
division of
APPLIED DATA RESEARCH, INC.





APPLIED DATA RESEARCH, INC.

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

AN INTRODUCTION TO AMBIT/L, A DIAGRAMMATIC LANGUAGE FOR LIST PROCESSING

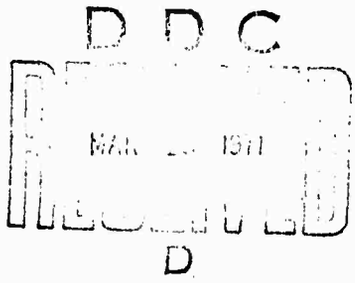
by

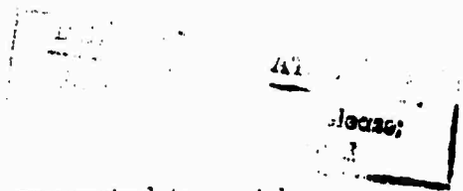
Carlos Christensen

THIRD EDITION

CA-7102-2211

February 22, 1971





The work reported in this paper was supported in part by the Advanced Research Projects Agency of the Office of The Secretary of Defense under ARPA Order No. 1228.

ABSTRACT

AMBIT/L is a list-processing programming system. It integrates the general use of recursive functions with a pattern-matching style of programming. Two-dimensional directed-graph diagrams are used to represent the data, and similar diagrams appear throughout the program as the "patterns" of rules. The system has a simple core, but extends out to accomodate the always complicated requirements of input-output, traps and interrupts, and storage management; it is a large system. The PDP-10 implementation of AMBIT/L is described in this paper.

Note on Publication History: The first edition of this paper was quite formal and complete but was not a good introduction and is out of print. The second edition was more introductory and is being published in the Proceedings of SYMSAM II, the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March 23 - 25, 1971. The third edition is a slight correction and improvement to the second and has, as appendices, the useful formal definitions of the first edition.

CONTENTS

INTRODUCTION

AMBIT/L is a programming system for list-processing. It is used for writing essentially non-numeric programs which operate on data structures of moderate or extreme complexity. This applications area includes artificial intelligence, graphics, algebraic manipulation, and such software as compilers, interpreters, simulators, and editors.

A remarkable and unique feature of AMBIT/L is its use of two-dimensional diagrams. Both the data and the program are represented by diagrams in which conventional text plays only a secondary role. The data diagram depicts the current relations between variables and values, and lines in this diagram move as program execution proceeds. The program diagram uses patterns of data in order to reference and modify the data. These diagrams are not informal aids or supplementary documentation; they are the representations used for communication with the computer.

The value of a diagram in almost any context is familiar. But, in computer programming, the use of diagrams for data can have quite spectacular effects. In diagrammatic form, a collection of data can become an almost machine-like object, changing frequently in certain places but relatively fixed in others. An apparently difficult algorithm can be made easy by the selection of an appropriate structuring and constraint of the data.

The development of AMBIT/L has included some pleasant surprises. The expected heavy demand for an elaborate graphics terminal did not arise; instead, an ordinary teletype (or line printer) proved adequate to "draw" working diagrams using the ordinary characters. Problems with formatting

were eliminated by simple conventions which just happened to work for AMBIT/L. And the language structure seemed to flourish generally in the absence of the problems of delimiting, separating, and grouping which afflict textual programming languages.

In terms of more familiar languages, AMBIT/L can be viewed as operating on list-structured data resembling that of LISP [1]; incorporating the pattern-matching style of programming which is characteristic of SNOBOL [2]; adopting the block-structured program framework of ALGOL 60 [3]; and providing trapping and interrupt facilities found in PL/I [4].

Another view of AMBIT/L is to consider it a specialized version of AMBIT/G [5,6,7,12], a language of much greater theoretical interest and, for the present, of lesser practical value. This view is both historically and logically correct. AMBIT/L followed AMBIT/G by about two years and a future objective is to derive AMBIT/L from AMBIT/G by a completely formal sequence of "constraints" on the program and data of AMBIT/G.

We have remarked that AMBIT/L operates on list-structured data resembling that of LISP. In fact, however, AMBIT/L data is a substantial generalization of LISP data, and operations can be performed which would, at best, be shady practice in LISP. An AMBIT/L program can create and use circular lists in a natural way. And a program can preclude or reduce automatic garbage collection by gathering up cells which are known to be free and returning them "manually" to the free list.

In every pattern-matching language the method of scanning the data is of vital importance. Two very efficient mechanisms for pattern-matching are built into AMBIT/L. First, the rules of the language beguile the programmer into writing "accessible patterns", that is, patterns which can be interpreted as a specific set of "walks" through the data structure which do not require backing up and trying again. Second, the language includes convenient facilities for symbol table management and associative addressing which are implemented by well-concealed hash-code addressing mechanisms. The virtue of the overall design is that these efficiency mechanisms do not distort or complicate the fundamental idea of diagrammatic pattern matching.

The AMBIT/L programming system has been operational on a Digital Equipment Corporation PDP-10/50 time-sharing computer since September 1969. The system includes its own diagram generator, compiler, link editor, interpreter, and debugging facility.

AMBIT/L has been used to implement its own compiler (which was bootstrapped), its own debugging system, and certain of its built-in functions such as those for input-output and for long-integer arithmetic. But the most notable application of AMBIT/L has been to IAM.

IAM [8] is a large and complex system for interactive algebraic manipulation. IAM is written entirely in AMBIT/L. As of July 1970, it was composed of 301 separately compiled AMBIT/L program blocks; it has block nesting to a depth of 13 levels; has a listing of about 1500 pages of two-dimensional diagrams; and compiles to reentrant interpretive object code about 40,000 36-bit words long.

This paper describes those aspects of AMBIT/L which are unusual and important. It provides an outline of the language, some of the design motivation, and some examples. It is an appropriate basis for evaluation of the language; but it should, ideally, be supplemented by a few moderately large example programs. There are no prerequisites for the paper; but the reader without experience with ALGOL 60 and LISP may find certain passages difficult.

THE DATA

It is usual to regard the data of a high level language as a vague abstraction. For example, the ALGOL 60 Report [3] set a high standard for precision in its description of program syntax; but the report did not give a description, precise or otherwise, of the syntax of the data on which a program operates. The philosophy is "the programmer takes care of the program and the program takes care of the data". This view is appropriate and effective for simple data structures.

The AMBIT languages were designed for applications which require complicated data structures. A principal feature of each of these languages is a carefully designed and precisely defined representation for the data on which it operates: string structures for AMBIT/S [9], list structures for AMBIT/L, and general structures for AMBIT/G [5]. The philosophy becomes "choose the right representation for the data and programming will be easy". This dictum must be applied twice; first, the system designer must provide a good facility for data representation and, second, the programmer must use that facility effectively.

An AMBIT/L program operates on a single object called the data graph. This object is represented as a diagram of the sort called directed graph by mathemacticians, and the term "graph" is used in this sense.

The form of the AMBIT/L data graph will be described by giving the form of the universal data graph of AMBIT/G and then applying certain constraints to produce the form of the specialized data graph used by AMBIT/L. This approach provides a foundation for the design of AMBIT/L

and, at the same time, relates the development of AMBIT/L to the current work on AMBIT/G.

The Universal Data Graph

A data graph is a diagram composed of nodes and links. The diagram is thought of as being in a vertical plane so that directions up, down, left, and right can be applied.

A node is a node boundary and an associated node name. A node boundary is a rectangle with a horizontal base, with a height which is uniform for all nodes in a given diagram, and with a well-proportioned width. A node name is a pair of character sequences, the type and subname. The type is written along the upper edge of the node and right-justified on that edge; and the subname is centered inside the node boundary.

We note that it would have been simpler to use a single character sequence as a node name. However, the type-subname pair is used as a node name because it permits the organization of nodes into subsets according to type. This, as we shall see, is useful both in specifying constraints and in writing programs.

A link is a line segment, straight or curved, with an arrowhead at one end. The plain end of the link is the tail, and the arrowhead end is the head. Both ends must lie on a node boundary; the node whose boundary is at the tail of the link is the origin, and the node at the head is the destination. A link is said to point from its origin and point to its destination.

Each link in a data graph has a link name associated with it. In AMBIT/G this link name can, in general, be an arbitrary node; it is almost like a second origin of the link. In contrast, we shall see that the AMBIT/L link name is a highly specialized object. Since our main interest is in AMBIT/L, we will not describe the AMBIT/G notation for link names. It is sufficient to assume the existence of some means to associate a link name with each link.

The data graph must contain exactly one node for every distinct node name: furthermore, each node must be the origin of exactly one link for every distinct link name. This requirement gives rise to the two important properties of the data graph, functionality and permanence.

> Functionality. A link name and an origin determine no more than one destination. That is, a given node can be the origin of no more than one link of a given name.

This restriction models the fact that most memory devices produce a single value for a given address. Together with a complementary restriction on AMBIT/L programs to be discussed later, it is the basis for the efficient interpretation of AMBIT/L programs.

> Permanence. Only the destination of a link can change during program execution. That is, links and nodes are not created or destroyed.

This property models the permanence of memory hardware, which persists beneath various schemes for paging or allocation. It greatly simplifies the programming language since it eliminates the fundamental tests and actions which would otherwise be required by the creation and destruction of nodes and links.

A data graph contains a special node called the undefined destination. A link is undefined if its destination is this node. A node is undefined if every link of which it is the origin or the destination is an undefined link. By convention, the omission of a link or a node from a data graph is equivalent to its inclusion as an undefined link or node.

This convention is important because it permits a finite representation of a data graph which would otherwise have infinitely many nodes and links. In particular, if all links are assumed to be undefined at the beginning of program execution, then the data graph can be represented as an empty diagram.

The abstraction of a data graph is a set of triples. Each triple corresponds to one link in the data graph diagram and consists of (1) the node name of the origin of the link, (2) the name of the link, and (3) the node name of the destination of the link. Each node name is a pair of character sequences, the type and subname. Each link name is a character sequence.

The purpose of the abstraction is, of course, to make clear what is and is not formally essential in the diagram of a data graph. It is apparent, for example, that the length or curvature of a link is not important so long as we can interpret it as a triple of the proper form.

The data graph as defined above is said to be universal; that is, there are no restrictions on the destinations of its links. The data graph could be used in this form; but to do so would be to fail to recognize the special properties or constraints which a given program imposes on its data.

The Constraints

A constraint restricts the set of nodes to which a certain link may point. A variety of constraints may be used. For example, a constraint may take a link out of circulation by requiring that it will always point to the undefined destination node. Another kind of constraint may require that a link point to a node whose type is a given character sequence. More complicated constraints establish dynamic relations among two or more links.

The AMBIT/G system permits the user to begin with the universal data graph and apply constraints to design his own data structures. Such a system is so general that it is difficult to build and to use, and it has never been completely implemented.

In contrast, the AMBIT/L system presents the user with a set of built-in constraints which are not under his control and which produce a specialized data graph suitable for list processing applications. These constraints are given informally in this section. When these constraints assert that certain links or nodes are not allowed or do not exist, the intended

The data graph must contain exactly one node for every distinct node name; furthermore, each node must be the origin of exactly one link for every distinct link name. This requirement gives rise to the two important properties of the data graph, functionality and permanence.

> Functionality. A link name and an origin determine no more than one destination. That is, a given node can be the origin of no more than one link of a given name.

This restriction models the fact that most memory devices produce a single value for a given address. Together with a complementary restriction on AMBIT/L programs to be discussed later, it is the basis for the efficient interpretation of AMBIT/L programs.

> Permanence. Only the destination of a link can change during program execution. That is, links and nodes are not created or destroyed.

This property models the permanence of memory hardware, which persists beneath various schemes for paging or allocation. It greatly simplifies the programming language since it eliminates the fundamental tests and actions which would otherwise be required by the creation and destruction of nodes and links.

A data graph contains a special node called the undefined destination. A link is undefined if its destination is this node. A node is undefined if every link of which it is the origin or the destination is an undefined link. By convention, the omission of a link or a node from a data graph is equivalent to its inclusion as an undefined link or node.

This convention is important because it permits a finite representation of a data graph which would otherwise have infinitely many nodes and links. In particular, if all links are assumed to be undefined at the beginning of program execution, then the data graph can be represented as an empty diagram.

The abstraction of a data graph is a set of triples. Each triple corresponds to one link in the data graph diagram and consists of (1) the node name of the origin of the link, (2) the name of the link, and (3) the node name of the destination of the link. Each node name is a pair of character sequences, the type and subname. Each link name is a character sequence.

The purpose of the abstraction is, of course, to make clear what is and is not formally essential in the diagram of a data graph. It is apparent, for example, that the length or curvature of a link is not important so long as we can interpret it as a triple of the proper form.

The data graph as defined above is said to be universal; that is, there are no restrictions on the destinations of its links. The data graph could be used in this form; but to do so would be to fail to recognize the special properties or constraints which a given program imposes on its data.

The Constraints

A constraint restricts the set of nodes to which a certain link may point. A variety of constraints may be used. For example, a constraint may take a link out of circulation by requiring that it will always point to the undefined destination node. Another kind of constraint may require that a link point to a node whose type is a given character sequence. More complicated constraints establish dynamic relations among two or more links.

The AMBIT/G system permits the user to begin with the universal data graph and apply constraints to design his own data structures. Such a system is so general that it is difficult to build and to use, and it has never been completely implemented.

In contrast, the AMBIT/L system presents the user with a set of built-in constraints which are not under his control and which produce a specialized data graph suitable for list processing applications. These constraints are given informally in this section. When these constraints assert that certain links or nodes are not allowed or do not exist, the intended

meaning is that these links or nodes are constrained to remain in the undefined state.

Only two link names, vertical, and horizontal are allowed. If the tail of a link lies on the upper or lower side of a node boundary, then the link is named vertical. If the tail of the link lies on the left or right side of a node boundary, then the link is named horizontal. It is convenient to allow the use of down and right as synonyms for these link names, since these words suggest the directions in which the links are usually drawn.

Only eleven distinct types, to be listed in the following paragraphs, are allowed; and special constraints are associated with each. An allowed type is a single letter, chosen so that it is the initial letter of a word used to describe the nodes involved.

A mark node has type M. Its subname must be an identifier. A mark is terminal; that is, it cannot be the origin of a defined link. Furthermore, aside from being distinguished as marks, the only property of mark nodes is their existence as mutually distinct link destinations. For example, a program cannot test for the presence of a certain letter in the subname of an arbitrarily selected mark.

A basic symbol node has type B. Its subname must be the character '%' followed by (1) a single printing character or (2) a multi-letter name for a single non-printing character. Thus, for example, the subnames %A, %$, and %CR represent the first letter of the alphabet, the dollar sign, and the (non-printing) carriage return, respectively. The set of basic symbols is finite; the present implementation provides a subset of the 128 ASCII characters. The basic symbols are terminal, but AMBIT/L has built-in functions which accept them as arguments and thus establish semantic content. Most important are the input-output functions, which associate each basic symbol with appropriate typographical symbols.

A function-name node has type F. Its subname must be an identifier. It is terminal. Each function-name node is permanently associated with a function definition. This definition may be part of the user program being

executed or it may be "built-in" as part of the system. In either case, a function-name node can be "stored" as the destination of a link and later used to specify an indirect call on a function.

A label node has type L. It is similar to a function-name node, except that it is associated with a label in the program being executed and is used for indirect transfer of control.

An integer node has type I. Its subname must be a decimal integer. Only one node is permitted for each numerically distinct number; for example, 13 is allowed but +13 and +013 are not. It follows that two links have numerically identical destinations only if they point to the same node. The present implementation accommodates, for all practical purposes, all possible integers. An integer node is terminal.

A real node has type R. Its subname must be a construct similar to the floating-point constant of FORTRAN. Only one node is permitted for each distinct real number. The present implementation is limited to a finite subset of the real numbers. A real node is terminal. Together, the integer and real nodes constitute the numeric data of AMBIT/L, and the system has the appropriate built-in functions to operate on these nodes.

A pointer node has type P. Its subname must be an identifier. Each pointer node has a vertical link, and the node which is the destination of this link can be called the "value" of the pointer node. Pointers are similar to the "variables" of familiar programming languages. In particular, AMBIT/L programs have a block structure which scopes pointer nodes exactly as ALGOL 60 block structure scopes variables. On the other hand, AMBIT/L does not have a type-declaration to restrict the values of pointers and takes a simpler, less efficient approach to numeric processing then ALGOL. In the present implementation, a pointer node cannot be the destination of a link; but this constraint now appears to be superfluous.

A string node has type S. Its subname must be a single quote followed by an arbitrary character sequence followed by a single quote. Any of the characters available in the implementation may be used, and

conventions are provided for mentioning the single quote and the non-printing characters. Each string node has a vertical link. The string nodes are a variant of the pointer nodes and are used for symbolic indexing of the tables associated with text processing in general and compilation in particular. A string node can be used as a "word" of text and its link can point to a representation of whatever is known about the word.

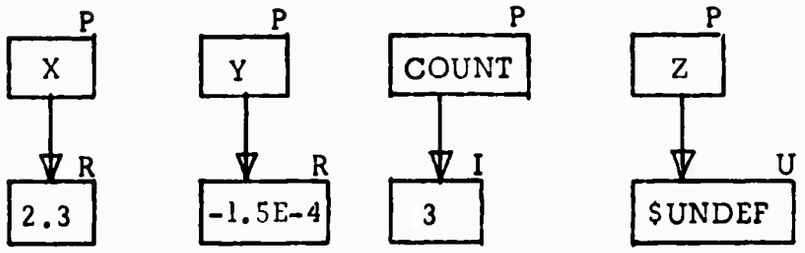
A token node has type T. Its subname must be a left parenthesis followed by a sequence of subnames followed by a right parenthesis. Except for a few uninteresting restrictions, the subnames may be those of any types of nodes, including other tokens. The tokens are an obvious generalization of the strings; the subname is composed of subnames rather than single characters. One intended use of tokens is to implement subscripted variables; more generally, they permit a list of nodes to determine the name of a single node.

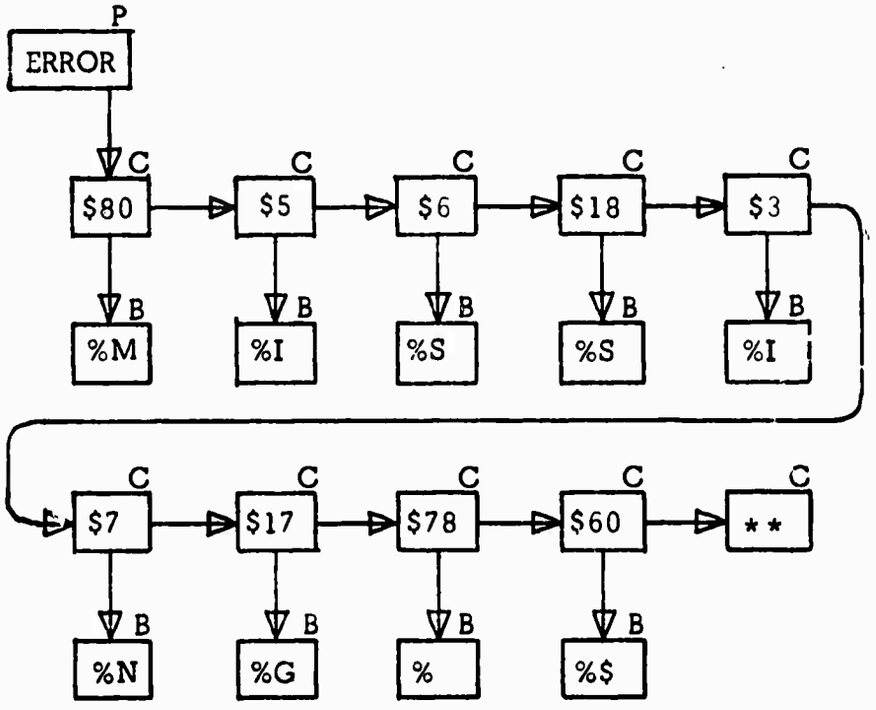
A cell node has type C. The subname of one cell, the null cell, must be a double asterisk, '**'. The subname of every other cell must be a dollar sign followed by an unsigned integer. Each cell has one horizontal link and one vertical link. The horizontal link must point to a cell node or the undefined node; this confines the horizontal links to the formation of a list, while the vertical links are used to point to the (unrestricted) values of the elements of the list. The cell is the only node with more than one link, and it is the only means for linking together a general data structure.

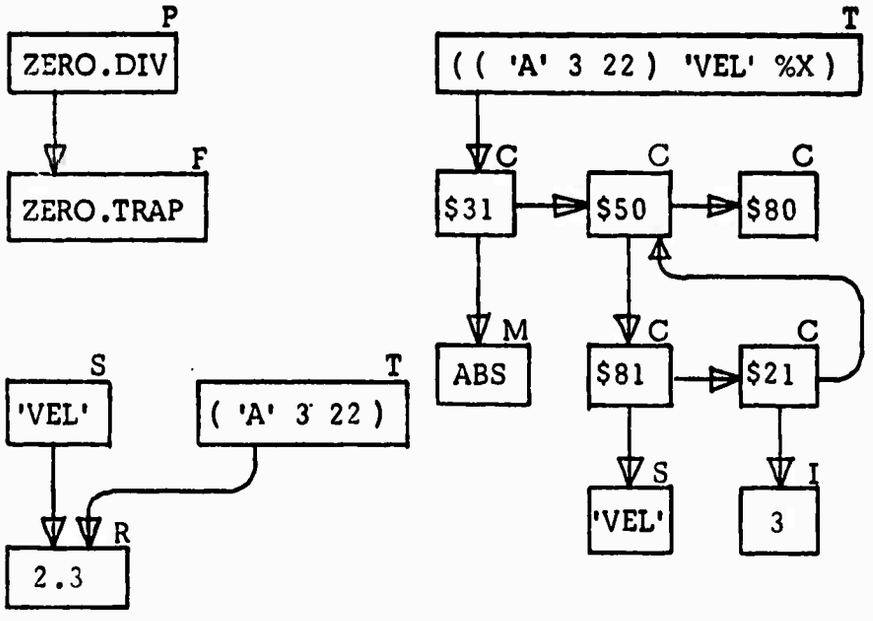
The undefined-destination node has type U and subname $UNDEF. It is terminal. As we have seen, this node is used to establish the undefined state of links and nodes.

Example

We now give an illustration of the data graph as it might appear at some point during the execution of a program.


P
X
R
2.3
P
Y
R
-1.5E-4
P
COUNT
I
3
P
Z
U
$UNDEF


P
ERROR
C
$80
C
$5
C
$6
C
$18
C
$3
B
%M
B
%I
B
%S
B
%S
B
%I
C
$7
C
$17
C
$78
C
$60
C
**
B
%N
B
%G
B
%
B
%$


P
ZERO.DIV
F
ZERO.TRAP
T
(('A' 3 22) 'VEL' %X)
C
$31
C
$50
C
$80
M
ABS
C
$81
C
$21
S
'VEL'
I
3
S
'VEL'
T
('A' 3 22)
R
2.3

The top row of this data graph has four pointers which play the role of variables; each has a down link to a node which is its current value. Thus P/X has as its value the real node R/2.3, P/Y has value R/-1.5E-4 (which is -0.00015), and so on. The rightmost variable and value are redundant since the destination U/$UNDEF would have been assumed for the link if it had not been shown.

The next five rows are a single structure; namely, the pointer P/ERROR whose value is a list of nine basic symbols (which spell out "MISSING $"). The use of the null cell, C/**, to terminate a list is standard practice. The next row begins with a pointer whose value is a function-name. This value exemplifies the fully general capability of the language for function (and label) variables.

In the lower-left corner are a string node and a token node. As it happens, both have R/2.3 as current value. Note that P/X also has this value and that, although R/2.3 appears in two places, it is considered to be the same node. We shall see later that although a subname of a pointer is an indivisible object, the subname of a string or token can be broken down or constructed by a program. For example, S/'VEL' could be broken down into a list of three basic symbols, namely B/%V, B/%E, and B/%L. The token shown here has three parts, and could be the name of an element in a two-dimensional array named S/'A'.

At the lower right is a token with a complicated subname. Its value is a list whose first element is a mark, whose second element is a sublist, and whose remaining elements are the value of P/ERROR. Note that the sub-list ends by pointing back at its "father" cell. There is <u>no restriction</u> against such forms of reentrance or circularity in the data graph.

The abstraction of the data graph is the following set of triples:

```
(P/X, down, R/2.3)
(P/Y, down, R/-1.5E-4)
(P/COUNT, down, I/3)
(P/Z, down, U/$UNDEF)
(P/ERROR, down, C/$80)
(C/$80, down, B/%M)
(C/$80, right, C/$5)
(C/$5, down, B/%I)
(C/$5, right, C/$6)
... and so on.
```

We have just discussed the "meaning" of a particular data graph in considerable detail without exhibiting the program which, presumably, created and operated on that data. AMBIT/L emphasizes this self-contained and independent existence of the data.

THE RULE

An AMBIT/L program is a simple framework in which a collection of rules is embedded. The framework is derived from the block structure of ALGOL 60 and will be discussed briefly in the next section. The rules are the executable units of the language and will be described in this section.

Each rule is a diagram. This section gives the structure (syntax) and interpretation (semantics) of rules. The description of the structure is based on the previously given definition of the data graph. The interpretation of rules is given in an intuitive and heuristic style which, we believe, is most natural for a human interpreter. Only after this interpretation is complete is the important question of uniqueness and efficiency of rule interpretation discussed.

A rule is not an unconditional command; rather, it instructs the interpreter to attempt to perform a process which may turn out to be impossible. A rule says "try to find such-and-such a structure in the data graph and then change it to be thus-and-so". The program logic (flow of control) of AMBIT/L is based on this heuristic property of rules. As we shall see in the next section, a program associates with each rule two successor rules. One of these is interpreted next according as the current rule succeeds or fails.

For purposes of definition, we will assume that a desired rule is generated by creating a data graph and then applying some (zero or more) extension operations to the diagram. These operations are:

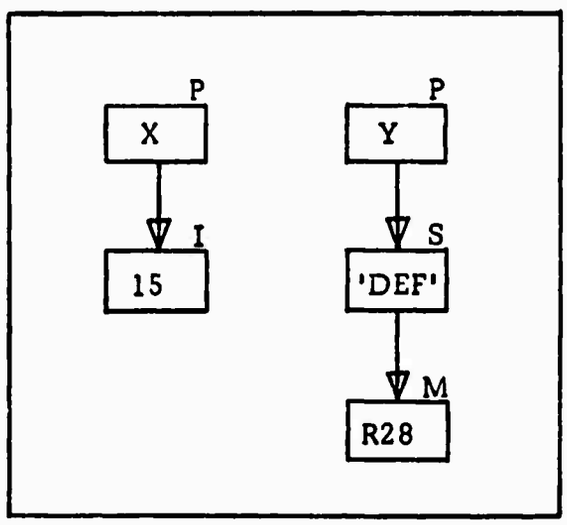
generalize a subname,
generalize a type,
add a modification link,
add a function link, and
add a flow link.

Note well that the notion of "extending" a data graph to produce a rule is only a method of describing the structure of rules. A programmer will, of course, draw a rule directly in its final form; and rules themselves are certainly not modified in this way during program execution.

Make No Extension

In the simplest case, a data graph can be used as a rule without any change. For example, consider the example rule,

E1.



This rule has the same form as a data graph and its abstraction is the set of three triples

(P/X, down, I/15)
(P/Y, down, S/'DEF')
(S/'DEF', down, M/R28)

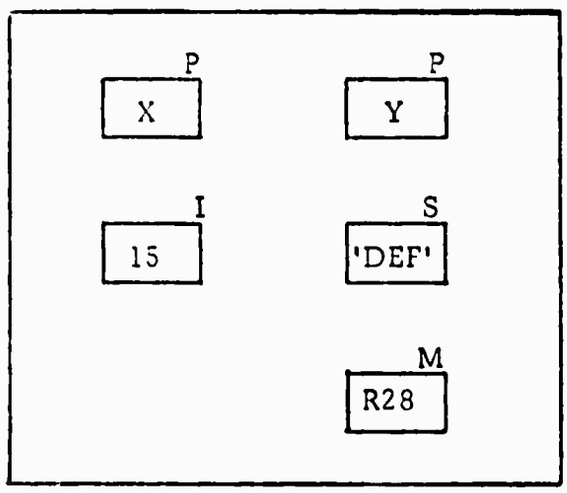
Generally speaking, a rule of this kind (unextended data graph) is interpreted as follows:

Try to find a set of links in the data graph which is identical to the set of links in the rule.

Such a rule is a pure test; its execution does not modify the data but can affect the flow of control by choice of successor. (Forms of rules which do modify data are discussed below.)

Consider, as a second example, the same rule with all links removed, namely

E2.



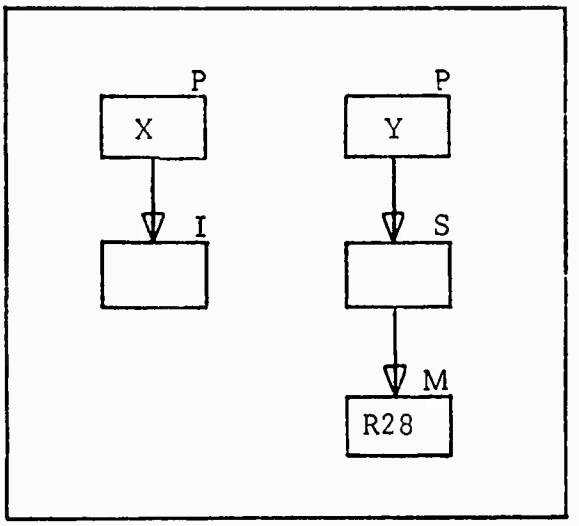
This diagram does satisfy the definition of the data graph although its abstraction is the null set (set of no triples). The rule always succeeds and it therefore a kind of "no-operation"; but we will refer back to it when modification links are discussed.

Generalize a Subname

A subname may be deleted from a rule to produce a new, more general rule. By repeated application of this operation, a rule with several deleted subnames is produced.

For example, starting with rule E1 above (which has the form of a data graph), we delete two subnames, 15 and 'DEF', and obtain the rule

E3.



This rule says "try to find the following in the data graph: first, a node P/X which points down to an (arbitrary) integer node and, second, a node P/Y which points down to an (arbitrary) string node which, in turn, points down to M/R28". More generally, a rule of this kind is interpreted as follows:

> Make a copy of the rule and try to bind each node (fill in the missing subname where necessary) so that for each link in the rule there is an identical link in the data graph.

Such a rule is still a pure test; but it can "match" many different states of the data graph and is, in this sense, "generalized".

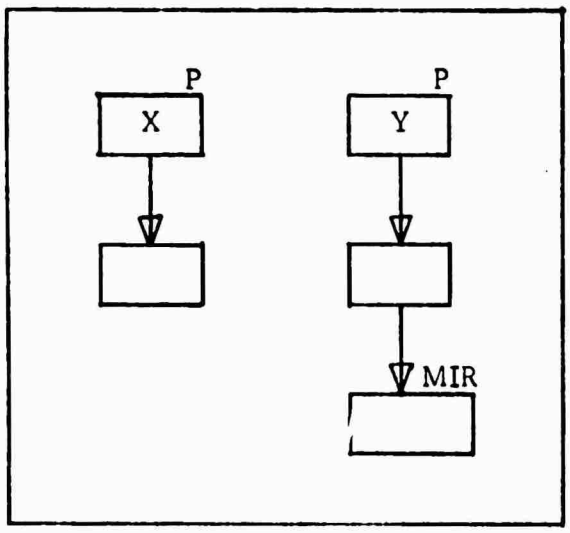
Generalize a Type

If the subname of a node is blank (because of the generalization of the subname), then the type of that node may be deleted or may be replaced by a type-set. A type-set is any subset of the allowed types (except U) in any order. For example, MIR is a type set and specifies a node which is a mark, integer, or real.

For example, starting with rule E3 above, we delete two types and a subname and then replace a type with a type-set. This gives E4.

In E4 the nodes in the left column are redundant; they assert that the down link from P/X points to some arbitrary node. Since the only test in AMBIT/L is a test for the destination of a link (and not the existence of a link or a node), these nodes test nothing; but the rule is nevertheless legal. The middle node on the right has a blank type, which suggests that it may match any type of node. However, since the node has a down link, the type-set CSTP is implied.

E4.

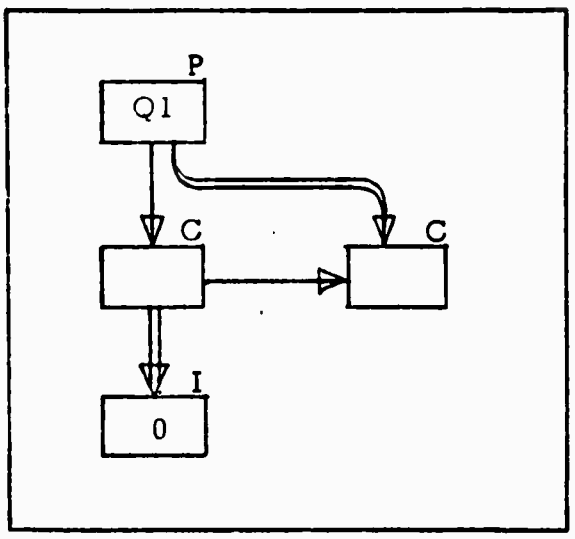


We now will expand the interpretation of rules by expanding the definition of binding. As used previously, binding simply meant filling in a blank subname. Now it means (1) filling in a blank subname, (2) filling in a blank type, and/or (3) deleting all but one type from a type-set.

Add a Modification Link

Modification links are the means by which a rule specifies the modification of the data graph. A modification link resembles an ordinary link except that it is (at least partly) a double line. One modification link can be added to a rule wherever an ordinary link could or does appear in the rule. As an example of the use of modification links, consider the rule

E5.



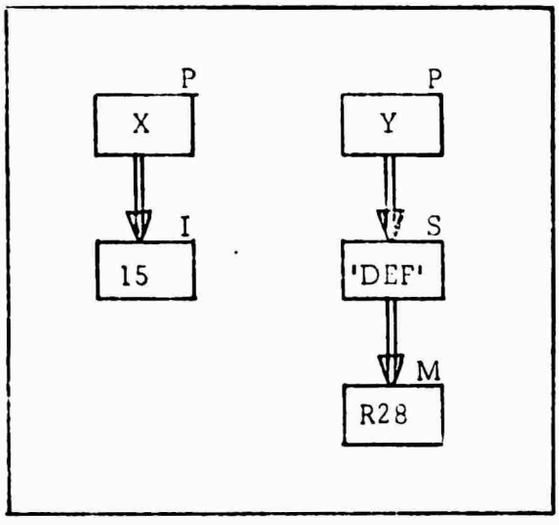
This rule may be read as "the pointer Q1 selects some element (we don't care what its value is) of a list; set that element to zero and advance Q1 to the next element in the list". Note that the two down links from Q1 represent the "before and after" positions of the link and suggest movement of the link. A remarkable aspect of AMBIT/L is that, because of the freedom of two-dimensional notation, the "before and after" can be included in a single diagram. This contrasts with the case of SNOBOL and AMBIT/S, for which two one-dimensional "diagrams" are required for each rule.

The general interpretation of rules is now expanded to read

> Make a copy of the rule and try to bind all nodes so that for each ordinary link there is an identical link in the data graph. If you succeed, modify the data graph so that for each modification link in the rule there is an identical (ordinary) link in the data graph.

As a second example, consider a rule which has only modification links, namely

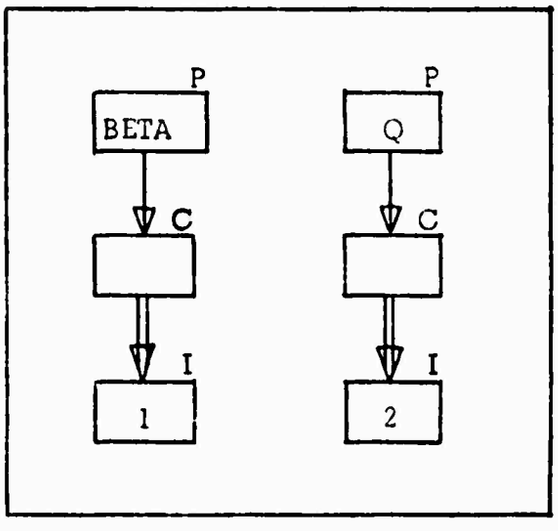
E6.



The matching phase of this rule always succeeds (see the discussion of E2), and it is the sort of rule which is used to initialize the data graph.

As a third example, consider a rule of innocent appearance,

E7.



Ordinarily, this rule will set the down links of two cells to the integers I/1 and I/2. But if BETA and Q happen to point to the same cell, then the setting of the down link is undefined; it is either I/1 or I/2 according to the order in which the two modification links are processed. Several years of experience

indicate (surprisingly) that once programmers are aware of this possibility, serious difficulties are very unlikely.

Add a Function Link

Function links are the means by which subroutines--both those built into the system and those defined in the program--are called as part of the interpretation of a rule.

To add a function link to a rule, select a node of type F and proceed as follows:

1. Determine the number of arguments, m, and results, n, associated with the function being called.
2. Select one of the four sides of the node boundary and make this side into a double line; the result if a call node.
3. From distinct points along the doubled side, draw m ordinary links to appropriate destinations. Consider these links ordered by their origins (from left to right or top to bottom), and call their destinations the origins [sic] of the function link.
4. From distinct points along one of the non-doubled sides, draw n ordinary links to appropriate destinations. Consider the links ordered by points of origin and call their destinations the destinations of the function link.

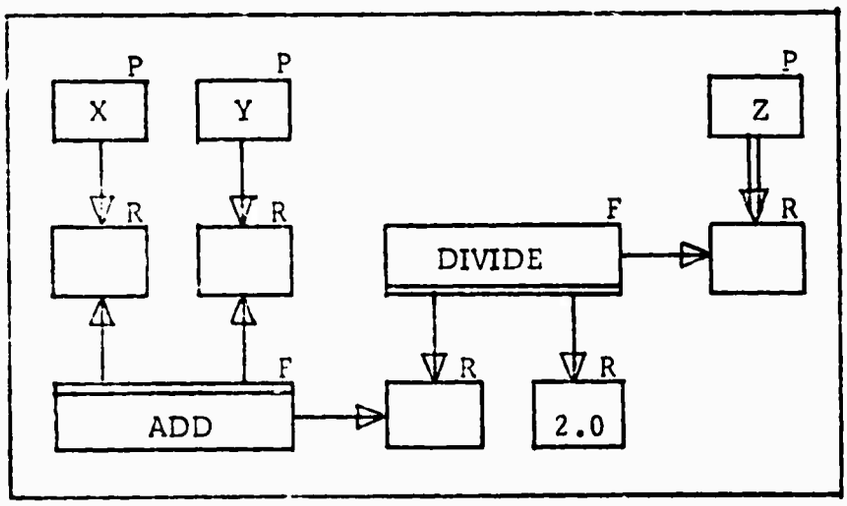
The origins and destinations of a function link are, in fact, the arguments and results of the function, respectively. Given the names of the origin nodes, the definition of the function uniquely determines the names of the destination nodes. If a function is built-in, its definition is found within the language definition itself. If a function is user-defined, its definition is a subprogram included in the same program as the function call.

We now give the general interpretation of rules for the final time.

Make a copy of the rule and try to bind all nodes so that (1) for each ordinary link there is an identical link in the data graph and (2) for each function link the origins and destinations are consistent with the definition of the function. If you succeed, modify the data graph so that for each modification link in the rule there is an identical (ordinary) link in the data graph.

Consider the performance of arithmetic as exemplified by the rule

E3.



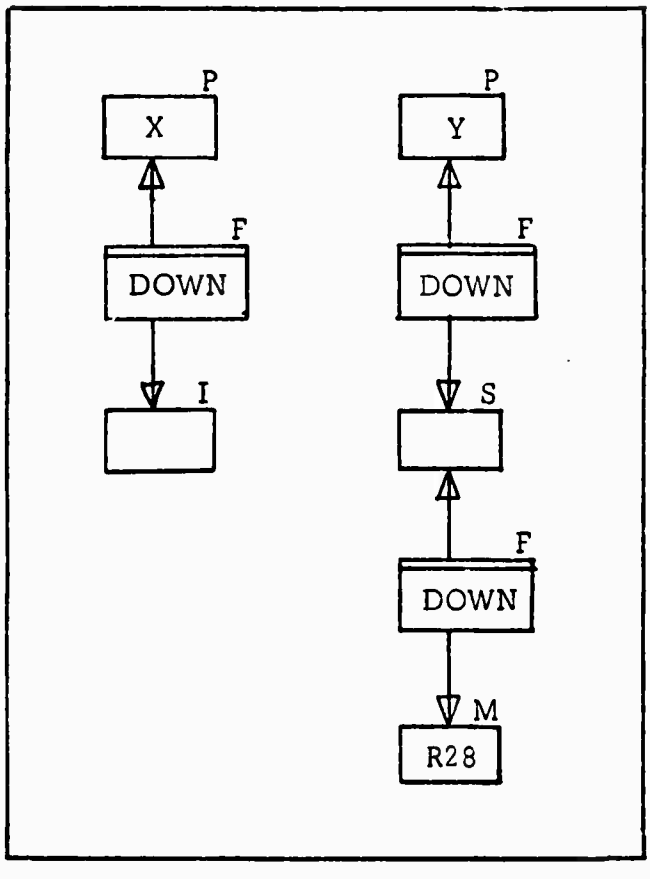
This rule is equivalent to the ALGOL assignment statement

z := (x+y)/2.0

where x, y, and z have been declared real ALGOL variables. The pointers X and Y locate the arguments to ADD; then the ADD function link locates the first argument of DIVIDE; and then DIVIDE locates its result. All this is the binding of nodes. Then the modification link causes the down link of Z to be set to the final result.

Consider next the silly rule

E9.



Note that if the function DOWN is defined as simply reading the down link of its origin to obtain its destination, then this rule is equivalent to E3 above. This example shows how we could, as an exercise, express all rules entirely in terms of function links.

Add a Flow Link

All ordinary and function links are processed before any modification links. Otherwise, however, the interpretation of rules leaves the order of link processing undefined. This vagueness allows the interpreter, human or automatic, to choose an interpretation which reduces the cost of interpretation.

There are circumstances in which the programmer wishes to exercise control over this ordering, however. He may do so in order to make his own

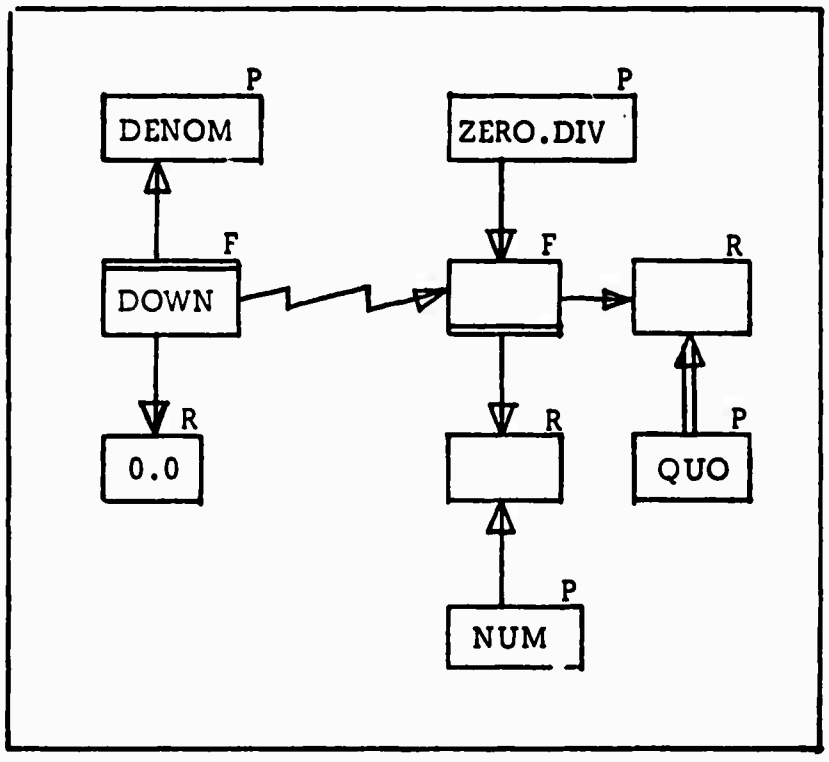
contribution to optimization, or to exclude an interpretation which, because of side-effects of a function, would be an error. For this purpose, the flow link is used.

A flow link resembles an ordinary link except that it is a jagged line (see example). Its tail can lie anywhere on a node boundary because there is no down-or-right distinction among flow links.

A flow link from one call node to another specifies that the function call at the tail of the flow link must be processed before that at the head. We will not discuss the meaning of a flow link whose origin or destination is an ordinary node; the currently implemented interpretation of this case is not very good.

An example of the use of flow links to control program logic is given in E10, below. This rule says "If the down link of DENOM points to zero then perform the function whose name is the value of ZERO . DIV, whose argument is the value of NUM, and whose result will become the value of QUO". The flow link assures that the function will be performed only if the value of DENOM is R/0.0. The rule also illustrates the use of a function name variable, the pointer P/ZERO.DIV.

E10.



Restrictions

We now introduce certain restrictions on the structure of the rule. These restrictions are very fundamental; each makes an important contribution to the character of AMBIT/L as an efficient programming language.

R1. The undefined node. A node of type U (undefined) must not appear in a rule.

There is only one node of type U, namely U/$UNDEF, the undefined node. This node was built into AMBIT/L to provide a "don't know and don't care" destination for a link. More precisely, the undefined node is used by the interpreter as a private device to record the fact that the program does not have control over the destination of a particular link.

According to the restriction just given, the undefined node cannot be named literally in a rule. Furthermore, it is the case that any programmed attempt to "read" a link whose destination is the undefined node is a run-time error. Therefore, the undefined node cannot be referenced in any way in a rule and it is, in this sense, a "private device" of the interpreter.

The obvious use of the undefined node is as the initial value for all links; but, as part of the important activity of storage management, the interpreter will, at predictable times during program execution, "lose" the value of a link by setting it to point to U/$UNDEF. This treatment of data makes possible the optimization of AMBIT/L programs for a production environment.

R2. Cell names. The use of a cell with a specific subname other than the null cell is not permitted in a rule.

The purpose of a cell is to "split a link"; that is, one can think of a single link which points to a cell as becoming the two links which point from a cell. If we deny the program access to the specific name of a cell (as this restriction does), then the interpreter can assume responsibility for getting additional cells when they are no longer in use. This mechanism will be discussed in the section on functions.

R3. Accessibility. Every node in a rule must be accessible. A node is accessible if (1) it has a specific name, or (2) it is a destination of an ordinary or function link whose origins are accessible.

The accessibility of the nodes of a rule and the functionality of the data graph are, more than anything else, the characteristic features of AMBIT/L. Furthermore, their analog appears in the other AMBIT languages, AMBIT/S and AMBIT/G, and their presence unites this family of pattern-matching languages.

The match phase (binding of nodes) of a rule can, according to the accessibility restriction, be construed as a series of parallel walks. One side of a parallel walk is in the rule; it begins with a node which has a specific name and proceeds, link by link, to other nodes in the rule. The other side of the parallel walk is in the data graph; it begins with the same node as that which began the rule walk, and it proceeds in parallel, link for link, with the rule walk. As the parallel walk proceeds, the nodes of the rule are bound. This "AMBIT scan" is discussed in detail in an earlier paper [10].

The method of pattern matching just described can be implemented very efficiently; in fact, AMBIT/L rules are compiled into code which bears no resemblance to that for a general pattern-matching facility. Each step of a walk has a relatively small maximum cost which is independent of the state of the data graph. It follows that the cost of interpreting a rule does depend on the cost of function calls, but does not depend directly on the state of the data graph.

A related consequence of the accessibility restriction, which can be proven by consideration of parallel walks, is as follows: For a given rule and a given data graph, there is at most one way in which the nodes of the rule can be bound to the data graph.

Each of the AMBIT languages seeks to appear to be a simple pattern-matching language. This appearance is desirable because pattern-matching is a useful and natural method of programming. Under this simple exterior, however, lie restrictions which eliminate the usual scans and searches of pattern-matching. These restrictions permit the compilation of programs into efficient and practical machine code.

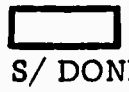
THE PROGRAM

The framework of AMBIT/L is a relatively conventional part of the language. Its use of block structure, function definition and transfer of control are based on ALGOL 60, although we have generalized or simplified them in a few places; furthermore, the design of the internal stack used in our implementation is almost identical to that used for ALGOL 60 [11]. Accordingly, our discussion of the program framework will be brief.

An AMBIT/L program is thought of as a character sequence. In support of this view, a rule is viewed as a single character; an oversized and complicated character, but a character nonetheless.

As in ALGOL 60, a program is a block. A block is a sequence of declarations, function definitions, rules and other blocks. Flow of control is established by attached labels and transfer lists as indicated by the following example:

LAB1:



S/ DONE F/LOOP;

This is a single rule, abbreviated to a small rectangle, as it might appear in a program. It is labelled LAB1; and after each execution of the rule the interpreter will proceed to the rule labelled DONE or LOOP, according as the rule succeeds or fails. There are, of course, various conventions to make the use of labels and transfer lists more concise and convenient.

The programmer writes subroutines in the form of function definitions and invokes these subroutines by means of call nodes in his rules. We have already described call nodes, but we should reconsider their role. The purpose of a call node is to map its arguments onto its results during the pattern-match phase of rule interpretation. The call node can be interpreted (we now assert) only after its origin nodes (arguments) in the rule have been bound; and the effect of its interpretation is either to bind all its destination nodes (results) in the rule (if the function succeeds) or to cause the rule to fail (if the function fails).

A function definition declares a function-name and associates with it a list of zero or more argument pointers, zero or more result pointers, and a block. Because of the ALGOL-like use of blocks, it is easy to program recursive function definitions.

The block associated with a function-name is executed when a corresponding call node is encountered. Upon entry to the block, the argument pointers have been set to point to the nodes specified by the origins of the call node. During execution of the block, the argument and result pointers are used like ordinary pointers and, at some time, the result pointers are given values. Exit from the block is accomplished by transfer to one of the special labels RET or -RET according as the function call is to succeed or fail. If the success exit is used, the values of the result pointers are used to bind the destinations of the call node.

The execution of a complete user program by the system proceeds in three steps.

1. The built-in macros are applied to the user program. These macros define the way in which certain useful but redundant notations shall be expressed in more basic notation.

2. A copy of the environmental block is obtained and the user program is embedded at a specified place. The environmental block contains the definitions of all the built-in functions as well as declarations for system pointers, and so on. It is written as a normal AMBIT/L block except for the definitions of a few built-in functions which are primitive and must be defined in English.

3. The user program, augmented by the environmental block, is executed.

The next two sections of this paper describe some of the built-in functions and built-in macros of AMBIT/L.

BUILT-IN FUNCTIONS

The library of built-in functions is discussed here; these are the functions which a programmer can use (provided he understands them) without defining them. A complete and formal definition of these functions could be given by simply listing the full environmental block. However, it is more appropriate to select for informal discussion a few topics of particular interest.

Arithmetic

The present implementation accomodates a finite set of real numbers. The real numbers are represented in the 36-bit floating-point format of the PDP-10 which allows $2^{35} + 1$ distinct (normalized) values. Each of these values can be expressed, in an equivalent decimal form, as a real node; and these nodes are called the active reals. The built-in macro set replaces any non-active real in a program by the nearest active real; and functions which produce reals enforce the same approximation.

In sharp contrast, the present implementation accomodates virtually any integer. An integer i outside the range $0 \leq i < 2^{15}$ is represented as a list of n 36-bit binary words. This list is interpreted as an n-digit number with base 2^{35}. Since n can range from one to about 20,000, as required, integers of very large magnitude can be accommodated.

The usual set of arithmetic functions is provided. In the case of real nodes these are simply the means of calling the floating-point hardware instructions; but the integer nodes require an elaborate implementation. For small

integers, the functions are performed directly and rapidly by machine instructions; but in other cases, the functions are rather complicated programs written in AMBIT/L which constitute, in themselves, some of the more interesting applications of the language.

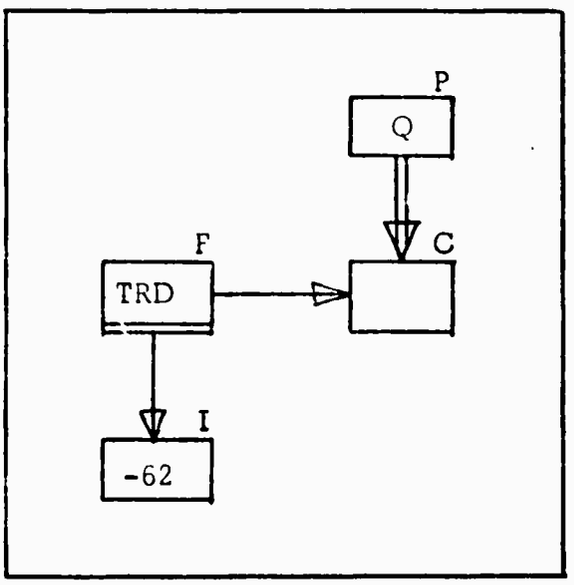
This discussion of arithmetic functions is a good place to comment on the many "trap functions" used by the AMBIT/L functions. Consider, as an example, the pointer P/ZERO.DIV which is declared in the environmental block. The division routine assumes that this pointer has as its value a function-name. When an attempt to divide by zero is detected by the division routine, this function is called with the dividend as argument and the imposed quotient as result. The rule E10, given earlier, illustrates this.

The important point is that this function is called indirectly through P/ZERO.DIV. This pointer is initialized in the environmental block to a default function for division by zero; but the user can, in a completely dynamic manner, change the pointer to point at some other appropriate function. The effect can be a call on a function which is dynamically but not statically accessible. The importance of this for systems programming is very great. It permits, for example, the kind of operations performed by the PL/I ON-statement facility.

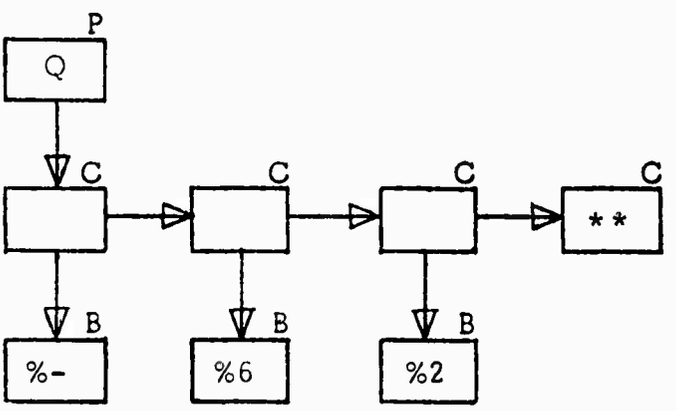
Conversion

In a representation of a data graph, each subname is represented as a sequence of one or more characters. In the case of integer, real, string, and token nodes, these subnames can be assembled from or disassembled into their constituent parts by the program. This is done by conversion functions.

The conversion function TRD takes as argument an integer or real node and produces as result an appropriate list of basic symbols. As an example of the use of TRD, consider the rule



This rule causes the creation of a list of basic symbols which appears in the data graph as follows:



The functions TRI (transfer to integer) and TRR (transfer to real) produce an appropriate numeric node from a well-formed display. TRR will "round-off", where necessary, to produce an active real; but otherwise TRI and TRR provide the inverse to TRD for numeric nodes. The conversion functions for integers and reals are used primarily in passing between the external form of numbers (list of characters) and the internal form (real or integer node); that is, they are used in conjunction with input-output.

The function TRD can also be applied to a string node to produce a display of the subname of the node; and the function TRS (transfer to string) provides the inverse of this operation. These functions provide the entire basis for symbol table management when an interpreter or a compiler is written in AMBIT/L. For each variable (for example) in the source text, a string node is created and its down link is set to a structure which accumulates information about the declaration and use of the variables.

Finally, the function TRD can also be applied to a token to produce a list of the elements in the subname; and TRT (transfer to token) is the inverse. These functions are a generalization of the string-processing functions just mentioned. They can be used to implement array storage, as suggested by the token in the example data graph. Storage of both strings and tokens is implemented by hash-coding techniques.

Cell Management

A program cannot mention cells by name; instead it must either arrive at a cell via an existing link or use the built-in function GET.CELL to get a new cell. This function has no arguments and one result. The result is a cell whose links are undefined (point to U/$UNDEF) and which is free. A cell is free if it is not the destination of any link. When program execution begins, all cells are free; as program execution proceeds, they are "used up" by the use of GET.CELL to build structures.

In a given implementation the number of cells available is finite and, usually, constant. When GET.CELL is called and finds no free cell, a garbage collection occurs. During this process, all cells (except C/**) which are not accessible by a sequence of links beginning at a pointer, string or token are redered into free cells. If not a single cell is found, the situation is grave and the trap function selected by the system pointer GCOL.CHOKE is called. Otherwise, GET.CELL takes up where it left off.

Garbage collection is a costly process. It is required only to the extent that the programmer does not know when he is allowing a cell to become

free. Usually the situation is mixed. When it is certain that a cell is becoming free, the function FREE.CELL is used in the program; otherwise, the cell is left to garbage collection. The correct use of FREE.CELL can produce important savings; an erroneous use can produce terribly obscure bugs.

Input-Output

Functions for input-output provide for communication with disk files (ASCII or binary) and a remote terminal (such as a teletype). The functions create, open, close, delete, or rename disk files; they make use of "logical names" for the files; and they can control the cursor within a file. Except for a few primitives which serve as the interface with the PDP-10/50 time-sharing monitor, the functions are written in AMBIT/L and are part of the environmental block.

An important pair of built-in functions are OUTSTRUCT and INSTRUCT, which are used to transmit an arbitrary (possibly cyclic) structure between the data graph and a binary disk file. These routines can be used to program the overflow of data onto the disk and the later retrieval of this data, a process we call data paging.

BUILT-IN MACROS

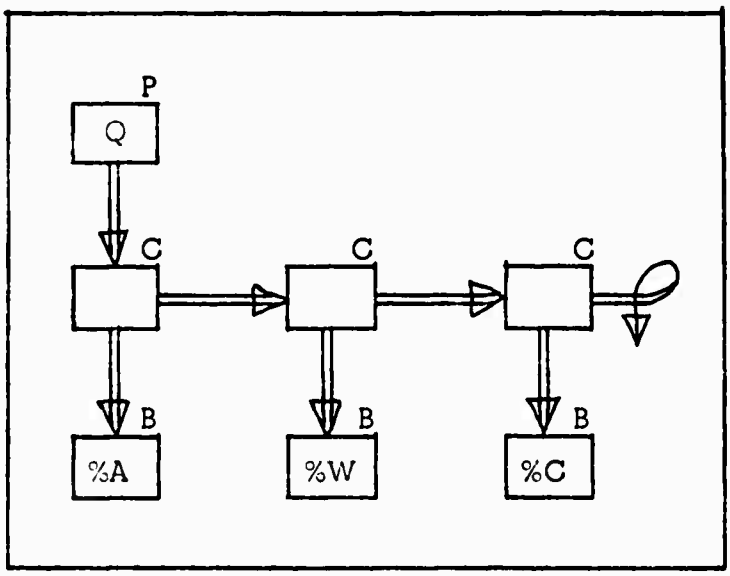
Let us designate the language we have thus far described as basic AMBIT/L. In addition to the notations allowed in the basic language, there are about a dozen special notations. A special notation is defined by (1) extending the syntax of AMBIT/L to include a construct which was formerly illegal and then (2) defining a macro which explains how to eliminate any use of the new construct by expressing it as a less convenient but otherwise equivalent construct in the basic language.

The Twisted Link Notation

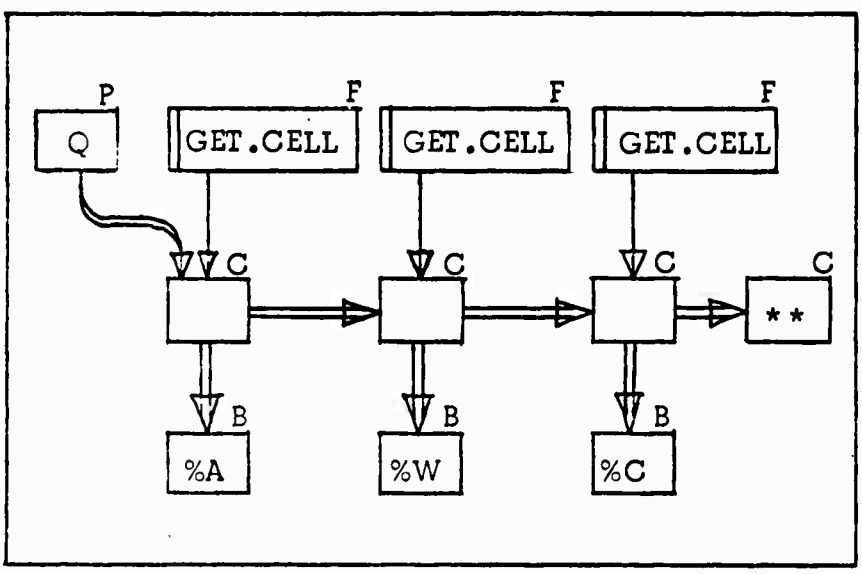
We begin by describing a very simple special notation called the twisted link. The definition of the rule is extended to permit any link except a flow link to twist back and cross itself once and then have its destination in empty space. The twisted link macro asserts that such a link is equivalent to a link which has the null cell, C/**, as its destination. An example appears in the next rule diagram.

The New Cell Notation

A program builds up a structure in the data graph by using the GET.CELL function to obtain individual free cells and then linking these cells into a structure. Consider the rule

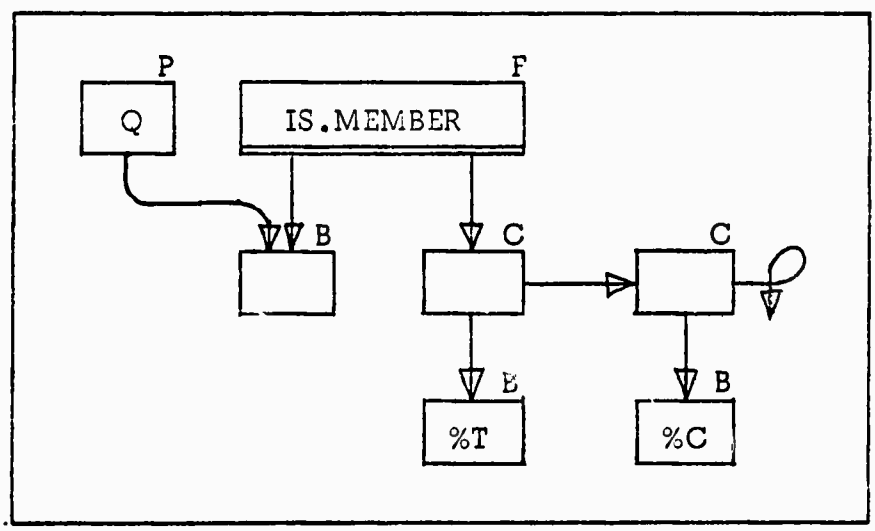


It is clear that something is missing from this rule since the three blank cells violate the accessibility restriction on rules. These cells are, in fact, uses of the special notation called new cell. The new cell macro asserts that what is missing here are some calls on the function GET.CELL. That is, the new cell macro (and the twisted link macro) changes the rule to read as follows:

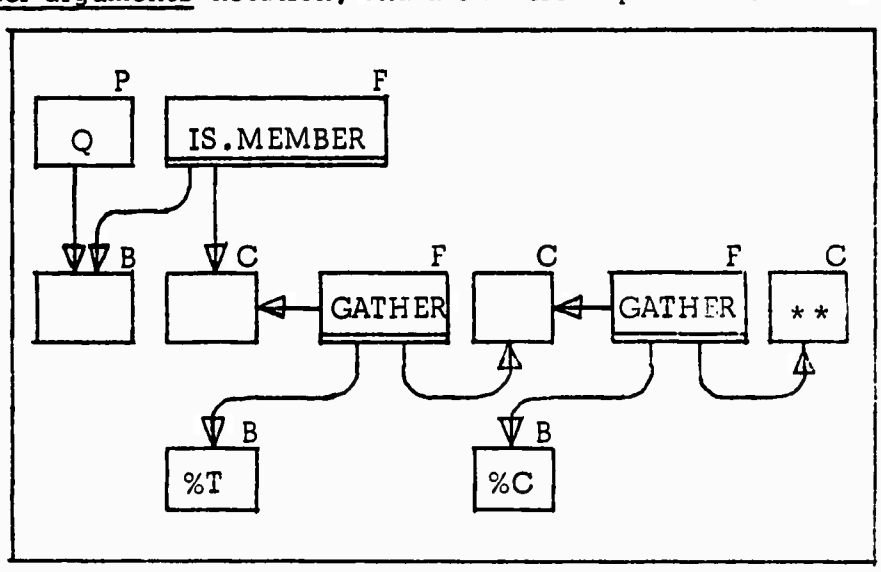


The Gather Arguments Notation

A function often has an argument which is, in turn, a list of cells. The list can be prepared in some rule before the rule which uses it as an argument; but it is useful to be able to "gather" the items of the list in the same rule as the function call. Consider the rule



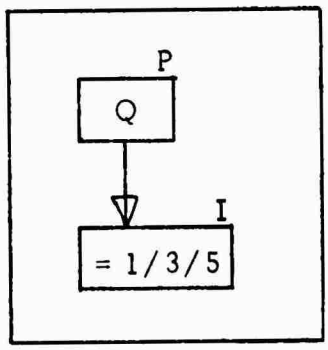
Again it is clear that something is missing from this rule since the blank nodes of type C again violate the accessibility restriction. But the case is more complicated here since the cells are connected by ordinary links. The cells use the gather arguments notation, and the macro expands the rule above to be



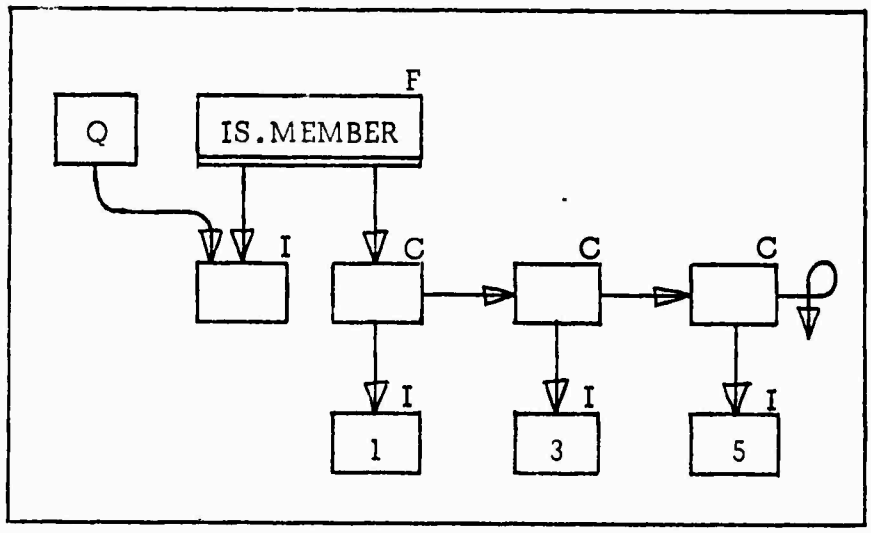
The function GATHER is a built-in function which uses GET.CELL to get a free cell, sets the links of that cell to the two arguments, and then returns that cell as the result. It is a tricky way of creating a data structure during the pattern match phase of rule interpretation.

The Name Set Notation

It is useful to be able to express concisely the assertion that the subname of a node is a member of a certain set of subnames. Consider the rule



This rule quite clearly asserts that Q points to the integer 1, 3, or 5. It is a use of the name set notation, and the macro expands it into the rule



User Defined Notations

A variety of the special notations which are built into AMBIT/L have now been illustrated. A formal way of writing the macros for these notations is being developed and has been used to define some of the special notations. The formalism is diagrammatic and expresses, as a general schema, the before-and-after diagrams which we gave for three examples above.

At present, the formalism for macro definition is being used as an aid to language design and definition and is not implemented in the system. When and if such a formalism is implemented, however, it will allow a user to define his own special notation and AMBIT/L will become an extensible language.

THE SYSTEM

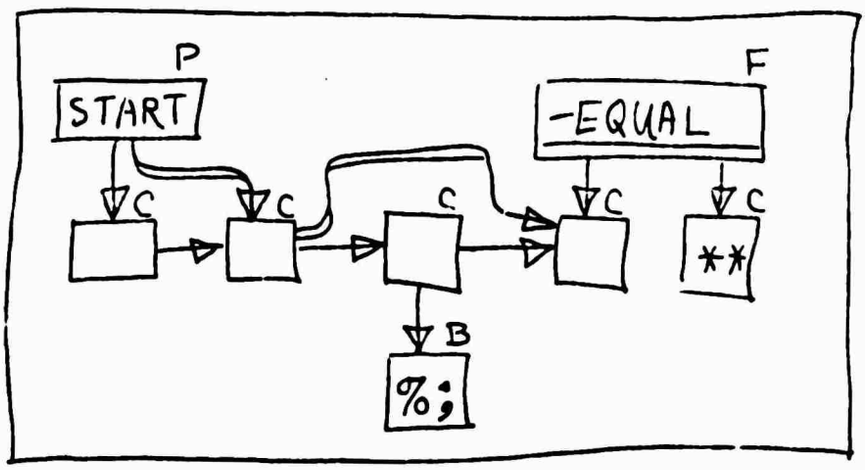
AMBIT/L is implemented under the standard monitor for the PDP-10/50 time-sharing system; but AMBIT/L is a complete subsystem and quite thoroughly insulates the user from the monitor. It tries to satisfy all the user's needs and, in doing so, performs actions the monitor never suspects, such as drawing diagrams at the teletype and running a private paging system.

The Diagram Generator

The use of AMBIT/L requires the communication of diagrams between the user and the system. This could involve an elaborate graphics console supported by complex software. Instead, we use a teletype and a processor called the diagram generator.

The following procedure has been used successfully for the input of large volumes of AMBIT/L programs:

Sketch the Program. The programmer begins by making a rough sketch of his program. The following rule illustrates the style of this sketch:



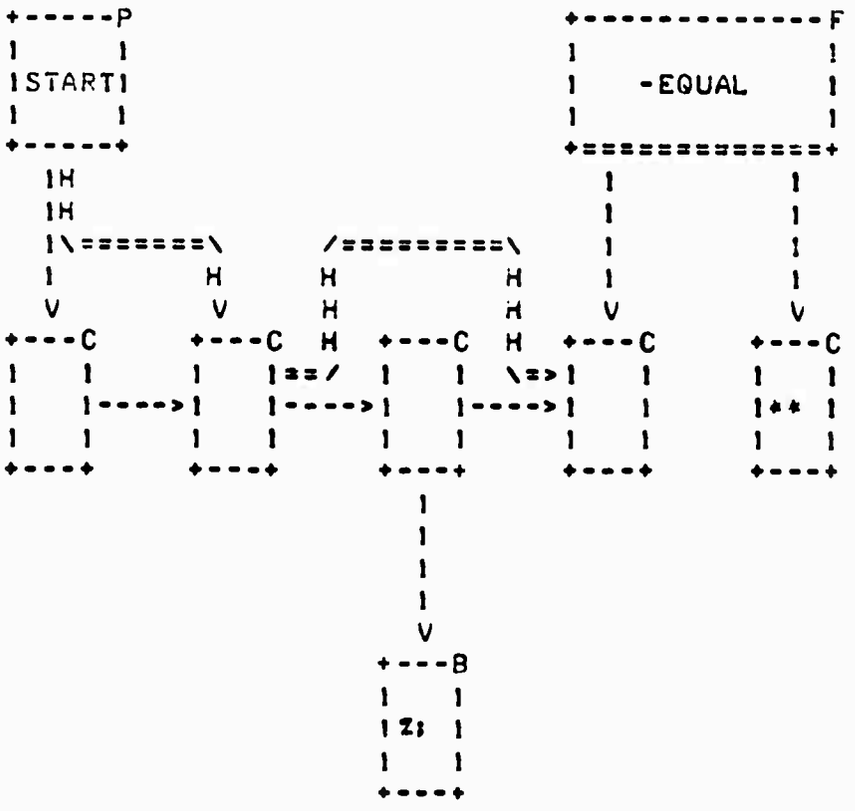
We will imagine that this rule is a fragment of a real program and follow it through subsequent steps of the input process.

Input the encodement. Next, the program sketch is given to a programming aide who uses a teletype to input the encoded source of the program. The process of translating from sketched diagrams into encoded source is simple and is done on the fly. The rule above is entered as the following text:

```
0?139    RULE
0?140    A1/P/START D/B1 BD/B2,
0?150    A4-A5/=F/-EQUAL D/B4 A5/D/B5,
0?160    B1/C R/B2,
0?170    B2/C R/B3 BMRURRDMR/B4,
0?180    B3/C R/B4 D/C3,
0?190    B4/C,
0?200    B5/C/**,
0?210    C3/B/%;,
```

The encodement considers the nodes to be arranged in rows (A,B,C,...) and columns (1,2,3,...) and the programmer must format his sketch accordingly. If the encodement of a link gives only the initial direction (up, right, down, or left) and destination of the link, then the diagram generator will provide a default routing for the link. With remarkable frequency, this routing is adequate. For the exceptional case (one appears in this diagram), the route of a link is explicitly encoded. The entire encodement language is a curious object, since it was designed for one-way communication. Except for editing to correct errors, it is written (typed-in) but not read by the user.

Output the listing. Next, the programming aide uses the diagram generator. This is a FORTRAN program which takes an encoded source file as input and produces diagrammed source as output. The output is a text file, but it "spells out" the following diagram.



It would be very useful to have another diagram generator which would draw diagrams of publication quality by using a graph plotter, and we have done some work toward this objective.

Use the listing. Finally, the programmer receives the diagrammed source listing, checks it, and discards the sketch. Thereafter, the programmer uses only the diagrammed source and the system uses only the encoded source. This pattern is broken only when the program must be changed; at that time a cycle similar to that which created the program is initiated on the portion to be changed.

The Compiler and Link Editor

A program has been defined as a single character sequence, and presumably would reside, as encoded source, in a single disk file. For practical purposes, however, a program can be broken up along the boundaries of its block structure.

Suppose, for example, that we wish to separate from the program a certain block, b, contained in the program. We can make a copy of b, file the copy under a name fn, and replace the entire block b in the program with the command "INSERT fn". This process of breaking up a program into insert files can be performed on any or all of the blocks in a program, however deeply nested those blocks may be.

This process has no effect on the logic of the program; however, it can have drastic effects on the economics of preparing and running the program:

1. Each insert block is compiled separately and independently of the compilation of every other insert block;
2. Each insert block is link-edited depending only on the previous link-editing of all containing blocks; and
3. Each insert block becomes a separate program segment of the reentrant binary code which is swapped in from disk, according to need, automatically, by the interpreter.

Thus the system works in units of blocks for compiling, linking, and actual run-time program storage.

The Run-Time Package

The object language produced by the compiler and bound by the link editor is not machine language; it is an interpreted binary language designed especially for AMBIT/L. Accordingly, an essential part of the run-time package is the interpreter.

In addition to the interpreter, the run-time core memory contains the following items: those built-in functions which are written in machine language; an area for the program segments which are swapped in and active; a base address table for the program segments; a stack very much like that used in an ALGOL 60 implementation and used to store pointer nodes; a set of words organized around a free list and used for cell nodes and certain internal lists; a hash vector for conversion of string and token names; and input-output buffers.

The interpreter takes full responsibility for the management of program storage. The programmer experiences no practical limit on the size of his program, and the time-sharing system does not know that something like software paging is going on. In contrast, the management of data storage is not automatic. Functions for the saving and retrieving structures are provided, together with traps for detection of storage overflow; but the program must explicitly specify where and when data is to be moved to disk.

The Debugging System

AMBIT/L is supplied with a simple but effective debugging system called DAMBIT/L. It is written in AMBIT/L (and was debugged mostly by itself) and is part of the environmental block.

At present, DAMBIT/L is not symbolic; that is, in order to refer to an identifier in a program the user must look up the binding of the identifier in the listing produced by the link editor. Because DAMBIT/L does not maintain a symbol table, and because the program segments of which it is composed will remain on disk except when actually used, the cost of having DAMBIT/L present, even during production runs, is very slight.

The debugging system accepts commands of the following general description:

1. Print out all or part of the structure which is the value of a specified pointer, string, or token node;

2. Transfer to a specified rule;

3. Plant specified debugging commands at a specified set of places in a program, where a place is a rule entry, function entry, or function exit;

4. Delete commands from specified places at which they have been planted;

5. Interrupt execution and ask for more commands from the teletype; or

6. Print a specified message.

The last two commands appear to be useful only when planted; but any command can be used directly or as a plant.

ACKNOWLEDGEMENTS

The design and definition of AMBIT/L was supported by the Advanced Research Projects Agency as part of the project "Research in Machine-Independent Software Programming" under ARPA Order Number 1228.

The AMBIT/L system was implemented on the PDP-10 as part of the LAM project by James Botto, Maynie Ho, Charles Jones, Michael Karr, Edward Krugman, and Michael S. Wolfberg under the direction of Charles Muntz and the author.

The author wishes to especially thank T. E. Cheatham, Jr. of Harvard University and Richard C. Jones, formerly president of Applied Data Research, Inc. for their encouragement and support during the development of AMBIT/L.

REFERENCES

1. McCarthy, J. et al. LISP 1.5 Programmer's Manual. MIT, Cambridge, Mass., 1962.

2. Farber, D. J., Griswold, R. E., and Polonsky, I. P. "SNOBOL, a string manipulation language." JACM 11 (1964), pp. 21-30.

3. Naur, P., ed. "Revised report on the algorithmic language ALGOL 60". Comm. ACM 6 (1963).

4. PL/I Language Specifications. Systems Reference Library C28-6571-4, International Business Machines Corp., New York, 1965.

5. Christensen, Carlos. "An example of the manipulation of directed graphs in the AMBIT/G programming language." In Interactive Systems for Experimental Applied Mathematics, M. Klerer and J. Reinfelds, eds., Academic Press, New York, 1968.

6. Rovner, Paul D. and Henderson, D. Austin. "On the implementation of AMBIT/G: a graphical programming language." Proceedings of the AFIPS/ACM International Conference on Artificial Intelligence, Washington, D. C., May 1969.

7. Henderson, D. Austin. "A description and definition of simple AMBIT/G --a graphical programming language." Massachusetts Computer Associates, Inc. (CA-6904-2811), Wakefield, Mass., April 1969.

8. Christensen, Carlos and Karr, Michael. "IAM, a system for interactive algebraic manipulation." To be presented at the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March, 1970.

9. Christensen, Carlos. "Examples of symbol manipulation in the AMBIT programming language." Proceedings of the ACM 20th National Conference, Cleveland, Ohio, August 1965, pp. 247-261.

10. Christensen, Carlos. "On the implementation of AMBIT, a language for symbol manipulation." Comm. ACM 9 (1966), pp. 570-573.

11. Randell, B. and Russell, L. J. ALGOL 60 Implementation. Academic Press, New York, 1964.

12. Christensen, Carlos, Wolfberg, Michael S., Fischer, Michael, "A report on AMBIT/G." Massachusetts Computer Associates, Inc. (CA-7102-2611), Wakefield, Mass., February 1971.

Appendix A

THE SYNTAX OF NODE NAMES

In the section called "The Data" the names of nodes were defined in an informal way; in this appendix, a complete and formal definition is given.

Each name is a type and a subname. There are eleven distinct types, so that each type-subname pair has one of the following forms:

I/integer	P/pointer
R/real	S/string
B/basic-symbol	T/token
M/mark	C/cell
F/function-name	U/undefined-destination
L/label	

In each of these, the type is a single letter which is given literally; for example, 'I', 'R', etc. On the other hand, the subname is given as a syntactic variable which will be defined below; for example, integer, real, etc.

The subnames are defined by means of syntactic formulae. In this notation, the lower-case letters and the hyphen are used to spell the names of syntactic variables; the right arrow, curly brackets, and vertical bar have special meanings; and the remaining characters are those actually used in subnames. The reader who is not familiar with the notation used in these formulae will find assistance after the formulae are given.

The following formulae define the eleven kinds of subnames:

F1. real → {sign} $_0^1$ unsigned-real

F2. integer → {sign} $_0^1$ unsigned-integer

F3. basic-symbol → % symbol

F4. mark

F5. function-name → qualified-identifier

F6. label

F7. pointer

F8. string → ' {quoted symbol} $_0^\infty$ '

F9. token → ({element} $_1^\infty$)

F10. cell → $ unsigned-integer | null-cell

F11. undefined-destination → $UNDEF

The preceding formulae introduce new syntactic variables which, in turn, are defined as follows:

F12. unsigned-real → unsigned-decimal {scale-factor} $_0^1$

F13. unsigned-decimal → {digit} $_1^\infty$. | . {digit} $_1^\infty$ | {digit} $_1^\infty$. {digit} $_1^\infty$

F14. scale-factor → E {sign} $_0^1$ unsigned-integer

F15. unsigned-integer → {digit} $_1^\infty$

F16. qualified-identifier → identifier $ unsigned-integer

F17. identifier → letter $\{\text{alphnum}\}_0^\infty \{ . \{\text{alphnum}\}_1^\infty \}_0^\infty$

F18. element →

real	integer
basic-symbol	mark
function-name	label
pointer	string
token	null-cell

F19. null-cell → **

F20. symbol → protected-symbol | free-symbol

F21. quoted-symbol → % protected-symbol | free-symbol

F22. protected-symbol → % | ' | other-protected-symbol

F23. free-symbol → alphnum | sign | . | (|) | * | $ | other-free-symbol

F24. alphnum → letter | digit

F25. letter → A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z

F26. digit → Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

F27. sign → + | -

F28. other-protected-symbol

F29. other-free-symbol

→ [implementation dependent]

The notation used for the formulae above can be adequately defined by a few examples.

= Formula 3 can be read as "a basic-symbol is defined as a character sequence composed of an instance of the character '%' followed by a symbol"; or, more briefly, "a basic-symbol is a '%' followed by a symbol".

= Formula 9 is read "a token is a '(' followed by a sequence of from one to infinity (i.e., one or more) elements followed by a ')'".

= Formula 10 reads "a cell is (1) a '$' followed by an unsigned-integer or (2) a null cell".

The definitions just given for real and integer subnames provide more than one representation for a given numeric value; for example, the three subnames 13, +13, +013 are all allowed for the value thirteen. The following rules define canonical numbers, a set of reals and integers which have one representation for each value:

1. A canonical number must be the shortest possible representation of its value. (The length is the number of characters in the representation.)

2. If a canonical number is a real, then the unsigned decimal in the real must have the form

$$\text{digit} \; . \; \{\text{digit}\}_1^{\infty}$$

When these rules conflict (as with 23. and 2.3E1) the second rule takes precedence (and 2.3E1 wins).

The integer and real nodes of the data graph must have subnames which are canonical numbers. The rules of a program are not so restricted because a built-in number macro replaces every number by its canonical form as part of the macro expansion of a program.

Appendix B

THE SYNTAX AND INTERPRETATION OF PROGRAMS

This appendix defines the syntax and interpretation of the complete program. The syntax of the program is given partly by invoking the definition of the rule already given (see the section called "The Rule") and partly by syntactic formulae which are given here. On the other hand, the interpretation of the program is given entirely in this appendix. The heuristic interpretation given in the section on the rule is not used. Instead, the interpretation of the rule is given in a step-by-step algorithmic manner which indicates, in a general way, the underlying implementation of the language. This new interpretation of the rule is given as an integral part of a complete interpretation for programs.

Syntactic Formulae. The syntax of the program is given in the same notation used in Appendix A.

F1. program → block

F2. block →

BEGIN o
$\{$ declarative o $\}_{o}^{\infty}$
$\{$ function-name o function-definition o $\}_{0}^{\infty}$
$\{$ attached-label o | imperative o $\}_{0}^{\infty}$
END

F3. declarative → declarator o { identifier o $\}_1^\infty$

F4. declarator → TEMP | PERM | MARK

F5. function-name → identifier

F6. function-definition → function-heading o block

F7. function-heading →

(o { argument o $\}_0^\infty$) o { result o $\}_0^\infty$:

F8. argument
F9. result
} → identifier

F10. attached-label → identifier o :

F11. imperative → rule-or-block o //o transfer-list

F12. rule-or-block → rule | block

F13. transfer-list → S/ o exit-label o F/ o exit-label

F14. exit-label → identifier | - o identifier | @ o identifier

The 'o' used in these formulae is a context-dependent object. It is read as "a separator which may be omitted unless both its left and right neighbors are alphnumpers". For this reading we need the following definitions:

F15. separator → {blank | new-line | comment$\}_1^\infty$

F16. comment → $ {non-new-line$\}_0^\infty$ new-line

F17. alphnumper → letter | digit | .

The syntactic variables blank and new-line have their obvious but uprintable meanings. Identifier, letter, and digit were defined in Appendix A.

The only remaining syntactic variable which is used in these formulae but not yet defined is rule. We adopt the definition of the structure of a rule given in the section on rules; it is a particular kind of diagram enclosed in a rectangular boundary. Furthermore, we view a rule as a single character; an oversized and complicated character, but a character nonetheless. Thus the rule becomes a part of the character sequence which is a program.

Syntactic Notes. The notes which follow are definitions and restrictions relating to the subject of declaration of identifiers. The substance of these notes cannot be expressed in syntactic formulae; they supplement the formulae and complete the definition of program structure.

When, in the following notes, we speak of an instance of some construct, we mean a specific character sequence in a particular place. For example, in the character sequence

ABC+2*(R3/ABC-421)

there are two (distinct) identifiers, 'ABC' and 'R3'; but there are three instances of identifiers, namely two instances of 'ABC' and one of 'R3'.

When we speak of an instance of some construct being contained in an instance of another construct, we mean containment in place. For example, there is an instance of a parenthesized expression in the example above. This contains one instance of 'ABC' (the rightmost) but not the other instance of 'ABC'.

Finally, when we speak of a constant being smaller than another, we mean the first has fewer characters than the second. The identifier 'R3' is smaller than the identifier 'ABC'.

N1. Declaration of identifiers. There are contexts in which an instance of an identifier is a declaration instance of that identifier. Such

an occurrence establishes a declared type for that identifier. Thus,

= In a declaration, an identifier is declared temporary pointer, permanent pointer, or mark according as the declarator is TEMP, PERM, or MARK;

= As the identifier which immediately preceeds a function definition, an identifier is declared function-name;

= In a function-heading the identifiers are declared temporary pointer; and

= In an attached label the identifier is declared label.

N2. Usage of Identifiers. Each instance of an identifier which is not a declaration instance is a usage instance of that identifier.

N3. Scopes. A scope is (1) a block which does not end a function definition or (2) a function definition. In other words, a scope is a block except where it expands to include a function heading which is just before the block.

N4. Immediate containment. Let c be an instance of some construct, such as an identifier, imperative or so on; and let s be any scope which contains c. Then s is said to immediately contain c if s is the smallest scope which contains c.

N5. Government. Let u be a usage instance of the identifier x. Consider all scopes which (1) contain u and (2) immediately contain a declaration instance of x. If there are any such scopes, choose the smallest. Then each declaration instance of x in the chosen scope is said to govern the usage instance u.

N6. Existence and uniqueness of declaration. For every usage instance of an identifier, there must be exactly one declaration instance which governs it.

N7. Agreement of use with declaration. The context in which a usage instance occurs must agree with its declaration. Specifically,

= An identifier which appears in a rule as the subname of a node with type P, F, L, or M must be governed by a pointer (either temporary or permanent), a function, a label, or a mark, respectively; and

= An identifier which appears in an exit label must be a pointer if it is preceded by '@' and must be a label otherwise.

N8. Locality. A declaration instance of an identifier and all the usage instances governed by that declaration instance are said to be local to the scope which immediately contains the given declaration instance.

The definition of locality is the main objective of these notes. The notion of locality partitions the set of all instances of identifiers in a program and establishes a one-to-one correspondence between scopes and the subsets of the partition.

We now come to the presentation of nine rules for the interpretation of AMBIT/L. These rules are concerned with the program, block, rule, ordinary-link, function-link, modification-link, transfer-list, function-name, and function-definition, in that order.

The Program. The interpretation of a program begins with certain editing operations on the program itself. The system macro expansions are performed, an environmental block is wrapped around the program to provide a library of functions, and those identifiers which represent permanent data are qualified once and for all.

I1. To interpret a program,

Set up working variables. Establish the following informal variables for use during program interpretation:

= the SQR (static qualifier register), used to distinguish scopes which exist before program interpretation;

= the DQR (dynamic qualifier register), used to distinguish scopes which are created during program interpretation;

= the PLR (parameter list register), used to pass lists of node names during interpretation of a function-link;

= the EVR (execution value register), used to record success, failure, or transfer as the consequence of a matching operation; and

= the TLR (transfer label register), used for the destination of a transfer.

Expand macros. The complete definition of AMBIT/L includes a set of built-in macros. These macros define the way in which constructs written in certain useful but redundant notations shall be expressed in more basic notation. Apply these macros to the user program.

Establish the environment. Obtain a copy of the environmental block, embed the user program at the proper place in this block, and call the result the augmented program.

The complete environmental block is given as part of the complete language definition; here we give only the following sketch:

```
BEGIN
   [declaration of built-in
   pointers and marks      ]
ADD ( A  B ) X :
     BEGIN
     [body of the addition routine]
     END;
SUB ( A  B ) X :
     BEGIN
     [body of the subtraction routine]
     END;
```

[definitions of all other
built-in functions ...]

[imperatives to initialize
the environment ...]

[the user program]

[imperatives to close
the environment ...]

END

Consider the contents of this augmented program. At the beginning, certain pointers and marks are declared; these are used primarily for communication between built-in functions and the user program. Next the built-in functions ADD, SUB, and so on, are defined. These are many and varied and will be discussed in a later section. Some of these, the primitive ones, have bodies written in English which, in this special case, the interpreter understands; others, the derived ones, are ordinary AMBIT/L function definitions. The block concludes with a sequence of imperatives, one of which is the given user program.

Qualify static identifiers. The permanent identifiers are those declared as permanent pointer or mark. These identifiers are now qualified once and for all as follows. Set the SQR (static qualifier) to zero and scan the augmented program. For each scope which is encountered,

= Increment the SQR by one;

= Form an unsigned-integer, qual, whose value is the value of the SQR; and

= For each instance of a static identifier x which is local to the currently selected scope, replace x by the qualified identifier x $ qual.

Note that this process does not qualify all identifiers; the remainder are qualified during program execution.

Execute the program. Interpret the qualified augmented program as a scope according to I2.

Scopes. A scope is the construct with which identifiers are associated. Temporary storage for the values of these identifiers is allocated upon entry to the scope and freed upon exit.

I2. To interpret a scope,

Qualify temporary identifiers. The temporary identifiers are those declared temporary pointer, label, or function-name. Proceed as follows:

= Increment the DQR (dynamic qualifier) by one;

= Form an unsigned integer, qual, whose value is the value of the DQR; and

= For each instance of a temporary identifier x which is local to the currently selected scope, replace x by x $ qual.

Execute the scope. Interpret the scope as a block (I3) or a function-definition (I10) as appropriate.

Clear temporaries. Set the down link of each temporary pointer which is local to this scope to U/$UNDEF.

Unqualify. For every instance of a qualified identifier x $ qual which was created by this interpretation of this scope, replace x $ qual by x.

Blocks. The block is the framework which binds the rules of a program together and provides for flow of control.

I3. To interpret a block,

Find the first imperative. Select the first imperative which is immediately contained in the block.

Execute block or rule. If the selected imperative begins with a block, interpret that block as a scope (I2); if it begins with a rule, interpret that rule (I4).

Interpret transfer-list. Interpret the transfer list (I8) with which the selected imperative ends. The TLR (transfer label) now contains a qualified identifier, lab. Try to find an attached label which contains lab and is immediately contained in the current block. One of the following will apply:

= The attached label is found and is followed by an imperative; in this case, select that imperative and go to execute block or rule, above. You have simply transferred from one rule to another in the same block.

= The attached label is found and is followed by END; in this case, interpretation of the block is now complete. You have exited the block by flowing out the bottom and interpretation of the block is completed in a normal way.

= The attached label is not found; in this case, set the EVR (execution value) to transfer. Interpretation of the block is now complete. You have exited the block, but you are in the midst of an incomplete transfer.

Rules. The definition of the rule given here shows clearly the importance of the accessibility restriction; the interpretation goes to completion if and only if that restriction is met.

I4. To interpret a rule,

Match the pattern. Initialize the EVR (execution value) to success and then do the pattern match. (Matching will require checking off links as they are processed and modifying node names in the rule as they are bound to specific names; such changes should be made in a temporary way and should be undone when an interpretation of a rule is complete.) Proceed as follows:

= Match a link. Consider every link in the rule which (1) is an ordinary or function link and (2) has specific names for all origins of the link and (3) is not checked. If there are such links, select one at random, interpret it according to I5 or I6, check it off, and go to test link match; otherwise, go to test pattern match.

= Test link match. If the EVR (execution value) is success, then go to match a link; otherwise, the interpretation of the rule is now complete.

= Test pattern match. If all ordinary or function links are checked and all nodes have specific names, then go to modify memory; otherwise, report error "rule violates accessibility restriction" and abort.

Modify memory. Interpret the modification links in the rule, once each according to I7, in random order.

Ordinary links. In a fully algorithmic interpretation of a rule, an ordinary link is a command to test and/or bind the destination of a link in a rule by reading a link in the data graph.

Both I5 and I6 require a definition of inclusion. A generalized name g includes a specific name s iff (1) g has a blank subname and a type-set which is either blank or includes the type of s or (2) g and s are equivalent.

I5. To interpret an ordinary link,

Identify its parts. Let x be the origin (it will be a specific name) of the link; let ln be the link name (down or right); and let y be the destination (it may be a generalized name).

Read the link. Determine a specific node namely $\bar{y}$ such that a link with origin x, link name ln, and destination $\bar{y}$ is in the data graph.

Test for undefined. If $\bar{y}$ is U/UNDEF, report error "reference to undefined link" and abort.

Bind destination. If $\underline{y}$ includes $\bar{y}$, then replace $\underline{y}$ by $\bar{y}$ as the destination of the link in the rule; otherwise, set the EVR (execution value) to failure.

Function links. A function link is a command to execute, in the very midst of the pattern match phase of a rule, a complete subroutine (called a function-definition).

I6. To interpret a function link,

Identify its parts. Let $x_1, x_2, \ldots, x_m$ be the names (they will be specific) of the origin nodes of the link; let fn be the function-name which appears in the call node of the link; and let $y_1, y_2, \ldots, y_n$ be the names (they may be generalized) of the zero or more destinations;

Submit arguments. Place the list $x_1, x_2, \ldots, x_m$ in the PLR (parameter list);

Call the function. Interpret the function name fn according to I9;

Test success. If the EVR (execution value) is not success, the interpretation of the function link is complete;

Accept Results. Get a list of node names, (they will be specific) $\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_{\bar{n}}$, from the PLR. If $\bar{n} \neq n$, report error "function returned wrong number of results" and abort;

Bind Destinations. If y_i contains $\bar{y}_i$ for all $i = 1, 2, \ldots, n$, then replace the y_i with the $\bar{y}_i$ (thus making the names of all destinations specific); otherwise, set the EVR (execution value) to failure.

Modification links. A modification link is a command to change the destination of a link in the data graph.

I7. To interpret a modification link,

Identify its parts. Let x be the origin (it will be a specific name), let ln be the link name (down or right) and let y be the destination (it will also be a specific name).

Write the link. Modify memory so that it contains a link with origin x, link name ln, and destination y.

Transfer-lists. The transfer-list dispatches control (the center of attention of the interpreter) from the completion of one rule to the initiation of another. This transfer of control can be indirectly specified by a label-valued pointer. The transfer can also set the execution value and thus produce the success or failure of a block as a whole.

I8. To interpret a transfer-list,

Test for transfer. If the value of the EVR (execution value) is transfer, the interpretation of the transfer-list is complete;

Select an exit-label. The transfer-list will have the form

S/sx F/fx

where sx and fx are exit labels. Assign sx or fx to the TLR (transfer label) according as the EVR is success or failure;

Evaluate indirect label. If the value of the TLR is of the form

@ qid

where qid is a qualified identifer, then,

= Let x be the node name which is the value of the pointer P/qid in the current data graph;

= If the type of x is L, then assign the subname of x to the TLR; otherwise, report error "indirect transfer to a non-label" and abort.

Reset execution value. If the value of the TLR is of the form

- qid

where qid is a qualified identifier, assign qid to the TLR (that is, drop the negation) and assign failure to the EVR; otherwise, assign success to the EVR.

Function names. A function-name designates a function-definition; and, in fact, when a function link is interpreted, the name of the link is replaced, in position, by a function-definition which specifies the meaning of the link.

I9. To interpret a function-name,

Get the definition. The function-name being interpreted, fn, is an optional '-' followed by a qualified identifier, qid; and it is the sub-name of some call node in a program. Try to find a function-definition, fd, in that program which is immediately preceded by qid. Then

- If you fail to find fd, report error "call on non-active function" and abort; and
- If you succeed, replace fn by fd and go to execute the function, below. (You have to squeeze a whole function definition into a single node boundary; write small or use some equivalent notational technique.)

Execute the function. One of the following applies to the function definition, fd, just obtained:

- It is written in AMBIT/L; in this case, interpret it as a scope (I2); or
- It is written in English (and is a primitive function in the environmental block); in this case, interpret it informally.

Negate the execution value. If the function name fn is a '-' followed by a qualified identifier, then invert the EVR (replace success with failure or failure with success).

Discard the definition. Replace the function definition fd by the function-name fn, thus restoring the call node to its original state.

Function-definitions. Upon entry to a function-definition, the argument pointers are set to point to the origins of the call. After the function body has been interpreted, the values of the result pointers are transmitted as the destinations of the calling function link.

110. To interpret a function-definition.

Identify its parts. The function-definition has the form

$$(a_1\ a_2\ \ldots\ a_m)\ r_1\ r_2\ \ldots\ r_n\ :\ b$$

where the a_i and r_i have just become qualified identifiers and b is a block.

Accept arguments. Get the list of node names, $x_1, x_2, \ldots, x_{\bar{m}}$ from the PLR (parameter list). If $\bar{m} \neq m$, then report error "function received the wrong number of arguments" and abort; otherwise, for $i = 1, 2, \ldots, m$, set each pointer P/a_i in the data graph to point down to the node named x_i.

Execute block. Interpret the block b appropriately (13).

Submit results. Assign to the PLR a list of node names, $y_1, y_2, \ldots, y_n$, such that for each $i = 1, 2, \ldots, n$ there is a pointer P/r_i in the data graph which points down to a node named y_i.

(END)