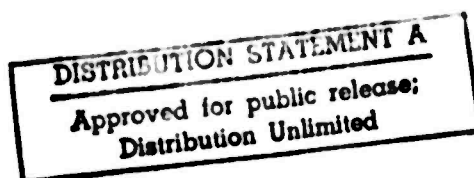


AD720313

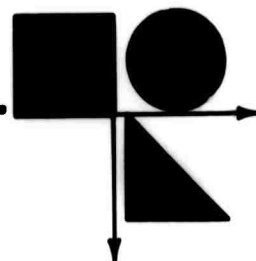


Massachusetts

COMPUTER ASSOCIATES

division of

APPLIED DATA RESEARCH, INC.

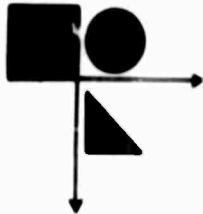


Reproduced by
**NATIONAL TECHNICAL
INFORMATION SERVICE**
Springfield, Va. 22151

DISCLAIMER NOTICE

THIS DOCUMENT IS THE BEST
QUALITY AVAILABLE.

COPY FURNISHED CONTAINED
A SIGNIFICANT NUMBER OF
PAGES WHICH DO NOT
REPRODUCE LEGIBLY.



APPLIED DATA RESEARCH, INC.

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

FINAL REPORT - TASK AREA I (Volume I)

(21 June 1968 - 31 December 1970)

FOR THE PROJECT
RESEARCH IN MACHINE-INDEPENDENT
SOFTWARE PROGRAMMING

Principal Investigators:

Task Area I	Carlos Christensen	(617) 245-9540
Task Area II	Anatol W. Holt	(617) 245-9540

Project Manager:

Robert E. Millstein (617) 245-9540

ARPA Order Number - ARPA 1228

Program Code Number - 8D30

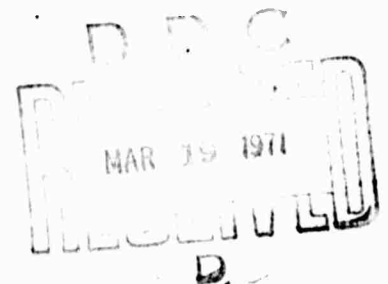
Contractor:	Massachusetts Computer Associates, Inc., Division of ADR
Contract No.:	DAHC04-68-C-0043
Effective Date:	21 June 1968
Expiration Date:	30 September 1971
Amount:	\$696,800.00

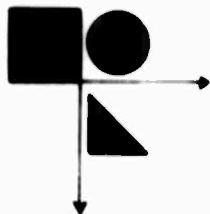
Sponsored by

Advanced Research Projects Agency

ARPA Order Number - 1228

CA-7102-2611





APPLIED DATA RESEARCH, INC.

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

A REPORT ON AMBIT/G ~~(Volume I)~~

by

Carlos Christensen
Michael S. Wolfberg
Michael J. Fischer *

CA-7102-2611
February 26, 1971

* Consultant to Applied Data Research, Inc.
Address: Department of Mathematics, M. I. T.,
Cambridge, Massachusetts

This is the first of four volumes of the final report on Task Area I of the project "Research in Machine-Independent Software Programming". This research was supported by the Advanced Research Projects Agency of The Department of Defense and was monitored by U. S. Army Research Office-Durham, Box CM, Duke Station, Durham, North Carolina 27706, under Contract DAHC04-68-G-0043.

ABSTRACT

AMBIT/G is an experimental language for software programming. It is oriented toward the manipulation of complicated data structures. Two-dimensional directed-graph diagrams are used to represent the data, and similar diagrams are used throughout the program as the "patterns" of rules to modify the data. An AMBIT/G system has been implemented on the Multics System at M.I.T. The implementation is ostensive and is intended for experiments in the use of AMBIT/G. It is written partly in AMBIT/G and partly in PL/I. This report begins with fundamental concepts and then proceeds to describe the implementation in great detail. The AMBIT/G programs for the AMBIT/G interpreter and the AMBIT/G loader are described and then displayed in full. Instructions for the input, execution, and debugging of a user program are given. Many examples are included, carefully chosen to illustrate and teach important features of AMBIT/G.

CONTENTS

→ Volume I

	Abstract	
1.	Summary	1
2.	Fundamentals data graph, constraints, program, general philosophy specific languages.	3
3.	Representation of Programs overview, program syntax, correspondence between program graphs and diagrams.	15
4.	The Interpreter overview, the compiler, interpretation of 'linkrep's, user-defined functions, error messages.	30
5.	The Loader overview, error messages, loader syntax, sample encodement, sample error.	48
6.	Initialization and the Built-in System hints, built-in nodes, built-in links, built-in function definitions, built-in rules, built-in data, built-in functions, sample error.	65
7.	The Debugging Facility lexical conventions, statements, statement forms, sample session.	99
8.	The Implementation credits and acknowledgements, internal view, files, FL/I data formats, PL/I implementation of the inter- preter and loader.	114
9.	Further Work	152
10.	Project Bibliography	157

Volume II

11.	Examples of AMBIT/G Programs observations, introductory examples: reversing a list, two forms of input, function calling, LISP gar- bage collector, another garbage collector, an inter- active program, sorting, factorial computation and recursion.	
-----	---	--

Volume III

12.	The AMBIT/G Interpreter as an AMBIT/G Program description, listing.	
-----	--	--

Volume IV

13.	The AMBIT/G Loader as an AMBIT/G Program description, listing.	
-----	---	--

CHAPTER 1

SUMMARY

This report is large. However, the casual reader can obtain a useful introduction to AMBIT/G by reading a few pages of the first two volumes. Specifically, we suggest that he begin with the next chapter of this volume, on fundamentals, and then read the first three sections of the second volume, concluding with the introductory examples, three programs for reversing the order of a list.

The report is large because it contains many complete AMBIT/G programs and because these programs require diagrams rather than text for their representation. The general trend of the report is from general and philosophical discussion to detailed and practical specifications. At the beginning we do not assume a prior knowledge of AMBIT/G, and at the end we give listings of large and complicated AMBIT/G programs. Much of this information is the unrefined output of our current research on AMBIT/G and therefore is not presented in a tutorial way.

After the chapter on fundamentals, the report proceeds to its main business, which is the definition and implementation of the AMBIT/G system. The definition and implementation are, in fact, partly identical since some of the implementation is written in AMBIT/G. Chapter 3 gives the representation of AMBIT/G programs in the form of AMBIT/G data and provides the basis for accepting a program as a data structure on which an interpreter program can operate. The chapter makes use of an interesting formalism for the specification of "grammars" for AMBIT/G data.

Once we have a way of thinking of a program as data, we can discuss an interpreter. Chapter 4 describes the AMBIT/G program (given in Volume III) which is our interpreter. An especially important part of this chapter is a discussion of the definition and use of functions in AMBIT/G.

Our implementation of AMBIT/G requires that both data and programs are presented to the system in an abstract input language (as textual descriptions of diagrams). Chapter 5 describes an AMBIT/G program (given in Volume IV) which "loads" pages of this input to produce an internal data graph. The chapter gives a formal syntax for the input language and includes an example of the use of the loader.

The AMBIT/G System is not empty when a user program arrives for interpretation. Certain information on the requirements of the program must be submitted in advance. More important, a variety of nodes, links, functions, rules, and pre-set data is built into the system in order to give the user a practical point of departure for programming. These facilities are described in Chapter 6.

A special subsystem for symbolic debugging is included in the AMBIT/G System, so that the user may inspect the data in a natural and interactive way. Chapter 7 describes the use of this subsystem in detail.

Although the interpreter and loader are both written in AMBIT/G, there is necessarily an underlying foundation for our Multics implementation. This foundation is composed of PL/I routines and is described in great detail in Chapter 8.

Chapter 9 contains some suggestions for further work on the implementation. The first volume concludes with an annotated bibliography of the papers and programs which have been produced by the project or are related to the project.

The second volume consists entirely of example programs. It is these examples which best display the concepts of AMBIT/G. In fact, nearly every example was chosen to illustrate a particular aspect of the AMBIT/G System.

We have already noted that the last two volumes are the complete program listings for the interpreter and the loader. These listings are included in the report for three reasons: The programs contribute to the formal definition of AMBIT/G, they form the basis for the implementation, and they are large-scale examples of AMBIT/G programming.

The scope of this report is limited to the definition and implementation of AMBIT/G and does not include other work done as part of the project. The interested reader is directed especially to our work on character recognition (Ledeen and Wolfberg), formal definition of BASEL (Jorrand and Hammer), description of simple AMBIT/G (Henderson), constraints (Third Semi-Annual Report), and the design of AMBIT/L (Christensen).

CHAPTER 2

FUNDAMENTALS

The project "Research in Machine-Independent Software Programming" is devoted to the capture and analysis of the techniques of software construction. By tradition and necessity, these techniques have been expressed fully only in machine-language programs; and in that form they are as obscure and exotic in our times as the operations of arithmetic were in the European Middle Ages. We seek a significant remedy to this situation by breaking away from current programming languages and following a fundamentally new approach to software programming. The practical results of this speculative venture are incorporated in an experimental programming system called AMBIT/G.

The AMBIT/G programming system is, first of all, a high level system for the construction of software. The term "high level" is often applied to a programming language to indicate the use of some combination of English and mathematical notation. We intend a more general use of the term. In our broader sense, a successful high level system provides a complete framework of concepts and techniques for programming in addition to a language; that is, it channels and supports the thoughts of the programmer as well as his utterances.

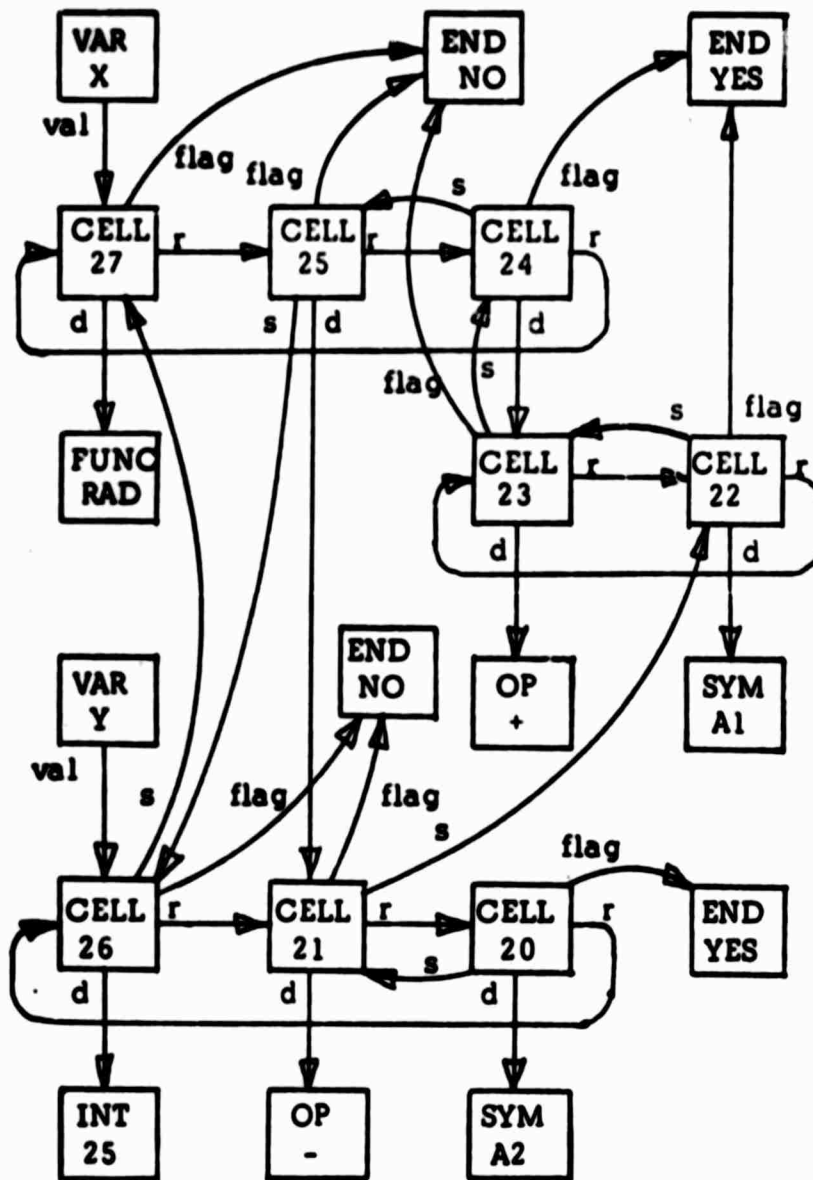
Our work on AMBIT/G has a simple underlying assumption. We believe that the characteristic activity of software construction is the design and use of complicated data structures, such as stacks, queues, rings, lists, and special tables. Indeed, the most important "construction" activity seems to be the structuring of data rather than programs. Accordingly, AMBIT/G is data-oriented to an unprecedented extent. At the beginning of a new programming task, the AMBIT/G user establishes a formal and "machinable" statement of the representation and properties of his data. Only when his data design is complete does he begin programming.

English and algebra, as used in COBOL, FORTRAN, and PL/I, for example, are an effective combination for commercial and scientific programming. However, these textual, essentially linear notations are not a natural medium for the description of structure in general or software data structures in particular. AMBIT/G rejects these notations in favor of another high level medium, the diagram.

The expository value of a diagram is well known. Flow charts of programs are very familiar and (more relevant to the present discourse) informal diagrams of data have been used for years to supplement program documentation. On the other hand, the formal adoption of a diagram as the "actual" data is quite unique to AMBIT/G and has a powerful effect; the diagram becomes an almost machine-like object, changing frequently in certain places and relatively fixed in others, a passive machine operated by a program but subject to its own built-in constraints.

THE DATA GRAPH

An early use of informal data diagrams was in the representation of LISP lists, and many variations have since been used in papers on software. We obtained a formal model for data by restricting and simplifying the notation rather than elaborating it. The final result is a precisely defined form of diagram called a data graph. The following diagram is an example of a (small) data graph.



The diagram is composed of nodes and links. A node is a rectangle with a node name written inside; this node name is a type written above a subname. There may be many nodes of a given type, and these are distinguished from one another by their subnames. In the diagram above, for example, there are eight nodes of type 'CELL'; their subnames are the integers from '20' through '27'. A link is a line which begins at an origin node, passes close to its link name, and ends (with an arrowhead) at a destination node. Every node of a given type has a similar set of links. For example, every 'CELL' in the diagram is the origin of four links which are named 'flag', 'r', 's', and 'd', and every 'SYM' is the origin of no links.

The types, subnames, and link names used in the data graph are selected by the programmer for each particular program. It is the facility for building special data structures, not the structures themselves, which is built into the AMBIT/G system.

Every data graph must be functional, that is, a given node (as origin of a link) and a given identifier (as link name) must specify no more than one node (as destination of the link). This allows the unambiguous specification of a "walk" along the links of a diagram by giving a starting node name and a sequence of link names. Purposeful link walking is an important activity of software programs.

The data graph must also be permanent: that is, nodes and links cannot be created or destroyed during program execution. In fact, the only permitted operation is the "swinging" of a link so that its pointed end moves from one node to another. This restriction reflects the fact that machines (including memory hardware) tend to be permanent.

Once the fundamental data representation has been established, certain superficial but useful "abbreviations" are introduced. For example, the type is dropped from within a node boundary and is indicated by giving the node boundary itself a distinctive shape. Or link names are dropped by establishing for each different link a characteristic point of origin on the node boundary. Such convenience notations make the diagram much more readable.

We do not intend that the programmer write out a large data graph; an architect does not draw every brick and nail of his building. However, the postulated existence of the data graph provides a reliable basis for the programmer's thinking. It is the basis for the design of constraints on data and the writing of programs.

CONSTRAINTS

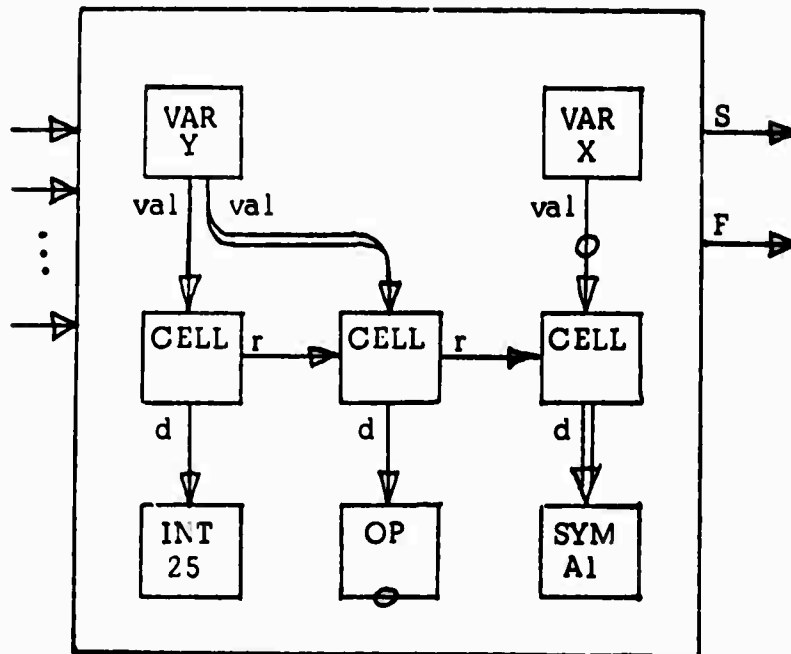
If a data graph has n nodes, then each link in the data graph has n states, one for each possible destination. Further, if the data graph has a total of k links, then the entire data graph has n^k different states.

The programmer uses formal statements called constraints to stipulate that certain states of the data graph can never occur during the execution of his program. A constraint may fix a given link to a single destination for all time; or it may restrict the link to destinations of a specified type; or it may establish a more general and dynamic dependency of the link on other links. When a program is being debugged, the program interpreter (a human reader or computer executor) can check for operations on the data which are inconsistent with the constraints and report these to the programmer.

Ultimately the data graph must be encoded in bits and stored in some computer memory. The amount of computer memory required will be a function of the number of states available to the data graph; therefore, constraint of the data graph reduces the memory required. Thus constraints are useful both for debugging the program and for optimization of storage. Constraints are a vital and growing aspect of AMBIT/G.

THE PROGRAM

An AMBIT/G program includes a collection of rules connected by flow lines as in a flow chart. Each rule is itself a diagram and uses a notation which closely resembles that of the data graph. An example of a single rule is as follows:



This rule is executed when "control" enters along one of the incoming flow lines at the left; and its execution results in control exiting to another rule along the success or fail flow lines to the right. The inside of the rule can be interpreted in three paragraphs, as follows:

First frame the data graph as follows: Select 'VAR Y', follow the 'val' link, and call its destination c1. Is c1 a 'CELL' node? Select c1, follow the 'd' link, and answer: is its destination 'INT 25'? Select c1, follow the 'r' link, and call its destination c2. Is c2 a 'CELL' node? Select c2, follow the 'd' link, and call its destination o1. Select c2, follow the 'r' link, and call its destination c3. Is c3 a 'CELL' node? (Should the answer to a frame question be "no", you have detected the consequences of a programming error; take the day off and get undefined.)

Next test the data graph as follows: Is o1 an 'OP' node? Select 'VAR X', follow the 'val' link, and answer: is its destination c3? (Should the answer to a test question be "no", take the fail exit from the rule.)

Finally, (if you haven't gone away) modify the data graph thus: Select 'VAR Y' and set its 'val' link to point to c2. Select c3 and set its 'd' link to 'SYM A1'. (No questions are asked during modification. When you are done, take the success exit from the rule.)

The paragraphs just given imply a total ordering of actions which we now revoke: The actions (commands and questions) within a given paragraph can be interpreted in any order provided that each variable (like cl) is associated with a node in the data graph (by a "call" clause) before it is referenced (by a "select" clause).

Every (single-line) link in any rule must be a part of an anchored walk. An anchored walk begins with a node whose full name (type and subname) is given in the rule and repeatedly "steps" from one node in the rule to another, each time following a link from origin to destination. This restriction means that the pattern-match can be implemented very efficiently; in fact, none of the "searching" characteristic of general pattern-matching is ever required.

To complete this discussion of programs, some remarks on program structure (that is, the framework in which rules are embedded) is necessary. Since most of the unusual and novel concepts of AMBIT/G seem to be confined to the rules, we seriously considered adopting the program structure of some existing high level language and we decided that ALGOL 60 was the obvious candidate. The use of an ALGOL 60 framework presented serious problems, however.

The first problem arose in finding an analog to the ALGOL 60 function reference. At first it appeared that there was no natural place for functions in a pattern-matching rule. Eventually, however, we developed the idea that the function reference and the data link are not in conflict but, rather, are two aspects of the fundamental mechanism for assigning structure to data. The function reference became a new and important part of the notation for rules.

Our second problem arose with block structure. The ALGOL 60 block structure is the basis for automatic storage allocation, and recursive function evaluation. It has been extremely successful and has become a classic component of high level languages. However, block structure implies hidden mechanisms for storage management which are in direct

conflict with the objectives of AMBIT/C, which seek to give the programmer close control over all his data. We rejected block structure because we could not find a simple and practical way to control its machinery.

Other problems of a less fundamental nature arose and we were forced, after all, to accept a minimal program structure, far simpler than that of ALGOL 60, which involved the use of success and fail flow lines to connect rules and a very general mechanism for function definition and reference.

GENERAL PHILOSOPHY

The designer of a programming language soon learns that the goals he has set for himself are in conflict. A language should be powerful yet easily implemented, rich in expression yet easily learned, application-oriented yet general purpose, concise yet readable, easily programmed in yet efficiently compiled. Most existing languages are readily classified along one or more of these dimensions and often are noteworthy because of an extreme position with respect to one of them. PL/I is noted for being powerful but difficult to implement; BASIC is at the other extreme. ALGOL 68 is rich in expression; SNOBOL 1 is easily learned. SIMSCRIPT is application-oriented; ALGOL 60 is general purpose. APL is concise; COBOL attempts to be readable. EULER is easily programmed; FORTRAN can be efficiently compiled.

The motivation in the design of AMBIT/G was not simply to decide upon a position with respect to each of the above parameters and then build yet another language, distinguished from the others only by the particular combination of choices made. Rather, it was to study some of these apparent conflicts in an attempt to see just how they influence language design, and based on the insights so gained, to build a language which overcomes the weaknesses and limitations which any compromise, no matter how carefully chosen, necessarily imposes.

This seemingly impossible undertaking has indeed succeeded, at least in its initial stages, and the particular solutions take one of two forms. Some of the conflicts among goals disappear with radical changes in perspective. Other conflicts, which we were unable to so eliminate, can be parameterized so that the user and not the language designer is able to choose the point of compromise.

Four main ideas emerged from these considerations:

- a) Data is of primary importance and should be designed first with the care usually given to the language imperatives.
- b) Two-dimensional representation permits humans to deal with greater complexity than is possible with linear representations.
- c) People seem to have an ability to comprehend spatial patterns of far greater complexity than temporal.
- d) Redundant information in the form of constraints can be highly useful both to people and to the machine.

How is it that these ideas have been overlooked for so long? To some extent, they are not new. LISP, PL/I, ALGOL 68 and BASEL certainly have the ability to deal with highly structured data. SNOBOL owes much of its success to the pattern-replacement idea. Certain explicit constraints such as declarations of array sizes and data type are present in several languages. But the exciting discovery is that the ideas are not independent and cannot realize their full potential in isolation.

Many languages, as we have noted, can deal with highly structured data. However, few languages make it convenient to manipulate data which is not basically tree-structured (or at least acyclic). In LISP, for example, one can create arbitrary structures through the use of the functions RPLACA and RPLACD, but it is an exercise reserved for the expert.

An important reason for the preference for hierarchical data is that it can be linearized in a fairly natural way using parentheses, precedence, and other devices. But these methods do not generalize nicely to cyclic structures, so a conceptual barrier arises between the two types of data. The net result seems to be that users are encouraged to force all their data into the often inappropriate hierarchical mold. In two dimensions, however, cyclic graphs are as easily represented as trees, and it becomes natural to break away from tree structures wherever appropriate.

Pattern-matching gives SNOBOL a gestalt capability and in many cases results in surprisingly perspicuous programs. However, string data tends to have only limited sorts of interesting patterns, so many SNOBOL programs use the pattern-matching facility mainly to emulate the structured data found in other languages. By generalizing the types of data that can be manipulated, many more interesting types of patterns become manifest and the gestalt methods of programming can handle a far larger portion of the computational task.

Designers of programming languages have often regarded declarations as nuisances which are eliminated wherever possible and which are useful only if a language is to be "compiled". It is true that such constraints enable more efficient implementations of a language, but they also serve two other distinct and equally important functions. First, they greatly facilitate debugging by establishing a set of conditions under which the program must operate; any attempted violation indicates an error. The subscript bounds checking of PL/I and the type-checking of ALGOL are such conditions. Second, declarations of constraint are a reliable form of comment and thus help contribute to documentation. The programmer who says on a comment card that his program never stores a number bigger than 100 into the variable X states this as a matter of belief; the programmer who includes that statement as an explicit constraint knows it to be true as a matter of fact. The significance of this distinction cannot be overestimated in a typical program which is modified many, many times before completion.

While constraints can be extremely valuable, it is difficult with most programming languages to envision very many different kinds other than the

ones alluded to above. However, once data with complicated and dynamically changing structure is introduced, there becomes a much more pressing need for constraints. The added generality provides more directions in which program optimization is possible and necessary. But most important, constraints, together with the two-dimensional representation, are the tools the user needs to control the greater complexity possible with the more general data.

SPECIFIC LANGUAGES

AMBIT/G has been implemented several times in the past four years, and these implementations are listed in the Project Bibliography included in this report. Most of the remainder of this report is devoted to the most recent of these implementations, an AMBIT/G system on Multics.

We have already stated that the AMBIT/G system is experimental and is a vehicle for expressing basic ideas about programming. On one hand, the system carries the use of diagrams to an extraordinary extreme, includes very carefully developed facilities for definition and use of functions, and endows the data with unprecedented independence. On the other hand, we have excluded features which we considered to be trivial (rational arithmetic), over-sophisticated (block structure), or peripheral (graphic input-output). The resulting programming system is a theoretical model, not a practical language for software programming.

The AMBIT/G language described in this report is the most important result of the project. It is the basis for future development of both theory and practice. However, a very different language, AMBIT/L, has come to light, rather unexpectedly, as part of the project.

AMBIT/L is the result of a vigorous specialization and simplification of AMBIT/G to produce a practical language for list-processing. It has an applications area quite similar to that of LISP, but it uses the diagrammatic pattern-replacement style of AMBIT/G. The language is described in a separate paper listed in the bibliography and submitted with this report. Under auspices other than this project, it was fully implemented and then successfully applied to the construction of a large software system for interactive algebraic manipulation.

Thus two specific languages have resulted from the project: AMBIT/G, an adaptable framework for testing principles of language design, and AMBIT/L, a practical embodiment of the current results of our work on diagrammatic programming.

CHAPTER 3

REPRESENTATION OF PROGRAMS

The diagrams with which the programmer represents his program are represented in the AMBIT/G system as ordinary AMBIT/G data and are accessible to him in the same manner as any other data. This allows one to write programs which construct other programs or which modify themselves. It also permits the interpreter itself to be expressed as an AMBIT/G program, which we have chosen to do in order to give a formal description of the semantics as well as being an aid in the production of an implementation. The interpreter and the implementation are described elsewhere in this report.

OVERVIEW OF THE PROGRAM REPRESENTATION

The description of the program representation is in two parts. First we define a class of data graphs which we call program graphs and which constitute the class of legal inputs to the interpreter. Second, we attempt to show how to find a diagram which the program graph represents. We note that there is not a one-to-one correspondence between program graphs and diagrams; a given diagram may be represented by many different program graphs and conversely, many different diagrams may have the same program graph for their representation (e.g. diagrams which differ only in the positions of the nodes on the page).

Two diagrams with the same set of possible representations are semantically equivalent. However, we will see that it is possible for a diagram to be represented by two or more distinct program graphs upon which the interpreter will behave differently and perhaps produce differing results. This may occur, for example, when a rule contains calls on functions which have side-effects. Diagrams which lead to two or more inconsistent interpretations, even though syntactically correct, are considered to be semantically undefined and not a part of the AMBIT/G language.

Briefly, a rule is represented by a collection of nodes of a small number of pre-defined types and certain of their links. The nodes in general represent the pieces of the diagram such as the boxes and the arrows, while the links represent the relationships among the pieces.

These same nodes carry other links which are used to record miscellaneous information generated during the process of execution of the program. Such information includes the results of compilation of a rule, the calls on user functions which are currently active, and the arguments and results being passed to user functions. Further mention of these links is deferred to the chapter on the interpreter.

PROGRAM SYNTAX

Shapes and Links in the Representation

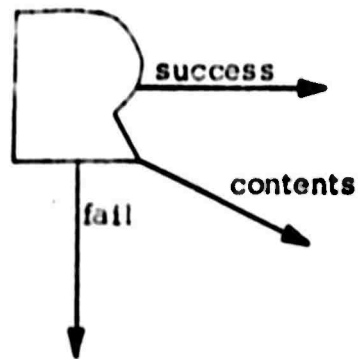
A program graph is a collection of nodes of the pre-defined types 'rule', 'linkrep', 'noderep', 'diamond', and 'flag', called program representation nodes, together with the links shown in the table below. The shapes used to picture these node types and the relevant links are shown following the table.

Links defined for a rule in state 'clear'

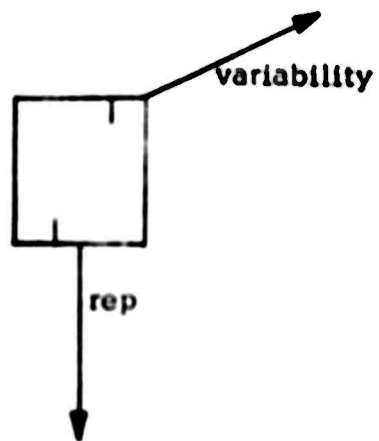
<u>Node Type</u>	<u>Link</u>	<u>Destination</u>	<u>Meaning</u>
rule	success	rule	success exit.
	fail	rule	fail exit.
	contents	linkrep	encoding of rule contents.
linkrep	mode	flag	mode of the link ('frame', 'test' or 'modify').
	org	diamond	list of tails.
	name	noderep	link name.
	dest	diamond	list of heads.
	next	linkrep	used to form list.
diamond	next	diamond	next element in heads or tails list.
	value	noderep	the list element itself.
noderep	variability	flag	tells whether or not the node is a dummy.
	rep*	user node	the node of the data graph represented by this rule node.

*Defined only if the 'variability' link points to 'flag fixed'.

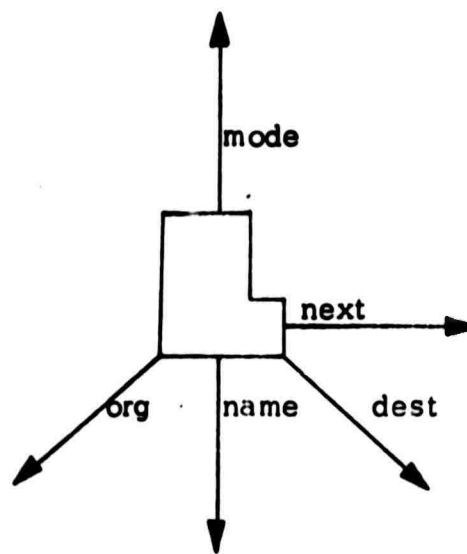
rule :



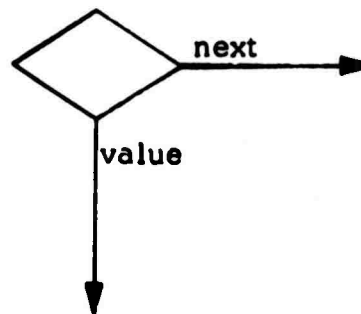
noderep :



linkrep :



diamond :

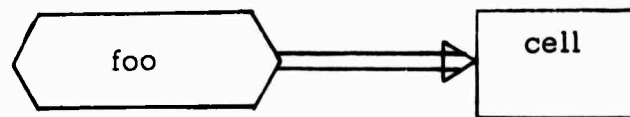


flag :

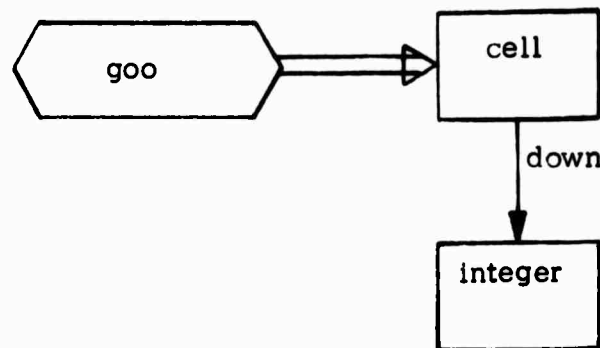


How to Read the Syntax Diagrams:

The syntax of AMBIT/G is specified by a "grammar" consisting of special diagrams. As in BNF, we introduce meta-linguistic variables which we call property symbols and represent by hexagons containing a character string. However, unlike BNF, we do not think of our grammar as generating a graph but rather as a means of testing a graph for certain well-formedness properties. Given a data graph, our grammar rules allow us to assign one or more properties to certain nodes of the graph. A hexagon from which emanates a double arrow is a defining instance of that property. A given node of the data graph is defined to have that property if the pattern beginning at the destination of the double arrow can be matched to a subgraph of the data beginning with the given node. For example, the syntax rule

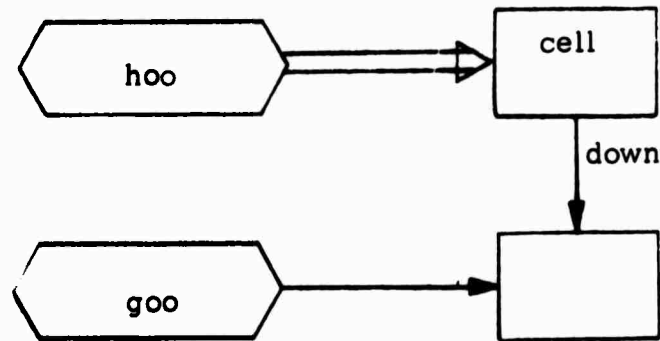


says that every node of type 'cell' has the property 'foo', whereas the rule

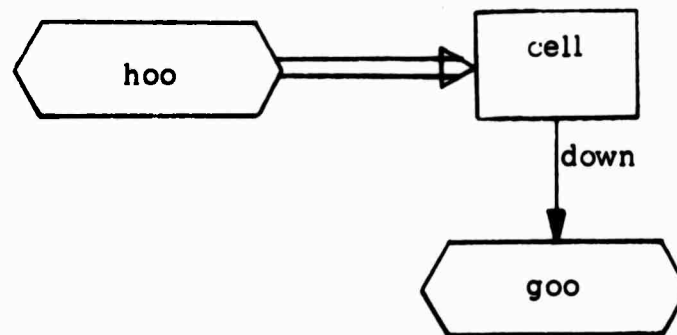


says that only 'cell's whose 'down' link points to a node of type 'integer' have property 'goo'.

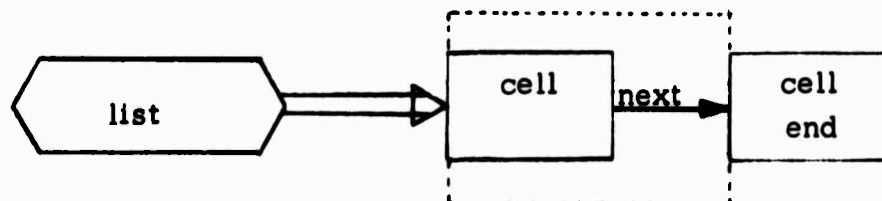
A property symbol may be used to qualify other nodes appearing in the diagram. For example, the rule



says that a 'cell' has the property 'hoo' providing that its 'down' link points to a node with property 'goo'. This may be abbreviated as

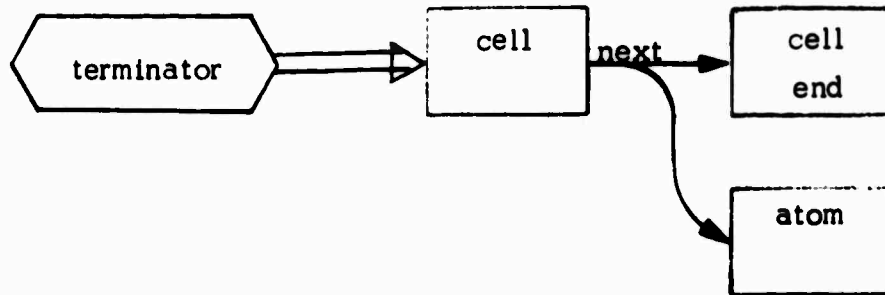


Two other notations may be employed in writing syntax rules, A section of the pattern may be enclosed in a dotted box to indicate zero or more repetitions of the enclosed pattern. For example,



specifies that a 'cell' is a 'list' if either it is the node 'cell end' or if 'cell end' can be reached from it via a chain of 'cell's along the 'next' link.

Finally, we allow an arrow of the pattern to branch, meaning an alternative is allowed. For example,

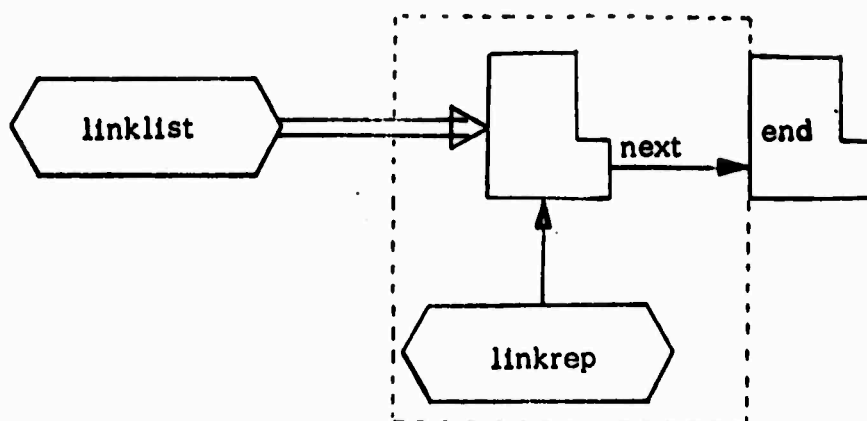
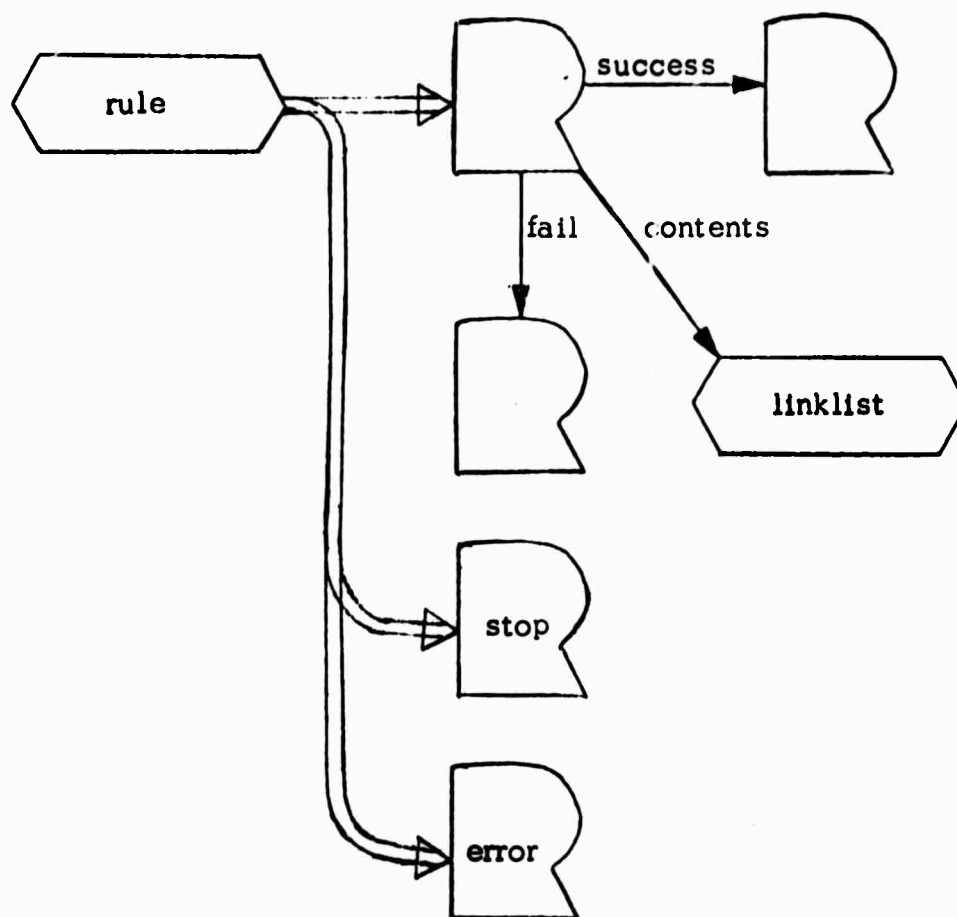


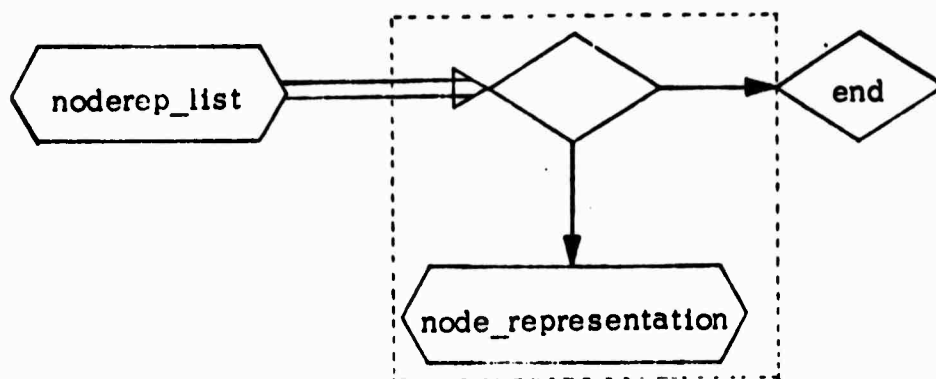
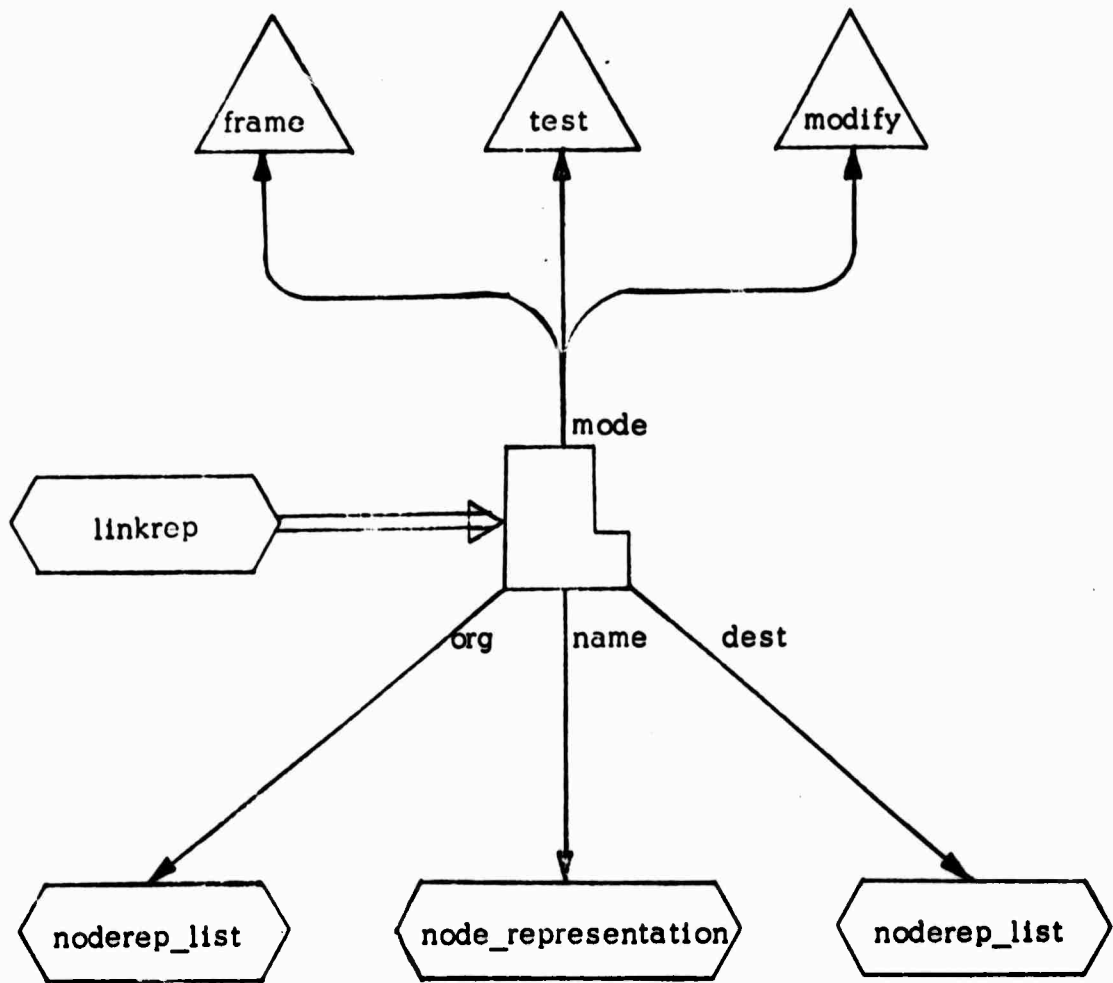
means that a 'cell' is a 'terminator' if its 'next' link points either to 'cell end' or to an 'atom'.

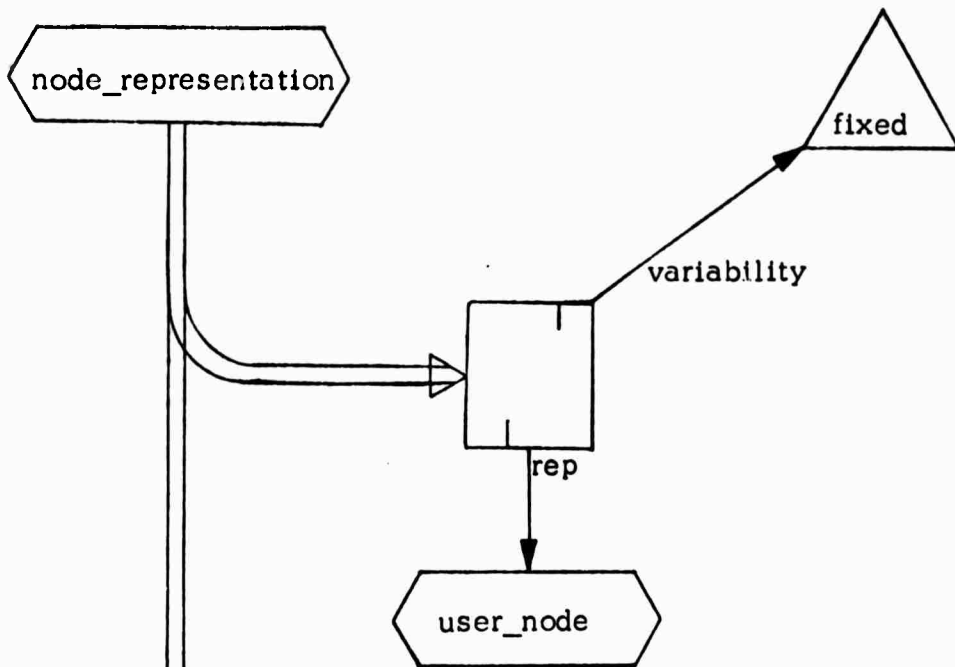
The Syntax of AMBIT/G

A collection of program representation nodes is by definition a program if some 'rule' node in the collection is assigned the property 'rule' by the following grammar. Any node with that property is a valid place at which to begin execution.

Grammar for AMBIT/G :







Additional Restrictions

In addition to the restrictions imposed by the grammar above, we constrain the "sharing" that may take place among nodes. Informally, we require that no node "belong" to the representation of more than one link.

More formally, we say that a 'diamond' D belongs to the 'org' ('dest') list of a 'linkrep' L if D is not the node 'diamond end' and also D is accessible from L by a path beginning with an 'org' ('dest') link from L and then continuing with zero or more 'next' links from 'diamonds'.

We say that a node N belongs to a 'rule' node R if either:

- a) N is not the node 'linkrep end' and N is accessible from R by a path beginning with the 'contents' link of R and continuing with zero or more 'next' links from 'linkrep' nodes; or
- b) N is a 'diamond' which belongs to the 'org' or 'dest' list of some 'linkrep' node which belongs to R; or
- c) N is a 'noderep' node which is the destination of the 'value' link of some 'diamond' which belongs to R; or
- d) N is a 'noderep' node which is the destination of the 'name' link of some 'linkrep' node which belongs to R.

We then require that each 'linkrep', 'diamond' and 'noderep' node belongs to at most one rule, and in addition, each 'diamond' belongs to the 'org' or 'dest' list (but not to both) of one 'linkrep' node.

CORRESPONDENCE BETWEEN PROGRAM GRAPHS AND DIAGRAMS

In this section, we show how, given a program graph, to find a diagram which that graph represents.

A 'rule' node together with the nodes which belong to it represent a single rule, diagrammed by a rule box. The surname of the 'rule' node is written in the upper-left corner of the rule box. The success and fail

exits of the rule lead to those rule boxes represented by the destinations of the 'success' and 'fail' links respectively.

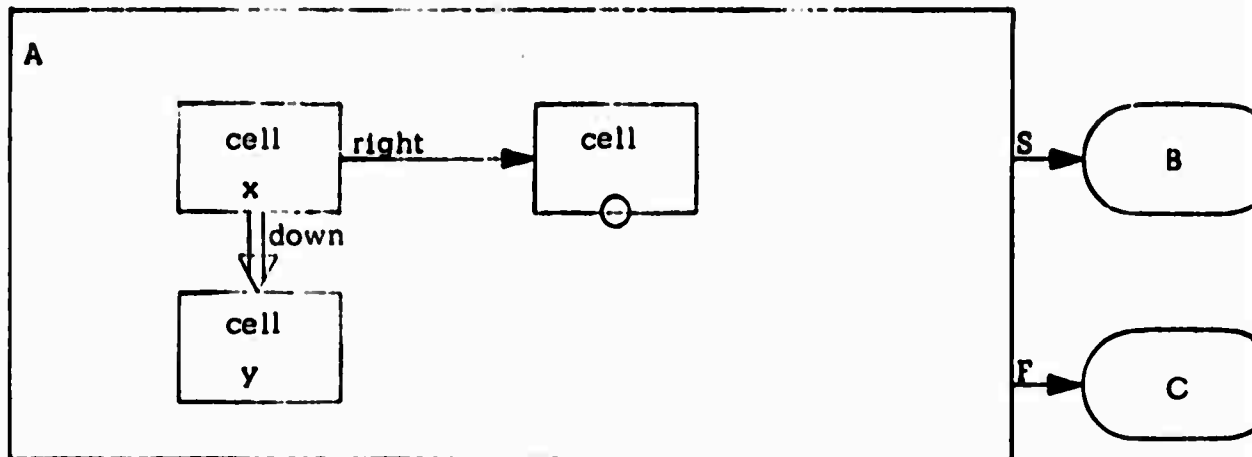
Each 'noderep' node belonging to the rule corresponds to a node box in the rule. A dummy 'noderep' (i.e., one whose 'variability' link points to 'flag dummy') is represented by a node box with no contents. A fixed 'noderep' whose 'rep' link points to a named node corresponds to a node box containing the full node name. At present, we have no way to diagram a fixed 'noderep' whose 'rep' link points to an unnamed node.

Each 'linkrep' node corresponds to an arrow of mode specified by the 'mode' link. The number of heads and tails of this arrow are determined by the lengths of the lists of diamonds hanging on the 'dest' and 'org' links respectively. The 'tails' and 'heads' of this arrow are attached to the node boxes corresponding to the 'noderep' nodes which are the destinations of the 'value' links of the 'diamond's in the 'org' and 'dest' lists respectively. The spur of this arrow is the node box corresponding to the 'noderep' node at the destination of the 'name' link.

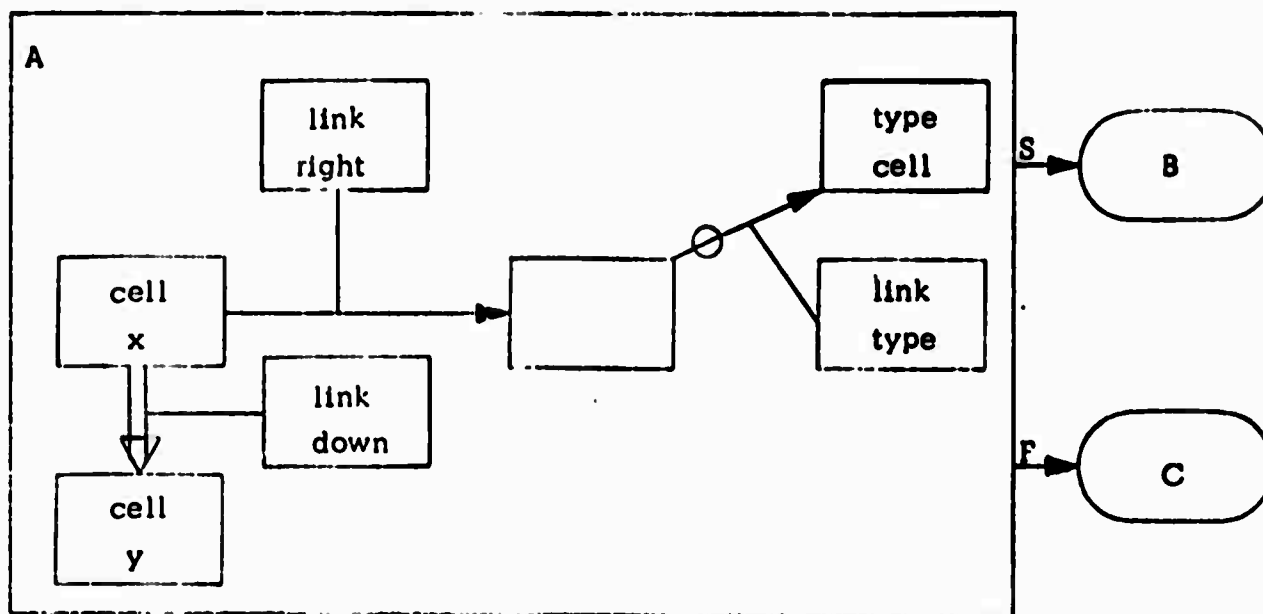
An example should help to make these ideas clear.

Example of Rule Representation

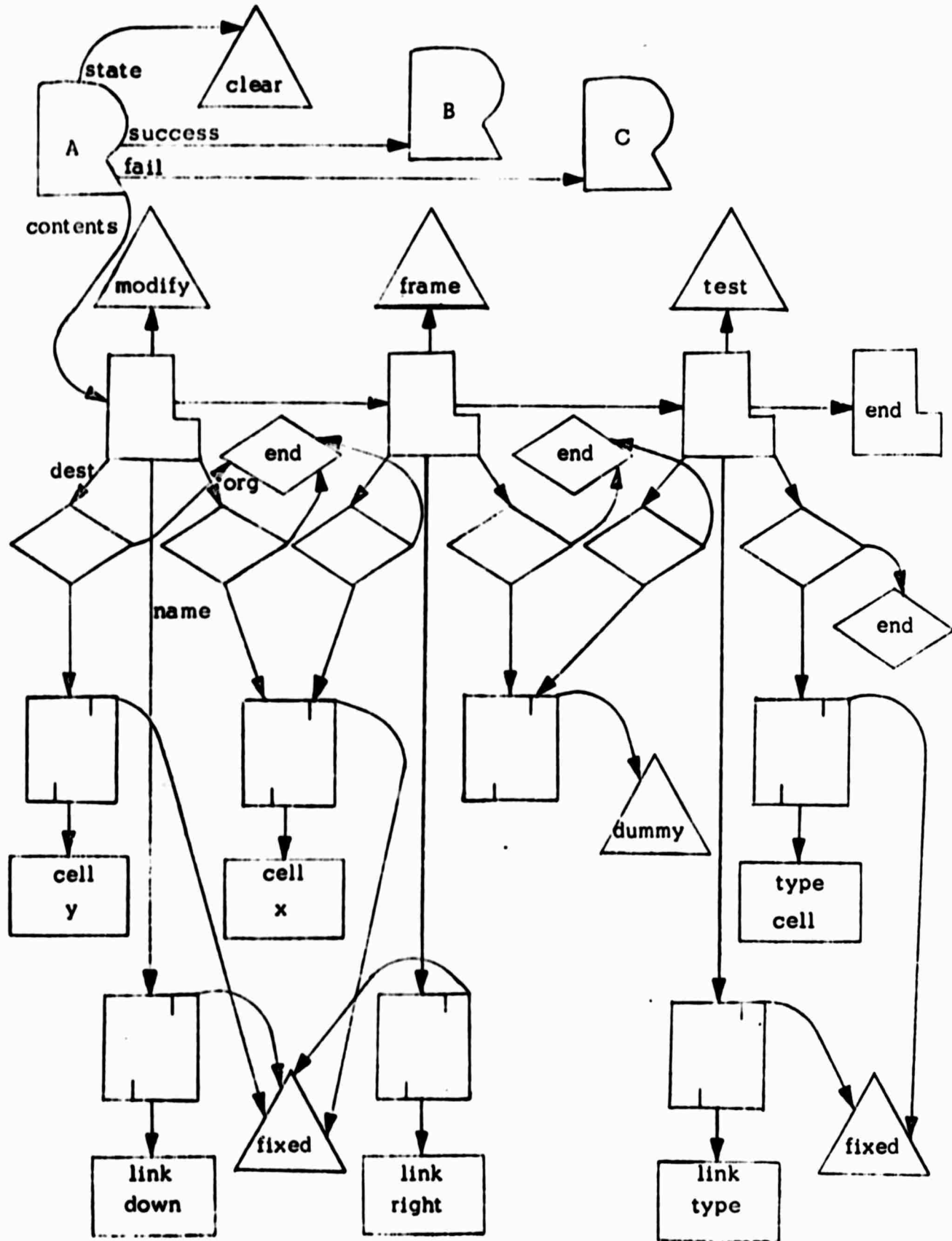
Sugared form of rule:



Desugared form of rule:



Data representation of rule:



CHAPTER 4

THE INTERPRETER

The AMBIT/G interpreter is an agent which, given an AMBIT/G data graph and the starting rule of a program represented within that graph, modifies the graph in successive steps according to one of the many possible interpretations of the AMBIT/G language.

Not all AMBIT/G programs will produce the same results on all implementations of the language; such programs we consider to be ill-formed. The decision to admit the possibility of certain syntactically correct programs whose semantics are unspecified is a compromise at best. It has the obvious disadvantage that it may be difficult or impossible to determine mechanically whether a given program is ill-formed, so that one may unwittingly use an illegal program to produce correct results at one installation and later have it fail at another.

On the other hand, to attempt to specify completely the effects of execution of all syntactically correct programs severely restricts the range of possible implementations at perhaps a considerable cost of efficiency. More seriously, one is forced to define and describe the results of "tricky" or pathological programs which should not be written anyway, greatly complicating the definition of the language.

Ours is not a new approach. FORTRAN for example does not specify the value of a DO-variable after normal exit from a loop. PL/I likewise has many "implementation-dependent" parameters.

OVERVIEW OF THE INTERPRETER

This section gives an informal description of the operation of the interpreter. It presupposes the reader is familiar with the representation of programs. It also uses the notation for describing paths through the graph that is defined in the chapter on the AMBIT/G symbolic debugger. While some attempt was made to be complete, this section should be regarded principally as an introduction to the formal definition of AMBIT/G.

The interpreter operates on a rule by rule basis, going through a cycle of several phases for each rule.

Most of the information recording the progress of execution of the program is stored in the several links originating from the 'rule' and 'noderep' nodes which represent the program; hence this information is available dynamically for inspection and modification.

Each 'rule' node has a 'state' link which tells the current status of execution of that rule. State 'clear' indicates a rule ready to be executed for the first time. Such a rule must first be compiled, after which its state is set to 'compiled'.

Execution then proceeds through the rule in three phases: 'frame', 'test', and 'modify', as indicated by the 'state' link. In each phase, 'linkrep's of the corresponding mode are examined one at a time in the order specified by the compiler and the appropriate action is taken. 'linkrep's of mode 'frame' cause dummy nodes to be matched (bound) to nodes of the data graph; those of mode 'test' cause the destinations of links to be tested, and those of mode 'modify' cause links in the data graph to be altered.

If any of the tests fail, the rule fails immediately -- the remainder of the 'test' phase and the entire 'modify' phase are then skipped, and the interpreter proceeds to the rule specified by the 'fail' link. If all the tests succeed then the 'modify' phase is performed as described, after which another interpretation cycle begins with the rule at the destination of the 'success' link. In either case, the state of the rule just executed is set back to 'compiled' to indicate that compilation need not be repeated on subsequent executions of that rule. Of course, if a user modifies a rule, he should reset the 'state' link to 'flag clear' which indicates that rule is in the 'clear' state.

Two rules have a special interpretation associated with them; 'rule error' causes an error message to be emitted and execution to terminate; 'rule stop' causes a normal return from the currently executing user subroutine, or if at the top level, a normal program stop. For both of these rules, the action is taken immediately and any contents of the rule are ignored.

THE COMPILER

The compiler is not invoked until just before a rule is to be executed, and on each call, it compiles only the single rule which is its argument.

Compilation consists of sorting the 'linkrep's by mode and ordering those of mode 'frame' so that later during interpretation every dummy node of the rule will have been bound to a node of the data graph before it is referenced. The compiler reports an error if such an ordering is not possible.

Compilation does not modify any of the links originally used to represent the rule. Rather, it adds information to the representation of the rule by setting additional links on the nodes of type 'rule', 'linkrep', and 'noderep', as shown in the following table:

Additional links defined for a rule in state 'compiled'			
<u>Node type</u>	<u>Link</u>	<u>Destination</u>	<u>Meaning</u>
rule	frame	linkrep	Head of a properly ordered list of the 'linkrep's of mode 'frame'.
	test	linkrep	Head of a list of the 'linkrep's of mode 'test'.
	modify	linkrep	Head of a list of the 'linkrep's of mode 'modify'.
linkrep	next1	linkrep	Used to chain together the elements of the 'frame', 'test', and 'modify' lists.
noderep	sets	diamond	If the node is a 'dummy', points to the 'diamond' in the 'frame' list which will bind the 'rep' link during execution. If the node is 'fixed', it points to 'diamond matched'.

The 'frame', 'test', and 'modify' links of the 'rule' node are set to point to the three new lists of 'linkrep's which the compiler creates using the 'nextl' link of the 'linkrep's. In addition, the compiler sets the 'sets' link of each 'noderep' to point to the 'diamond' of the 'linkrep' which is supposed to bind it, if any. (All the other 'frame' links which locate the node should just verify the prior setting.)

The compilation algorithm is fairly simple. First, all of the 'noderep' nodes belonging to the rule are marked as matched or unmatched according to whether they are fixed or dummies. The 'sets' link is temporarily used for this purpose. At the same time, the 'linkrep's are chained together in one big list, called the active list, temporarily using the 'nextl' link.

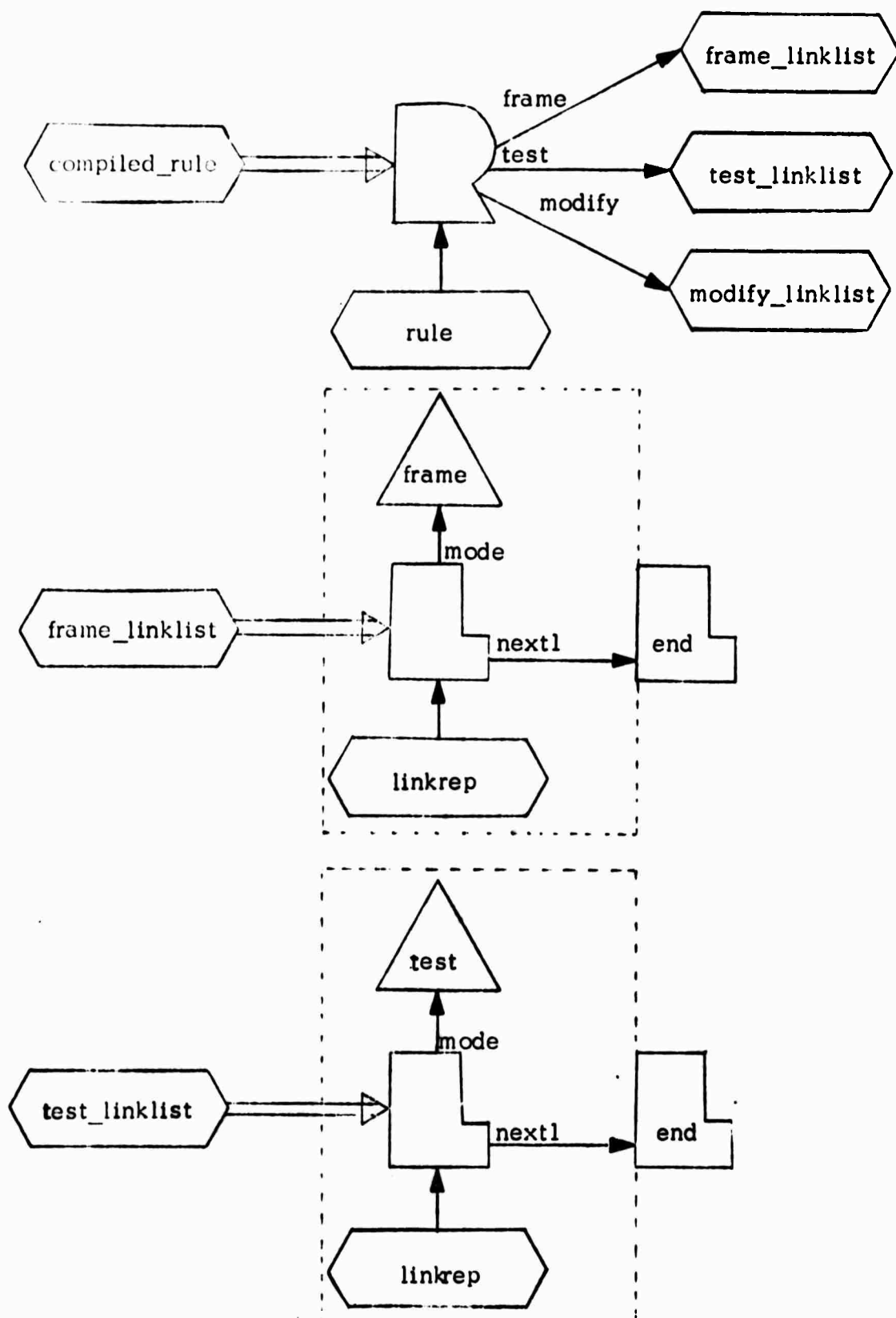
The active list is then scanned for an entry which is eligible for processing. An entry is eligible if the 'noderep's hanging from its 'org' and 'name' links have all been previously matched and, in the case of 'test' and 'modify' 'linkrep's, the destination 'noderep's have been matched as well. Whenever such an entry is found, it is removed from the active list and processed. The active list is then rescanned. When a complete pass over the active list fails to locate an eligible entry, the scanning phase terminates.

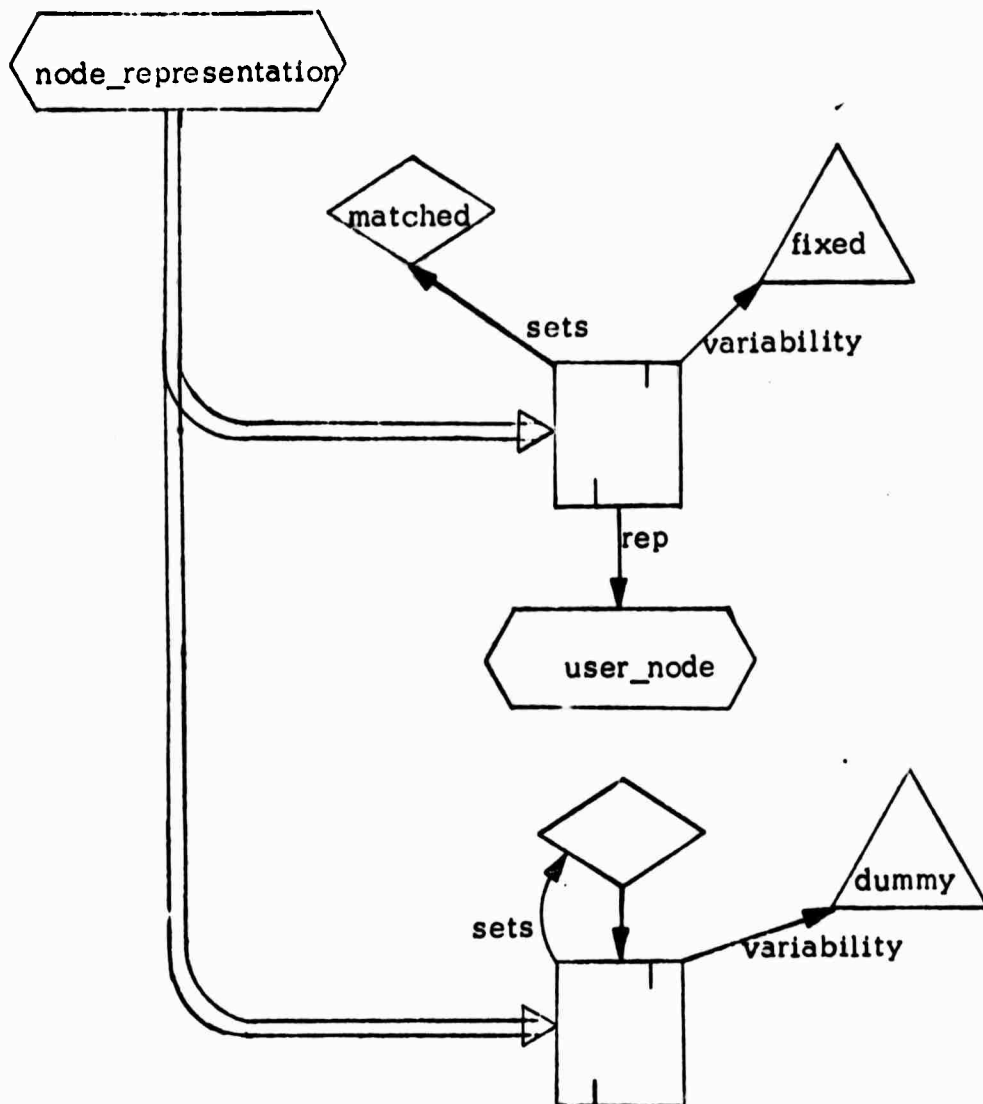
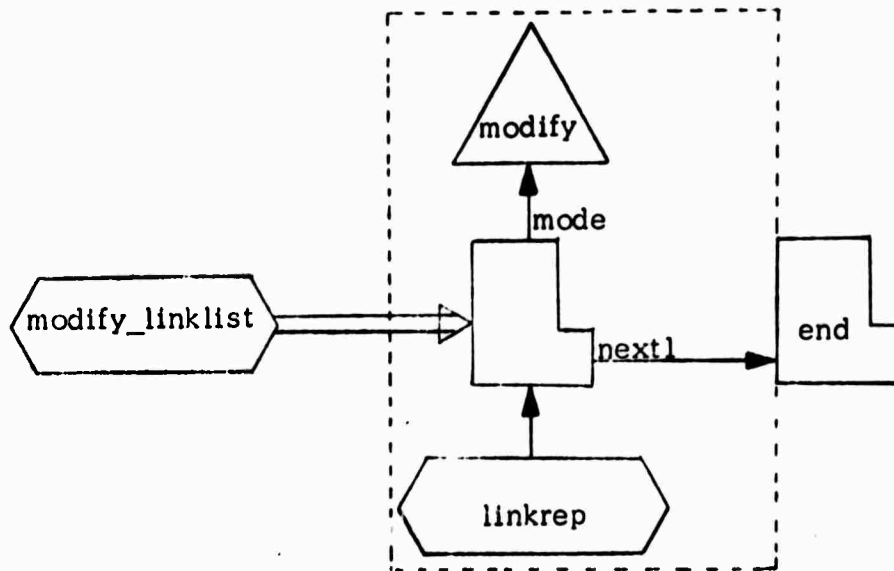
The processing of a 'linkrep' depends on its mode. 'test' and 'modify' 'linkrep's are processed simply by placing them at the ends of the 'test' and 'modify' lists respectively. 'frame' 'linkrep's are likewise placed on their respective list, but in addition, any destination 'noderep's are marked as being 'matched', possibly making additional 'linkrep's eligible for processing.

At the termination of the scanning phase, a non-empty active list indicates an error, for the rule then must contain a node that cannot be matched. If there has been no error, the three new lists of 'linkrep's are then attached to their respective points on the 'rule' node and compilation of the rule is complete.

In response to a successful compilation, the interpreter changes the state from 'clear' to 'compiled'.

Below is an augmentation of the syntax of AMBIT/G to describe compiled rules.





THE INTERPRETATION OF 'linkrep's

Each 'linkrep' appearing in a rule denotes an elementary action which is either a call on a user-defined function or the execution of a primitive operation. Which action is actually taken by the interpreter when it encounters the 'linkrep' L is determined dynamically and depends on:

- a) the mode of L;
- b) the number of arguments of L;
- c) the types of the arguments of L; and
- d) the f-name of L.

The mode of L is the destination of its 'mode' link and is either 'flag frame', 'flag test' or 'flag modify'. The arguments of L are those nodes of the user's data which are matched to 'noderep' nodes hanging off the 'org' link of L (see Figure 4-1). The f-name of L is the node 'L/name/rep/' (see Figure 4-2).

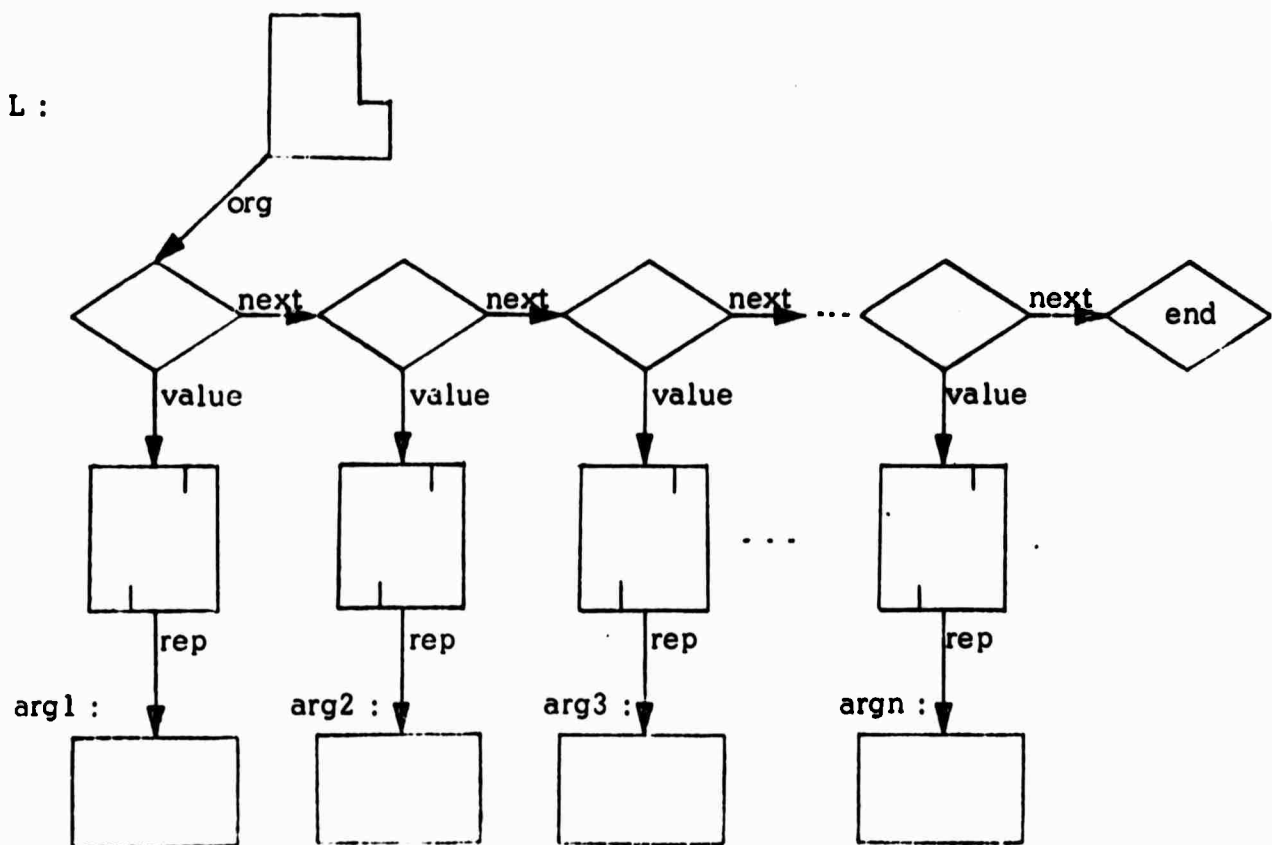


Figure 4-1: Arguments of L

L :

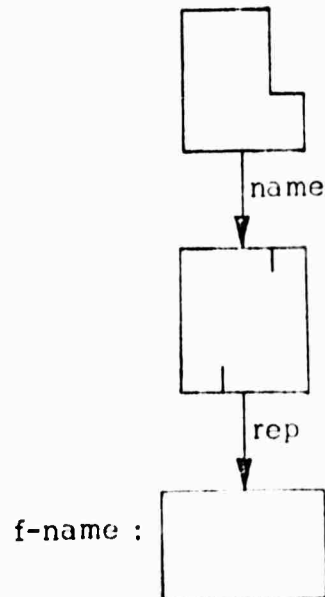


Figure 4-2: The f-name of L.

The mode of L determines the class of the action to be taken: modes 'frame' and 'test' result in a class read action, while mode 'modify' signifies a class write action. Basically, a read action is taken to obtain one or more values and is the generalization of reading a link. A write action returns no values and is executed solely for its side effects; it generalizes link writing.

For a 'linkrep' L of class read (write), a call is made by the interpreter on the primitive function 'read_function' ('write_function') with the f-name and list of argument types as parameters. This primitive returns a node of type either 'rule' or 'builtin'. If the node is of type 'rule', the interpreter then makes a class read(class write) call on a user function with execution beginning with that rule. Otherwise, the interpreter performs the action corresponding to that 'builtin' and class read (write). In most cases, this action is simply to pass the arguments (and sometimes other information) to the corresponding primitive routine.

For a class write action, there is nothing more to do. However, a class read operation returns one or more results. These results are then used by the interpreter either to set or to test the value of the 'rep' link of the 'noderep's hanging off of the 'dest' link of L. In mode 'test', equality must hold between corresponding results and 'rep' links or else the rule fails. In mode 'frame', a given result may be used either to set the 'rep' link or to verify a prior setting. Which of the two occurs depends on the setting of the 'sets' link of the particular 'noderep'. If it points back to the 'diamond' through which it was located, then a setting action takes place (see Figure 4-3) ; otherwise, a verification occurs (see Figure 4-4). An inequality in the verification indicates an error which the interpreter then reports.

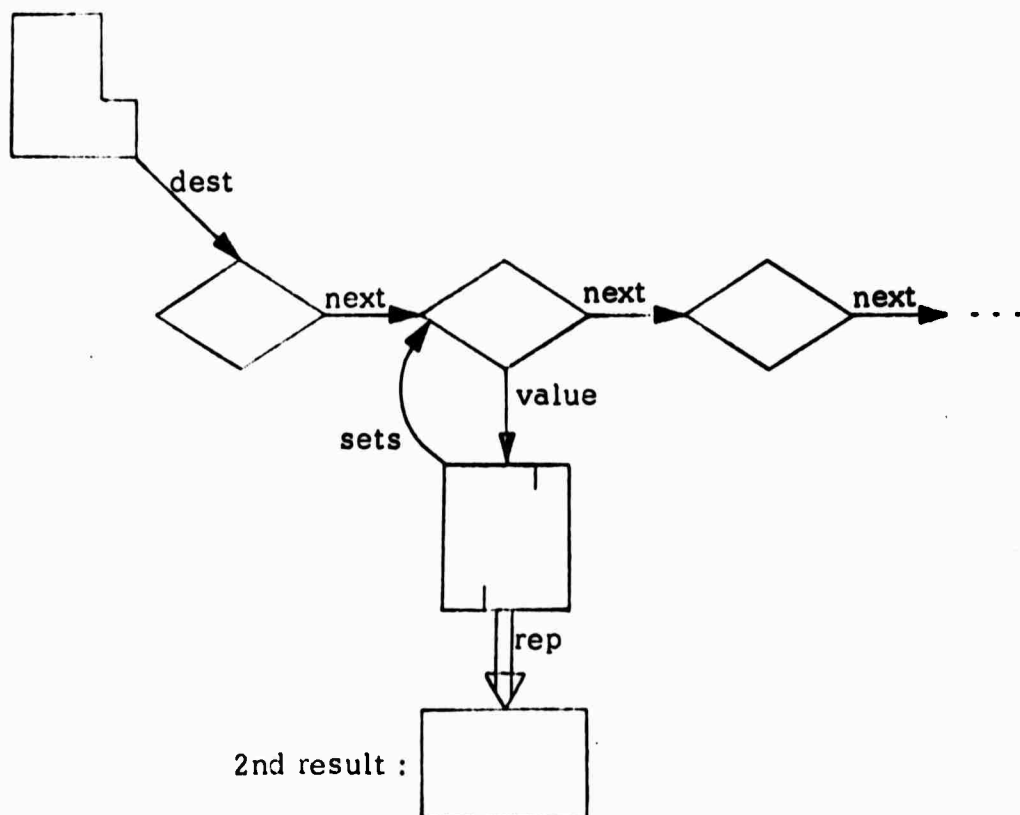


Figure 4-3 : Conditions for the setting of the 'rep' link.

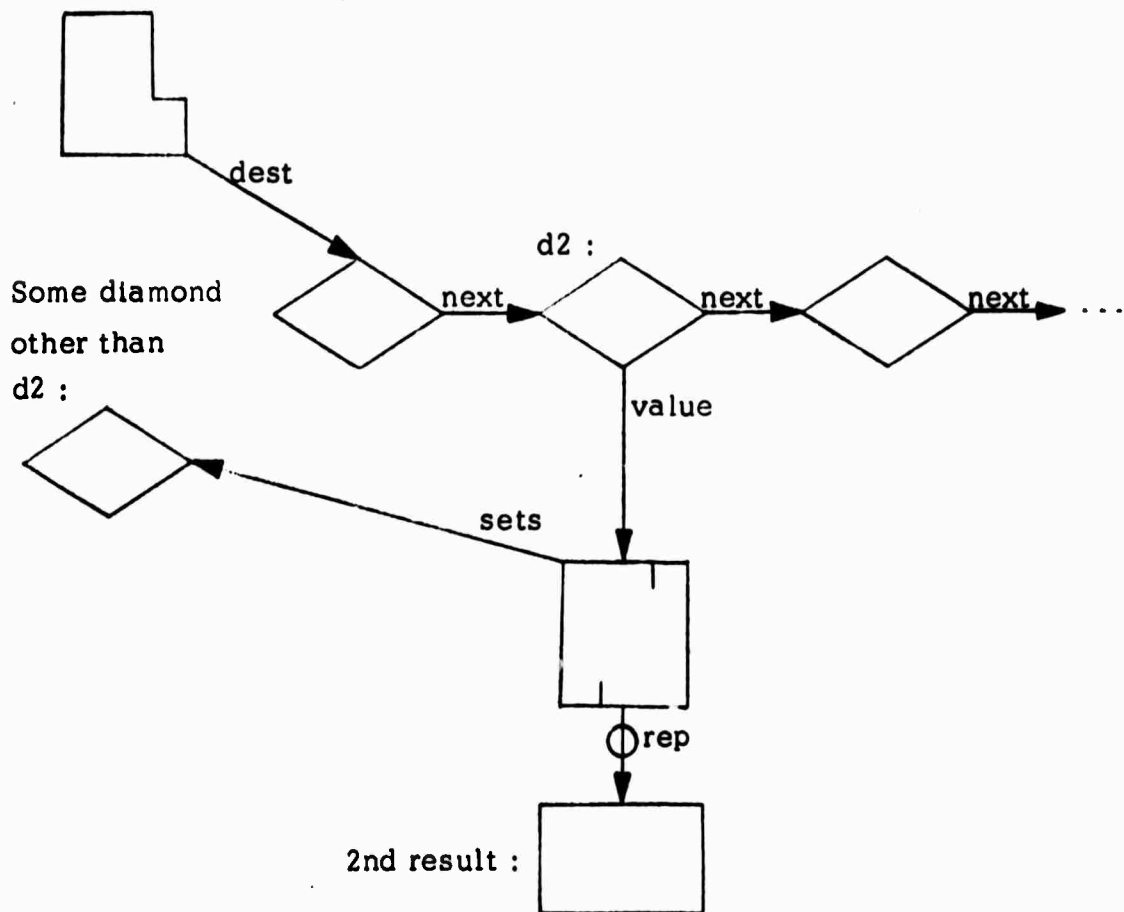


Figure 4-4: Conditions for the verification of the 'rep' link.

USER-DEFINED FUNCTIONS

There are three parts to a user-defined function: the definition, the call, and the return.

Function Definitions

A user function is defined by associating a 'rule' node with a particular call-class (i.e. read or write), f-name, and argument-type list. This is performed by a class write call on the primitive 'read_function' to define a class read call, or on 'write_function' to define a class write call.

AMBIT/G does not have any sort of block structure, and there is no well-defined collection of rules which can be called the "body" of the function.

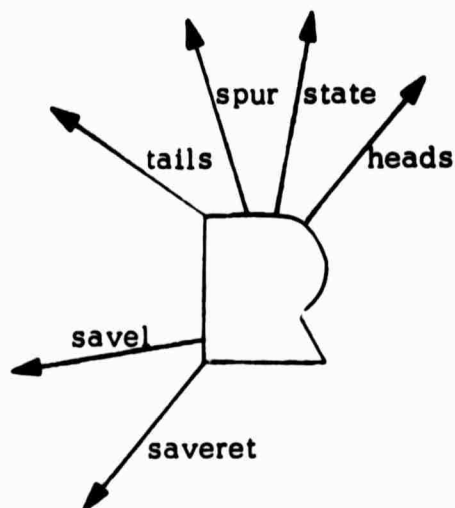
Rather, a given rule can be shared by any number of functions; this permits multiple-entry functions as a special case.

Function Calls

Once the interpreter determines that a user function is to be called, it performs the following set of actions:

- a) It sets up 'pipe's on the 'rule' node of the caller for the transmission of values between the caller and the function;
- b) It sets 'ptr next_rule/value' to the starting rule of the function to be called;
- c) It saves its current status on the 'rule' node of the caller;
- d) It sets 'ptr ret/value' to point to the 'rule' node of the caller, thus enabling the function to locate the call; and
- e) It begins interpreting 'rule go'.

Thus, all of the information relevant to the call is saved with the caller. The actual links used are shown below and summarized in the table which follows.



Links used in calling user functions

<u>Node type</u>	<u>Link</u>	<u>Destination</u>	<u>Meaning</u>
rule	tails	pipe	List of 'pipe's which contain the actual tail arguments to the user function called from within this rule.
	spur	user node	The actual link name that caused the user function to be invoked.
	heads	pipe	List of 'pipe's which contain the actual head arguments or which will receive the results of the user function called from within this rule, depending on the class of the call.
	state	flag	The mode of the link causing the call on the user function.
	savel	linkrep	The actual link representation that caused the call.
	saveret	rule	Used to save the old value of 'ptr ret/value'.

The arguments passed to the function depend on the class of the call, whether read or write. In either case, the interpreter builds two lists of 'pipe's equal in length to the 'org' and 'dest' lists of 'diamond's on the calling 'linkrep'. The actual origin arguments are then copied into the 'value' links of the 'tails' 'pipe's. For a class write call, the actual destination arguments are similarly copied into the 'value' links of the 'heads' 'pipe's, but for a class read call, these links are instead set to undefined (the node 'undef undef'). In either case, the f-name which caused the function to be invoked is stored as the destination of the 'spur' link of the 'rule' node of the caller, sometimes useful when the same code is to be used to implement several slightly different but similar functions.

The interpreter does not go directly to the desired function; rather, all function calls lead to 'rule go' which has a default definition which causes an immediate branch to the function. The reason for this indirection is to enable the user to extend or modify the action taken by the interpreter on a function call, for the user need only replace the default contents of 'rule go' with his own. It is to be emphasized that 'rule go' exists in the user's data base and is interpreted in just the same way as any other user rule. An example of an extension requiring modification of 'rule go' is the recursion package which extends the interpreter to handle recursive procedures. The reader is referred to the 'factorial' example for more details on this (at the end of Volume II).

The status that is saved consists of the state of the interpreter (whether 'frame', 'test', or 'modify'), the current 'linkrep' (the one causing the call), and the old value of 'ptr ret'. This information is saved on the links 'state', 'save1', and 'saveret' respectively.

The ordinary AMBIT/3 programmer may ignore most of the above details. He only needs to know that for a class read function, the arguments are to be found in the list of 'pipe's located by 'ptr ret/value/tails' and that the results to be returned by the function should be stored in the 'pipe's located by 'ptr ret/value/heads'. Similarly, for a class write function, both 'ptr ret/value/tails' and 'ptr ret/value/heads' contain arguments.

Function Returns

To return from a function, it suffices to branch to 'rule stop', but the programmer is instructed to branch always to 'rule ret', which has a default definition of always branching immediately to 'rule stop'. In this way, a modification of 'rule ret' will allow function returns to be intercepted much in the same way as function calls can be monitored by altering 'rule go'.

When the interpreter interprets 'rule stop', it performs almost an exact inverse of the five steps involved in a function call by doing the following:

- a) It turns its attention to the rule located by 'ptr ret/value' and halts if that value also happens to be 'rule stop';
- b) It restores the previous setting of 'ptr ret/value' from the 'saveret' link;
- c) It restores its previous status from the 'state' and 'savel' links;
- d) It processes any results produced by the function and frees the 'pipe's for later use; and
- e) It continues with the interpretation of the rule.

Recursion Faults

During the execution of a function a rule may be encountered whose state is 'frame', 'test', or 'modify', indicating that the rule is currently suspended because it contains a function call which is now being processed. This can occur only as a result of an attempted recursive function call. In this case, the interpreter does not try to execute the rule but rather points 'ptr next_rule/value' at it and then branches to 'rule help'. AMBIT/G does not support recursion, so 'rule help' normally branches immediately to 'rule error'. However, the user may supply his own version of 'rule help' to save the important information of the rule about to be executed, reset its state to 'compiled', and then branch to it. This is the final handle needed by the recursion package.

ERROR MESSAGES OF THE AMBIT/G INTERPRETER

The following is a list of the various error messages which may be typed out as a result of the interpreter's detecting an error condition. The use of three periods is to indicate a symbolic node name will be typed according to the state of the data.

1. System implementation error in the interpreter probably due to improper data; a frame does not match because " ... " is not the same as " ... " .
2. System implementation error in the interpreter probably due to improper data; a frame does not match because a link with origin " ... " and name " ..." points to destination " ... " instead of " ..." .
3. System implementation error in the interpreter probably due to improper data; a frame does not match because " ..." is not of type " ... " .
4. System implementation error in the interpreter due to an oversight by the implementor; a rule took an unexpected fall exit.
5. The interpreter is attempting to interpret a non-rule.
6. The interpreter is reporting a user-detected error.
7. The interpreter is attempting to interpret the rule " ..." which is in an unknown state of " ..." .

8. The interpreter is attempting to resume the interpretation of rule " ..." which is in an unknown state of " ..." .
9. The interpreter has detected an attempt to execute an undefined reading function.
10. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "link" .
11. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "type" .
12. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "read_function" .
13. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "write_function" .
14. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "name" .
15. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "link'" .
16. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "char" .
17. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "locate" .
18. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "load" .

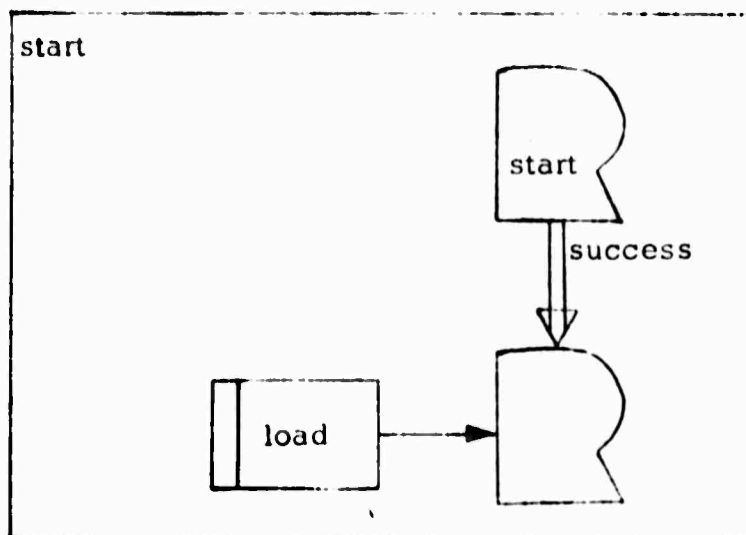
19. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "add" .
20. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "subtract" .
21. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "multiply" .
22. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "divide" .
23. The interpreter has detected a wrong number of tails or heads on a read-call on the builtin "sign" .
24. The interpreter has detected that the frame is inconsistent with the data graph.
25. The interpreter has detected an attempt to execute an undefined writing function.
26. The interpreter has detected a wrong number of tails or heads on a write-call on the builtin "link" .
27. The interpreter has detected a wrong number of tails or heads on a write-call on the builtin "read_function" .
28. The interpreter has detected a wrong number of tails or heads on a write-call on the builtin "write_function" .
29. The interpreter has detected a wrong number of tails or heads on a write-call on the builtin "link' " .
30. The interpreter has detected a wrong number of tails or heads on a write-call on the builtin "char" .

- 31. The interpreter has detected the "success" exit from a rule leads to " ... " , which is not a rule.
- 32. The interpreter has detected the "fail" exit from a rule leads to " ... " , which is not a rule.
- 33. The interpreter has detected a wrong number of results returned by a user function.
- 34. The compilation phase of the interpreter has detected the mode of a link is " ... " , which is neither "flag frame", "flag test", nor "flag modify".
- 35. The compilation phase of the interpreter has detected that a rule contains an unreachable node.
- 36. The compilation phase of the interpreter has detected that the destination of a "variability" link is " ... " , which is neither "flag fixed" nor "flag dummy" .
- 37. **System implementation error in the interpreter or loader probably due to improper data; a frame does not match because a link with origin " ... " and name " ... " points to destination " ... " , which is not of type " ... " .**

CHAPTER 5

THE LOADER

The AMBIT/G loader is called as a built-in read function as in the following sample rule:



Although all other built-in functions are defined in English, the loader was written as an AMBIT/G program, and thus its listing serves as a precise definition of its characteristics. However, we shall describe in this chapter its characteristics from a user or programmer viewpoint. Volume IV of this report contains a description of the loader as an AMBIT/G program. A programmer may wish to study the loader as an example of a large AMBIT/G program, but studying the AMBIT/G interpreter would also serve this purpose, and is probably more useful.

There is no important reason for the loader's being built-in other than convenience and efficiency and the bootstrap problem of how to load anything (the loader itself). As a function, however, the loader uses no more facilities than any AMBIT/G user function. A similar statement cannot be made, of course, for other built-in functions such as the primitives to read and write links. As the system exists, however, the loader is known to the interpreter as a primitive and is also known to the system in the way in which initialization leads to the start of execution. Namely, the initializer calls

upon the interpreter with an argument of 'rule start' shown as a sample rule above. That rule is interpreted (and compiled); this amounts to an automatic call on the loader. Note how 'rule start' includes a modification link which causes the modification of its own 'success' exit after loading to the first rule of the loaded program (or data).

OVERVIEW OF THE LOADER

The loader is called as a function with no arguments and one result. That result is a rule node which is meant to be the starting rule of the user's program. If the loader is called other than by an interpretation of 'rule start' the result may be used or not according to the programmer's fancy.

The loader reads source input one character at a time from the source file (e.g., 'foo.ambitg') by making read calls on the built-in function 'char'. It analyzes its input one statement at a time. Normally, a statement corresponds to one input line; however, many statements may be included on one line by using semicolons, and there is a method for continuing any statement across any number of input lines.

As dictated by the statements it reads, the loader deals with one encoded page at a time. We consider the true input to the loader to be pages of diagrams. Each page is hand-translated from these diagrams into several statements which represent the diagrams, but do not include coordinate (position) information. Just those aspects of connectivity which are essential to the AMBIT/G meaning of the page are encoded.

As the loader processes a page, it does not create any data. Recall that all AMBIT/G nodes were created at initialization, and their existence is permanent for the duration of AMBIT/G execution. All the loader does to AMBIT/G data, therefore, is connect various nodes by links which it sets. Usually, any link set by the loader was undefined (pointing at 'undef undef'), but no check is made for this. The loader is also capable of defining links, which it does by making write calls on the built-ins 'read_function' and 'write_function'.

The loader makes extensive use of the built-in 'locate', often to find an unnamed node of a given type. In such a case, if that node is not linked to be accessible from an accessible node it is lost in the data base forever.

Although the loader is used to load data, data can include an AMBIT/G rule as a glorified node. The loading of programs makes extensive use of this feature. The loading of a rule causes the representation of that rule in AMBIT/G data according to the specifications given in the chapter on representation of programs. In making up the representation of a rule, the loader links together a 'rule' node with unnamed 'linkrep's, 'diamond's, and 'noderep's along with the various named nodes used in rule representations. Each named node explicitly mentioned in a rule is located by the loader and ends up as the destination of a 'rep' link of a 'noderep' in that rule's representation.

Since the loader can be called by a user program as well as by the initial 'rule start' it does not output anything to the terminal unless an error condition is detected. An error causes an indicative message to be typed and execution to be terminated.

ERROR MESSAGES OF THE AMBIT/G LOADER

The following is a list of the various error messages which may be typed out as a result of the loader's detecting an error condition. The use of three periods is to indicate a character string will be typed according to the state of the data or the loader input. The following line is typed along with every loader error message except the last:

AMBIT/G Error: detected by the loader at statement n on page p
where n is an integer, and p is a page-title (string of characters).

1. System implementation error in the loader probably due to improper data; a frame does not match because " ... " is not the same as " ... " .
2. System implementation error in the loader probably due to improper data; a frame does not match because a link with origin " ... " and name " ... " points to destination " ... " instead of " ... " .

3. System implementation error in the loader probably due to improper data; a frame does not match because " ... " is not of type " ... " .
4. System implementation error in the loader due to an oversight by the implementor; a rule took an unexpected fail exit.
5. The first statement read by the loader does not begin with a " - " .
6. " ... " is an unknown statement.
7. A type-name is missing on a node on a loader page.
8. A page-name is missing for a "-rule-" or "-ruleref-" on a loader page.
9. There is an unknown "-" line in a rule contents on a loader page.
10. There is an incomplete statement on a loader page.
11. " ... " is an extra special character on a loader page.
12. " ... " is an undeclared page-name on a loader page.
13. System implementation error in the interpreter or loader probably due to improper data; a frame does not match because a link with origin " ... " and name " ... " points to destination " ... ", which is not of type " ... " .

AMBIT/G LOADER SYNTAX

In designing the syntax for the loader, we established some requirements based on the readability of a loader page. A loader page is intended to be a sequence of text lines representing one physical page or sheet of AMBIT/G diagrams. Such a page may be arbitrarily large and contain diagrams representing any mixture of rules, data, and link definitions.

We shall provide a grammar for the syntax in a BNF-like specification language. However, we will first give informally the lexical conventions which apply to loader input to produce the syntax of a statement.

The readability requirement affects the interpretation of spaces, tabs, and new-lines (carriage returns). Since we can't determine by reading, spaces and tabs are not distinguished; furthermore, to avoid a requirement for counting, any amount of space on a typed line is treated as a single space. Since a reader cannot see trailing space it is ignored. Similarly, since the exact position of a left margin can be uncertain, leading space is ignored. Since line spacing is difficult to see, any blank line (even if it contains a space) is ignored. A loader page should otherwise include only visible printing characters of the ASCII character set.

Another requirement of the syntax is its ability to represent AMBIT/G node names which may consist of any sequence of printing characters. Thus when a special meaning is given to a character, such as semicolon, there must also be a method of inputting a semicolon as a normal text character. This has been accomplished by giving the dollar sign a special meaning as a protection character. Namely, when a statement includes a dollar sign which is itself unprotected, that dollar sign protects the very next character from having special meaning. The following list presents all printable characters which have special meaning in the loader syntax:

\$; / ! ? () : * ,

Any of these special characters must be protected in order to be understood as a normal text character. Any other character may be protected by a dollar

sign, but protection has no effect; this means a user can't go wrong if he protects a character when he is not sure whether it has a special meaning.

When a dollar sign ends a text line, it can be considered as a protection of the new-line (or carriage return) character. Normally the end of a line denotes the end of a statement. If a line ends with an unprotected dollar sign, however, that special meaning is nullified, and instead the statement is interpreted as continuing on the next line. Any number of continuations may be given. Note that since leading and trailing spaces are ignored, it may be necessary to protect a space at the beginning of a line which is part of a continued statement.

We have described how an individual statement may spread over several lines. The complementary ability to include several statements on one line is provided by using an unprotected semicolon as a statement terminator.

Input to the loader may include comment statements anywhere. A comment statement begins with an asterisk and ends either at the end of a line (which is not continued) or at an unprotected semicolon. Within a comment it is not necessary to protect any other special characters.

The loader performs its inputting by calling a function named 'get_statement'. That function takes into account all the lexical conventions just described. It reads the input stream character by character and produces an output stream of one loader statement each time it is called. Its output does not include null statements nor comment statements. Protective dollar signs are removed, and unprotected special characters are converted into objects which are not ASCII characters, but serve as an extension to the character set. We will, however, denote these special objects in the loader syntax by the corresponding ASCII character. Each space produced by 'get_statement' is also one of these special objects; it will be denoted by 'SP'.

Spacing and the use of separate lines are used in the loader syntax grammar for readability and do not affect the meaning. Ends of statements are denoted by a semicolon, but recall a semicolon is required to end a

statement only when more text follows on the same line. Meta-variables (non-terminals) are underlined strings consisting of alphabetic characters and hyphens.

A vertical bar in the grammar represents disjunction. A matching pair of vertical brackets indicates they enclose an optional construct; namely, the syntax allows for either zero or one occurrence. A matching pair of vertical curly braces indicates they enclose a construct which may be repeated any number of times, including zero.

The grammar is designed not to be minimal nor reflect its implementation; it is supposed to be easily understood and to correspond to what it represents. The grammar provides the definition of a 'loader-input' which is the string of characters which the loader processes as a result of one invocation unless an error condition is detected. One 'loader-input' consists of any number of 'loader-page's finally followed by a 'start-statement'. Each 'loader-page' begins with a header which should contain a 'page-title' since that title is typed when an error is detected by the loader on that page. Then any number of 'data-node's are specified; each one may either refer to a 'node' in the data of a given type (possibly unnamed) or a 'rule' or a reference of a rule ('ruleref'). Next (and last) on a page are representations of links ('data-link's) to be initialized in the data. This includes 'success' and 'fail' links connecting rules on the page.

A rule is an elaborate generalization of a node with a substructure resembling the super-structure of a 'loader-page'. Namely, each 'rule' begins with a header and then contains specifications of 'rule-node's followed by 'rule-link's. Unlike a 'page', the end of a rule is clearly indicated by an '-endrule-' statement.

To permit the mentioning of 'data-node's in 'data-link's each node specification includes a 'page-name' which is an identifier whose scope is the current page. (We now realize this kind of identifier would have been better named 'instance-name', but 'page-name' is used throughout programs and documentation so it has been kept; we apologize to the reader.) 'rule' and 'ruleref' specifications also include a 'page-name' for the same reason. The user should think of the page-name as corresponding to a box in the

AMBIT/G diagram: either a box representing a node, a larger one representing a rule (with contents), or a rounded rectangle representing a rule reference. We have found it convenient to choose page-names in the spirit of encoding AMBIT/L diagrams: the page may be thought of as having a grid of rows and columns. Rows are named 'a', 'b', 'c', etc., and columns are named with the integers beginning at '1'. A page-name is then chosen according to the coordinate position of the box it represents. For example, 'b3' is the third column of the second row. The decision of whether to adopt this suggestion is at the user's discretion. He may prefer to employ words of mnemonic value.

Within a rule, 'rule-page-name's serve the same purpose as 'page-name's, but their scope is only the current rule. Thus if a page contains more than one rule, each rule may include the very same rule-page-names, and furthermore, they may be the same as page-names employed on the loader-page at large.

The syntax allows for specification of 'data-link's and 'rule-link's in two different forms: using a textual 'link-label' or by referencing a node by its page-name. The latter form corresponds to the more primitive view of a link with a spur to a node. The loader processes a 'link-label' xyz by treating it as a spur to the node 'link xyz'.

These explanatory notes have been intended to give the reader a "push" into the grammar; we do not consider it necessary to discuss all of its details. Following the grammar is a small sample of the encodement of a complete three-page AMBIT/G program,

Grammar of Loader Syntax

1. loader-input → { loader-page } start-statement
2. loader-page → page-header
{ data-node }
[-links- ; { data-link }]
3. page-header → -page- [[SP] page-title] ;
4. data-node → node | rule | ruleref
5. node → typed-node | named-node
6. typed-node → page-name sep type ;
7. named-node → page-name sep type sep subname ;
8. rule → rule-header
{ rule-node }
[--- ; { rule-link }]
-endrule- ;
9. rule-header → -rule- [SP] page-name [SP label] ;
10. rule-node → unnamed-rule-node | tested-typed-rule-node |
typed-rule-node | named-rule-node
11. unnamed-rule-node → rule-page-name ;
12. tested-typed-rule-node → typed-rule-node ? ;
13. typed-rule-node → rule-page-name sep type ;
14. named-rule-node → rule-page-name sep type sep subname ;
15. rule-link → rule-link-org sep rule-link-name sep rule-link-dest ;
16. rule-link-org → rule-link-org-dest
17. rule-link-name → frame-rule-link | test-rule-link |
modify-rule-link
18. rule-link-dest → rule-link-org-dest

19.	<u>frame-rule-link</u>	→	<u>basic-rule-link</u>
20.	<u>test-rule-link</u>	→	<u>basic-rule-link</u> ?
21.	<u>modify-rule-link</u>	→	<u>basic-rule-link</u> !
22.	<u>basic-rule-link</u>	→	<u>labelled-rule-link</u> <u>spurred-rule-link</u>
23.	<u>labelled-rule-link</u>	→	<u>link-label</u>
24.	<u>spurred-rule-link</u>	→	: <u>rule-page-name</u>
25.	<u>rule-link-org-dest</u>	→	<u>rule-page-name</u> ([<u>SP</u>] [<u>rule-page-name</u> { <u>list-sep</u> <u>rule-page-name</u> } [<u>SP</u>]))
26.	<u>ruleref</u>	→	-ruleref- [<u>SP</u>] <u>page-name</u> [<u>SP</u> <u>label</u>] ;
27.	<u>data-link</u>	→	<u>deflinks</u> <u>link</u>
28.	<u>deflinks</u>	→	-deflinks- [<u>SP</u>] <u>type</u> [<u>SP</u>] ([<u>SP</u>] <u>link-name</u> { <u>list-sep</u> <u>link-name</u> } [<u>SP</u>]) ;
29.	<u>link</u>	→	<u>link-org</u> <u>sep</u> <u>link-name</u> <u>sep</u> <u>link-dest</u> ;
30.	<u>link-org</u>	→	<u>page-name</u>
31.	<u>link-name</u>	→	<u>labelled-link</u> <u>spurred-link</u>
32.	<u>labelled-link</u>	→	<u>link-label</u>
33.	<u>spurred-link</u>	→	: <u>page-name</u>
34.	<u>start-statement</u>	→	-start- [[<u>SP</u>] <u>label</u>] ;
35.	<u>page-name</u>	→	<u>limited-identifier</u>
36.	<u>rule-page-name</u>	→	<u>limited-identifier</u>
37.	<u>label</u>	→	<u>subname</u>
38.	<u>link-label</u>	→	<u>subname</u>
39.	<u>type</u>	→	<u>identifier</u>
40.	<u>subname</u>	→	<u>identifier</u>
41.	<u>sep</u>	→	<u>SP</u> , /
42.	<u>list-sep</u>	→	<u>SP</u> , [<u>SP</u>]
43.	<u>page-title</u>	→	< any string without statement terminator >
44.	<u>limited-identifier</u>	→	< any <u>identifier</u> which does not begin with a minus sign >

45. identifier

→ < any string of printing characters >

(END)

A SAMPLE ENCODEMENT

Below is an actual listing of the file 'reversel.ambitg' representing the small yet complete three-page AMBIT/G program which follows.

reversel.ambitg 12/30/70 2035.3 est Wed

```
-page- r1-1
-links-
-deflinks- p (d)
-deflinks- end (r)
-deflinks- c (r,d)
```

```
-page- r1-2
a1 p x
a5 p y
b1 c
b2 c
b3 c
b4 c
b5 end c
c1 char P
c2 char O
c3 char T
c4 char S
```

```
-links-
a1 d b1
a5 d b5
b1 r b2
b2 r b3
b3 r b4
b4 r b5
b5 r b5
b1 d c1
b2 d c2
b3 d c3
b4 d c4
```

(Cont' on next page)

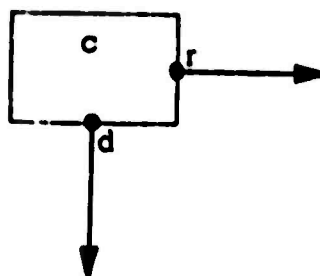
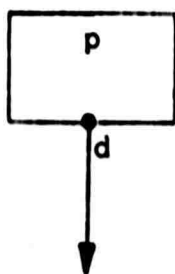

```
-page- r1-3
-rule- r1 reverse-1
a1 p y
a3 p x
b1
b2 c?
b3
---
a1 d b1
a1 d1 b2
a3 d b2
a3 d1 b3
b2 r1 b1
b2 r b3
-endrule-

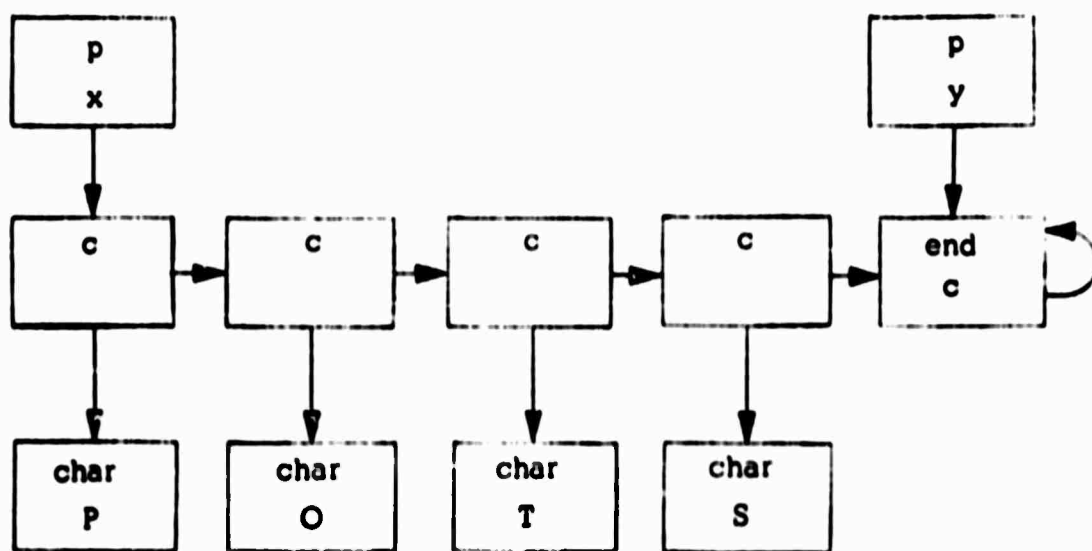
-ruleref- r2 stop

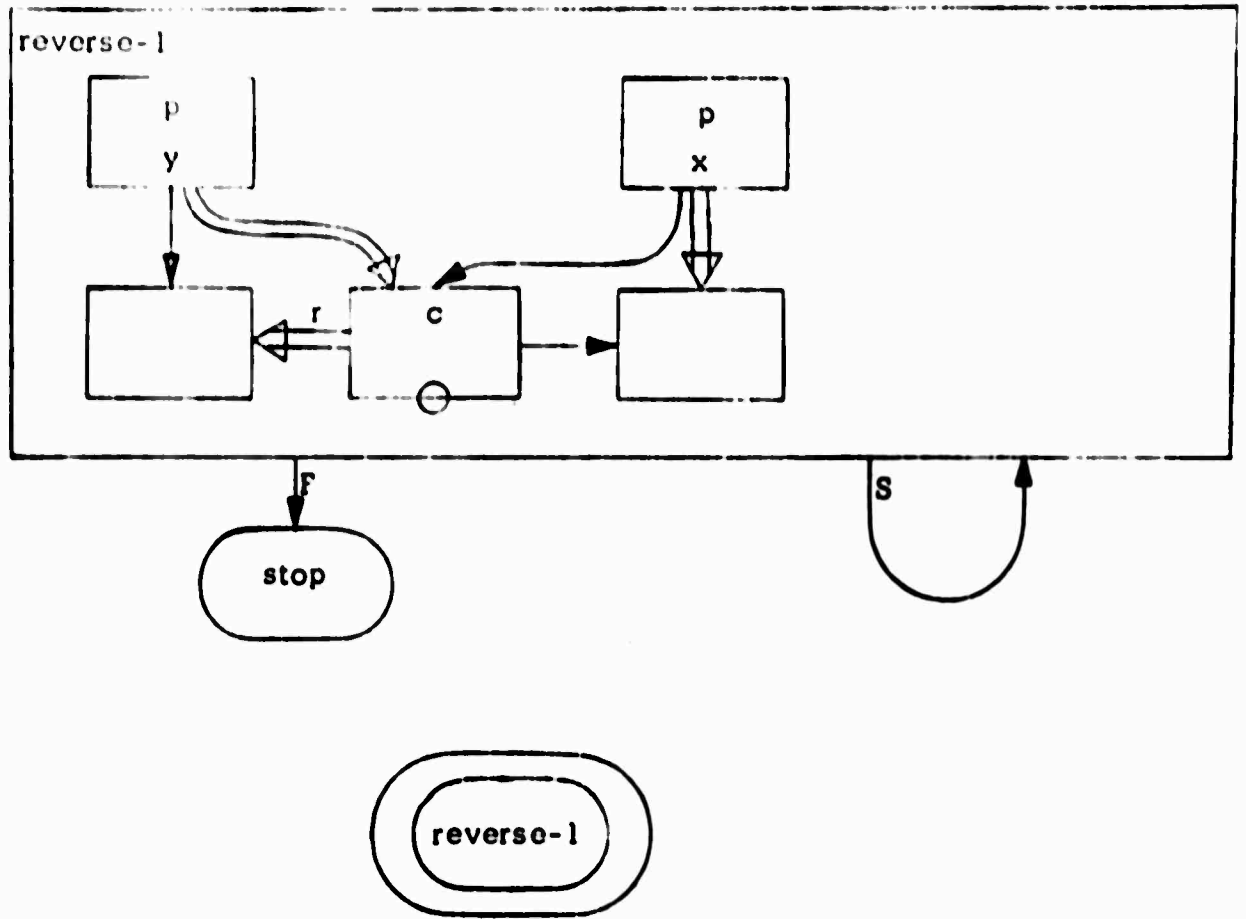
-links-
r1 success r1
r1 fail r2

-start- reverse-1
```

reversel







A SAMPLE ERROR

The following page is terminal output of an AMBIT/G run on Multics which causes a loader-detected error condition (number 12). Following the listing of the run is a listing of the program which caused the error. The arrows added to the output indicate lines typed in by the user.

→ hmu

Multics 13.1, load 8.0/41.0; 7 users

r 1441 .248 0+22

→ ambitg foo
AMBIT/G

AMBIT/G Error: detected by the loader at statement 5 on page "hello to you"
"c1" is an undeclared page-name on a loader
page.

This error occurred while interpreting the rule "rule start".
The interpreter was processing the link "()" load "()".

r 1441 15.172 0+135

→ pr foo.ambitg

foo.ambitg 12/27/70 1442.3 est Sun

-page- hello to you
-rule- r1 foo
a1 cell x
b1 cell y

(a1, c1) read_function! (a1)
-endrule-
-start- foo

r 1442 .368 0+11

CHAPTER 6

INITIALIZATION AND THE BUILT-IN SYSTEM

To make an AMBIT/G run using the Multics AMBIT/G System a user must prepare two source files. One file is for normal input to the system; it will be read by the AMBIT/G loader and by the user's program calling upon the built-in function which reads from the input file. The other file is read only by the AMBIT/G initializer early in the run and it plays no further role after initialization is complete; this file is called the hint file since its role in the system is considered to be outside of the definition of the AMBIT/G language. We call the contents of the hint file "hint information" or just "hints".

This chapter describes the syntax and semantics of the hint file, and a user's view of the initialization process is given. It also contains a complete description of the initial state of the AMBIT/G System as seen by a user's program. This includes all built-in nodes, defined links, initial links, and built-in rules. All built-in functions other than the loader are described and their built-in definitions are given.

HINTS

A hint file has three parts. It begins with any number (including zero) of settings of hint variables which control the AMBIT/G System initialization. These variables have default settings for every run, but the hint file can override the default settings. However, these overriding settings must be consistent with initialization of the built-in data and functions. The following table outlines this information:

<u>Hint Variable</u>	<u>Default Value</u>	<u>Meaning, Restrictions</u>
smallest_integer	-999	'integer's are built-in nodes, and these variables establish the range of created 'integer' nodes. 'smallest_integer' must not be greater than 'largest_integer'. Note that
largest_integer	999	

'largest_integer' should be greater than the number of statements included on a loader page, since the loader tallies a count of statements using 'integer's.

function_arguments	10	This variable indicates the maximum number of tail arguments which can be included in a function call. It also is an upper limit on the number of arguments for which a function may be defined. It must be at least 2.
defns_size	5000	This variable is used to limit the size of the segment used to store function definitions. A larger number does not raise costs, but it must be less than 65000. Its moderate setting may be useful for catching errors in a program which gets into a loop defining too many functions. It must be at least 347 plus twice the value of 'function_arguments'.
names_size	1000	This variable indicates the maximum number of names which may be accommodated in the symbol table (names_segment). It must be at least 200.
name_length	25	This variable indicates the maximum number of characters which may be used in a name. It must be at least 14.

The second portion of the hint file specifies the names and counts of all types of terminal nodes (having no links), which are to be created in additon to the built-in nodes. The given count must be greater than zero. The following list indicates the built-in terminal type names and the number of nodes which are always created.

<u>Terminal type</u>	<u>Count</u>
flag	13
link	36 (built-in) + 100 (for user)
builtin	15
undef	1
boolean	2
char	128
special	9

integer	largest_integer - smallest_integer + 1
type	(19 plus additional ones defined by the user in the hints)

The hint file may include one mention of any of the built-in terminal types other than 'integer' and 'type'. The given associated count will be added to the built-in count. For example, if the user wishes to extend the class of nodes of type 'char' by eight more nodes, he would include in the second portion of the hint file the terminal type name "char" and the integer '8'. Thus 136 nodes of type 'char' would be created.

The third portion of the hint file specifies the names, counts, and maximum number of links of all types of non-terminal nodes which are to be created other than the built-in nodes. The given count must be greater than zero.

Since various non-terminal nodes are used to represent a user's program and the need for certain other nodes varies according to the ways in which a program uses functions, the overriding mechanism is more complex than for terminals. If the hint file does not mention a built-in non-terminal type then the number of nodes of that type created is the sum of the built-in count plus the default additional count. However, if a built-in non-terminal is included in the third portion of the hint file the given count is added to the built-in count, thus overriding the default additional count. Furthermore, the given number of links is added to the built-in number of links. The given number of links for a built-in non-terminal may therefore be any positive integer including zero. The given number of links for user-defined non-terminal types must be greater than zero. The following list presents the relevant information for built-in non-terminals.

<u>Non-Terminal Type</u>	<u>Count</u>	<u>Default Additional</u>	<u>Number of Links</u>
rule	6	50	12
linkrep	11	500	6
pipe	4	100	2
cell	4 + function_arguments	function_arguments	2
charconn	103	1900	2

ptr	2	0	1
noderep	13	500	3
circle	42	0	1
diamond	15	1000	2
pname	2	100	3

A type name must not appear more than once in the hint file. Also, a built-in terminal cannot be promoted to a non-terminal.

Hint Syntax

The syntax of the hint file corresponds to the requirements for simple use of PL/T input functions. Below is given a grammar for the syntax in a BNF-like specification language. Spacing is used for readability and does not affect the meaning. Meta-variables (non-terminals) are underlined strings consisting of alphabetic characters and hyphens. A vertical bar in the grammar represents disjunction. A matching pair of vertical brackets indicates they enclose an optional construct; namely, the syntax allows for either zero or one occurrence. A matching pair of curly braces indicates they enclose a construct which may be repeated any number of times, including zero.

Grammar:

1. hints → { set-hint-variable NL }
; NL
{ terminal-spec NL }
" " NL
{ non-terminal-spec NL }
" " NL
2. set-hint-variable → hint-variable = integer
3. terminal-spec → "type" SP integer
4. non-terminal-spec → "type" SP integer SP integer
5. hint-variable → smallest_integer | largest_integer |
function_arguments | defs_size |
names_size | name_length
6. integer → [-] digit { digit }
7. digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
8. type → <any (non-null) string of printing characters ,
except double quotes must be in pairs>
9. SP → <space composed of any (non-null) mixture
of spaces and tabs>
10. NL → <new-line (carriage return)>

An Example

A short example of a hint file follows as an actual listing of the file 'reverse1.hints'. This file is the mate of the file listed as an example in the chapter on the loader. Furthermore, the 'reverse1' program is discussed in detail as the first example in Volume II of this report.

reverse1.hints 12/30/70 2035.3 est Wed

;		
"""		
"p"	2	1
"end"	1	1
"c"	4	2
"""		

BUILT-IN NODES

As it reads the three portions of the hint file, the initializer creates all nodes which can be used in the AMBIT/G run. It then attaches names to the named built-in nodes. These are named nodes of the interpreter and loader and other nodes which are part of the AMBIT/G System.

An AMBIT/G programmer should avoid using any built-in nodes in his programs other than for their intended purpose. The programmer is advised not to use type names which are built-in for anything but their intended use. For example, a programmer should not employ nodes of type 'circle' in a program to represent an arbitrary variable.

The following is a comprehensive list by type of all named built-in nodes. (Their order corresponds to ordering in the implementation of the 'nodes_segment'.)

flag

clear	compiled	frame	test	modify
ok	no	def	undef	fixed
dummy	any	general		

link

heads	spur	tails	state	success
contents	frame	test	modify	fail
savel	saveret	mode	org	name
dest	next	next1	value	sets
variability	rep	link'	read_function	write_function
locate	type	char	load	add
subtract	multiply	divide	sign	pname
node				

builtin

link	link'	read_function	write_function	locate
name	type	char	load	add
subtract	multiply	divide	sign	error

undef

undef

boolean

true

false

char

NUL	SOH	STX	ETX	EOT	ENQ	ACK
BEL	BS	HT	LF	VT	FF	CR
SO	SI	DLE	DC1	DC2	DC3	DC4
NAK	SYN	ETB	CAN	EM	SUB	ESC
FS	GS	RS	US	SP	!	"
#	\$	%	&	'	(	)
*	+	,	-	.	/	0
1	2	3	4	5	6	7
8	9	:	;	<	=	>
?	@	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
[	\	]	^	_	`	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	{		}
␣	DEL					

special

SP	/	,	!	?	(	)
:	*					

integer

(all integers between `smallest_integer` and `largest_integer`; e.g. ...

-5	-4	-3	-2	-1	0	1
2	3	4	5	...	)	

type

(any user-defined types plus:)

flag	link	builtin	undef	boolean	char	special
integer	type	rule	linkrep	pipe	cell	charconn
ptr	noderep	circle	diamond	pname		

rule

go	stop	error	help	ret	start
----	------	-------	------	-----	-------

linkrep

fl	tl	ml	begin	end
----	----	----	-------	-----

pipe

free	end	h
------	-----	---

cell

free	end	h
------	-----	---

charconn

free	end
------	-----

ptr

ret	next_rule
-----	-----------

noderep

(none)

circle

a	b	c	d	d1	d2	end
f	free	dest	l	l1	mode	n
r	ret	rule	fl	ml	tl	proceed
char	next_pname	pname_list	pname	node	org	name
prev_char	find1	find2	char_a	char_b	first_ch	last_ch
end_line	save_pname	special	find_pname	paren	page	line

diamond

end matched unmatched h

pname

end

BUILT-IN LINKS

The following is a complete list by node type of all defined links which are built-in. (Their order corresponds to ordering in the implementation of the 'nodes_segment' and 'defns_segment'.) All built-in link names are nodes of type 'link', and therefore the given names are their subnames.

rule

state	success	fail	saveret	savel	heads
spur	tails	contents	frame	test	modify

linkrep

org	name	dest	next	nextl	mode
-----	------	------	------	-------	------

pipe

next	value
------	-------

cell

next	value
------	-------

charconn

next	value
------	-------

ptr

value

noderep

rep	variability	sets
-----	-------------	------

circle

value

diamond

next	value
------	-------

pname

next	pname	node
------	-------	------

BUILT-IN FUNCTION DEFINITIONS

Each of the built-in functions is initially defined as if write calls has been made on 'read_function' and 'write_function'. These built-in definitions are initialized (in the 'defns_segment') before built-in links are defined. The following table summarizes these definitions. All built-in link names are nodes of type 'link' and therefore the given names are their subnames. (Their order corresponds to ordering in the implementation of the 'defns_segment'.)

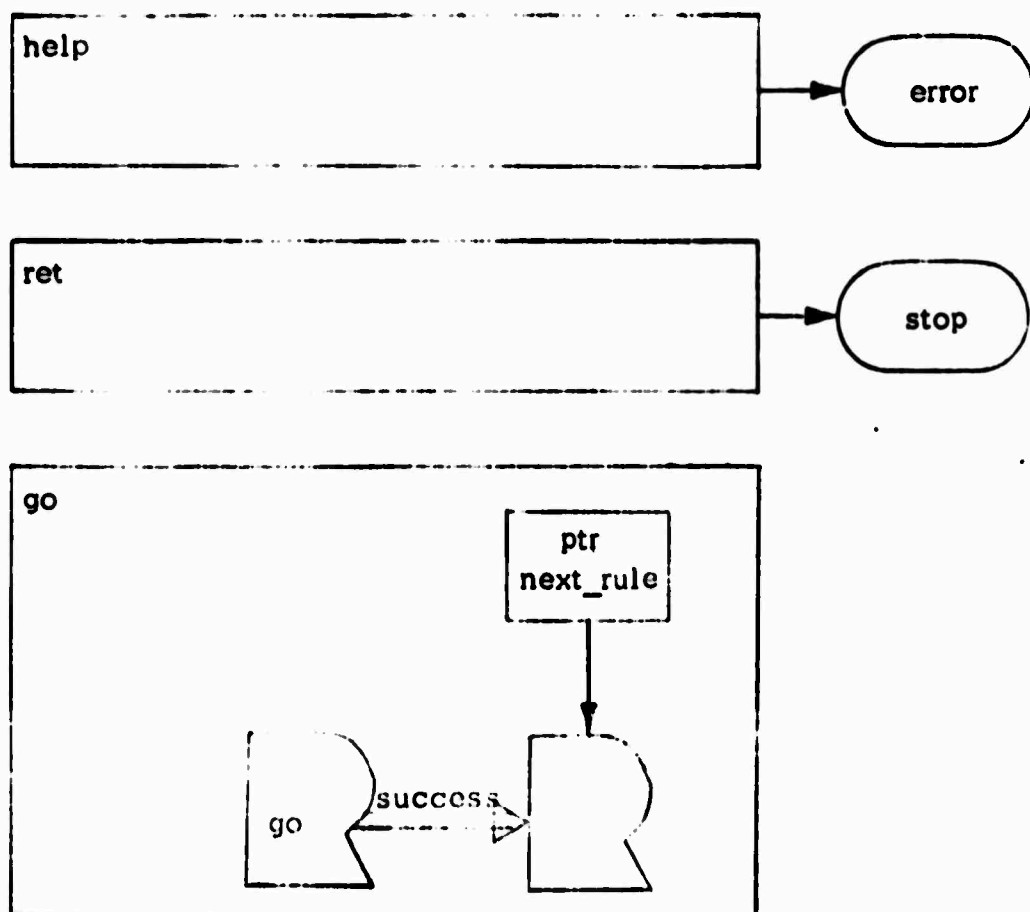
<u>Read/ Write</u>	<u>Link Name</u>	<u>Definition</u>	<u>Tail Types</u>
read	load	'builtin load'	(none)
read	type	'builtin type'	(any)
read	sign	'builtin sign'	integer
read	char	'builtin char'	(any)
write	char	'builtin char'	(any)
read	add	'builtin add'	integer, integer
read	subtract	'builtin subtract'	integer, integer
read	multiply	'builtin multiply'	integer, integer
read	divide	'builtin divide'	integer, integer
read	locate	'builtin locate'	type, charconn
read	name	'builtin name'	(any), charconn

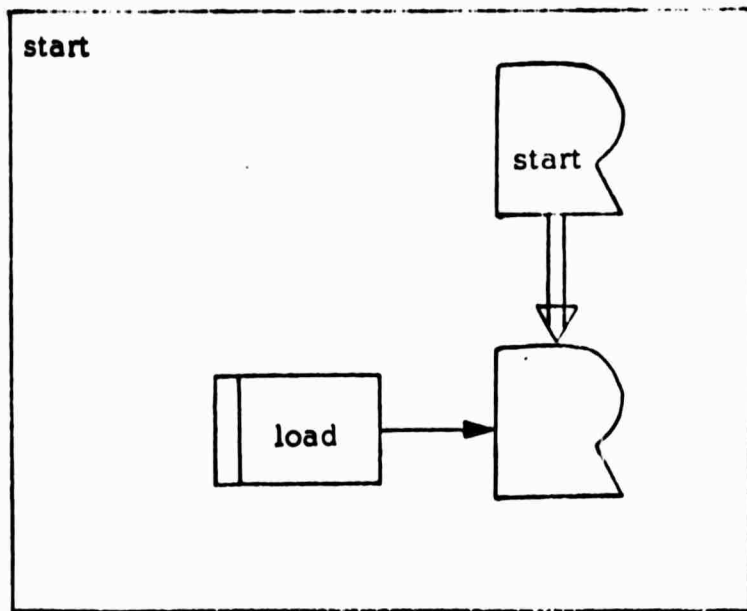
read	read_function	'builtin read_function'	cell, (any)
read	write_function	'builtin write_function'	cell, (any)
read	link'	'builtin link''	(any), (any)
write	read_function	'builtin read_function'	flag, (any)
write	write_function	'builtin write_function'	flag, (any)
write	read_function	'builtin read_function'	cell, (any)
write	write_function	'builtin write_function'	cell, (any)
write	link'	'builtin link''	(any), (any)

Although all of the above definitions are built-in, it is only necessary that the fourth last one be built-in. The existence of that one definition can serve as a bootstrap to define all of the others.

BUILT-IN RULES

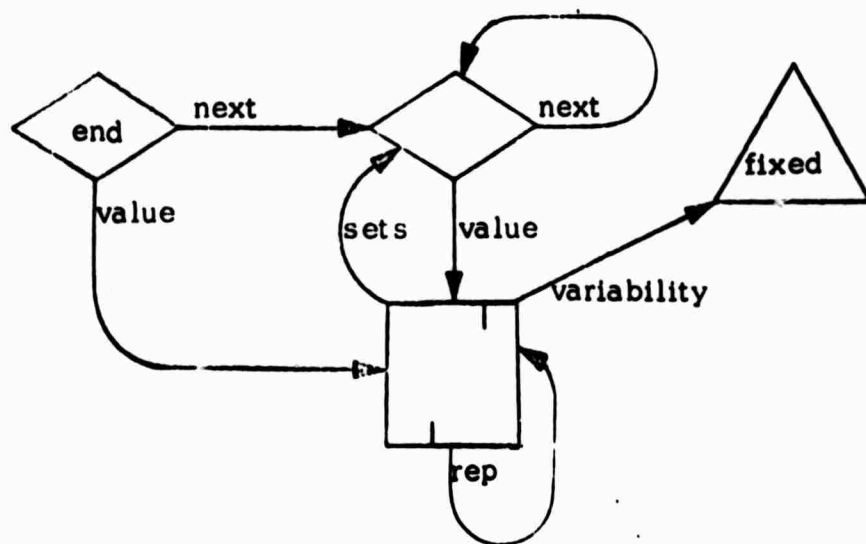
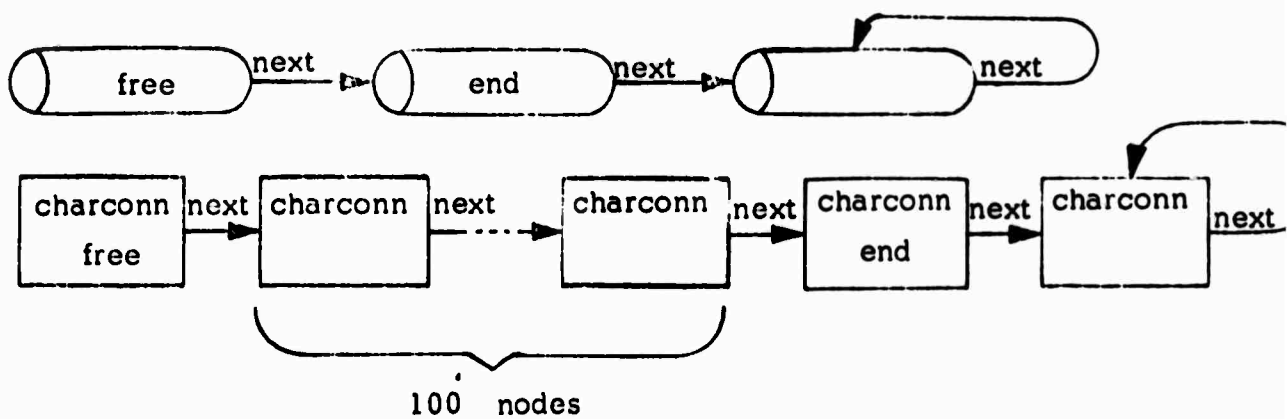
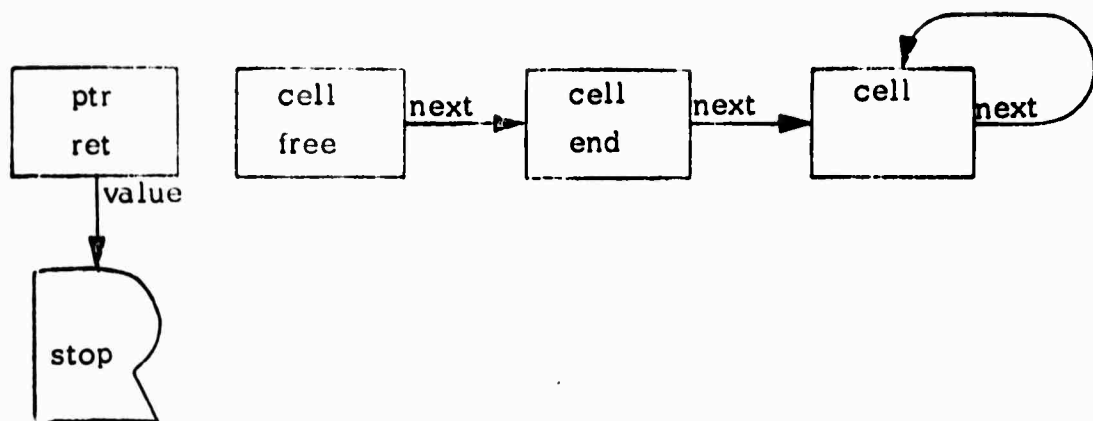
The initial AMBIT/G data graph includes six built-in rules. Two of these exist only as 'rule' nodes since the interpreter never attempts to look at their contents: 'rule stop' and 'rule error'. The other four built-in rules are initialized in the 'clear' state to be as follows. Note that the first two rules have no contents.





BUILT-IN DATA

The initial AMBIT/G data graph includes various nodes which are initially linked together in addition to the representations of the four above rules. All other built-in nodes have their built-in links undefined (pointing to 'undef undef'). The following diagram shows the initial data.



BUILT-IN FUNCTIONS

A description of each of the built-in functions (except the loader) of the AMBIT/G System is included here. Each description includes all possible error conditions and messages. Although a built-in function will normally be invoked by a call which depends upon its built-in definition, a program may give a built-in function as the definition of what is invoked for any arbitrary link name, etc.. When the AMBIT/G interpreter processes a link, it first finds out the definition of that link name as applied to the tails of the link. If that definition is a built-in then the interpreter checks for the number of tails and heads of the link; it reports an error if there is a discrepancy. The interpreter then gathers the arguments of the built-in function and performs a direct call on it as part of its interpretation.

To make it possible for the interpreter to be itself a legitimate AMBIT/G program, two nodes of type 'builtin' are involved with the reading and writing of links. The node 'builtin link' should be given as the head argument of a write call on 'read_function' (or 'write_function') to define for reading (or writing) a particular link name as a true link on a particular type of node. The node 'builtin link' is given as the head argument of such a call when defining a link to invoke the primitive link reading (or writing) function. This difference will be clarified for the reader by his observing the listing of the AMBIT/G interpreter where it processes these built-in functions. There is only one built-in function which is the primitive link reading function (and one for writing), and it will be described below.

type (read)

This function is called with one tail argument and one head result. The result of this function is the node of type 'type' which corresponds to the type of the argument. Since every node has a type there are no error conditions for this function.

link' (read)

This function is called with two tail arguments (arg1 and arg2)

and one head result. The type of arg1 is first determined; if arg2 is not defined as a true link on nodes of that type, error condition 'rl1' is signalled. Otherwise, the link with origin arg1 and name arg2 is read, and its destination is returned as this function's result. If the sought link is defined but has never been written, the result is the undefined node 'undef undef'.

The error messages for this function follow.

rl1: An attempt is being made to read an undefined link
 with origin " ... " and name " ... " .

link' (write)

This function is called with two tail arguments (arg1 and arg2) and one head argument (arg3) . The type of arg1 is first determined; if arg2 is not defined as a true link on nodes of that type, error condition 'wl1' is signalled. Otherwise, the link with origin arg1 and name arg2 is written to destination arg3. The previous destination of that link is lost.

The error messages for this function follow:

wl1: An attempt is being made to write an undefined link
 with origin " ... " and name " ... " to destination " ... " .

locate (read)

This function is called with two tail arguments (arg1 and arg2) and one head result. In general, arg1 is a 'type' node and arg2 is a list of printing characters; this function is used to locate by type (arg1) and subname (arg2) a particular node. If a null subname is given a unique node of the given type is located.

When initialization of an AMBIT/G run is complete all nodes are created according to the hints. During the execution, a call on 'locate'

either uses up a fresh node of the given type or it finds a named node which has already been located for the first time. Normally, the loader locates all nodes during the loading of data and rules. arg 2 is supposed to be a (possibly empty) list of connector nodes of type 'charconn'. These are forwardly-linked by 'next' links, and 'charconn end' terminates the list. Each other (if any) 'charconn' node of the list has a 'value' link which is supposed to point to a node of type 'char' which represents a printing character.

If arg 1 is not of type 'type', error condition 'loc 2' is signalled. If an element of the list of arg 2 is not of type 'char', error condition 'loc 3' is signalled. If an element of the list of arg 2 does not represent a printing character, error condition 'loc 4' is signalled. If a connector of the list of arg 2 is not of type 'charconn', error condition 'loc 5' is signalled. If the length of the list of arg 2 exceeds the maximum length of a name (according to hint variable 'name_length'), error condition 'loc 6' is signalled.

After all above checks are passed, if arg 2 is a non-empty list it is treated as a specification of the subname of the node being located. If that node is already known, it is returned as the result. If it is not already known (including the null subname case) a fresh node of the given type is obtained to be returned as result. If, however, all nodes of the given type have already been located, error condition 'loc 7' is signalled. If a non-null subname was given and a fresh node is to be obtained, but the system cannot accommodate another name (according to hint variable 'names_size'), error condition 'loc 8' is signalled.

The error messages for this function follow.

loc 1: The first argument of a call on the builtin "locate" is
" ... ", which is not of type "type".

loc 2: The second argument of a call on the builtin "locate" is
" ... ", which is not of type "charconn" .

- loc 3: The second argument of a call on the builtin "locate" is a list beginning with " ... " which includes a node of type "charconn" whose "value" link points to " ... ", which is not of type "char".

- loc 4: The second argument of a call on the builtin "locate" is a list beginning with " ... " which includes " ... ", which is an unprintable character.

- loc 5: The second argument of a call on the builtin "locate" is a list beginning with " ... " which includes a node of type "charconn" whose "next" link points to " ... ", which is not of type "charconn".

- loc 6: The second argument of a call on the builtin "locate" is a list of characters beginning with " ... " whose length exceeds capacity.

- loc 7: A call on the builtin "locate" is causing an attempt to locate a new node of type " ... " with second argument " ... ", and there is none.

- loc 8: A call on the builtin "locate" with arguments " ... " and " ... " is causing an attempt to create a new name, and that would exceed capacity.

name (read)

This function is called with two tail arguments (arg1 and arg2) and one head result. arg1 is any node, and this function returns the subname of the node as a list of its constituent characters. This result list is connected by 'charconn' nodes removed from a "free" list of 'charconn's headed by the 'charconn' node arg2. The list given as arg2 and the result list are forward-linked by 'next' links, and 'charconn end' terminates these lists. Each other (if any) 'charconn' node of the list has a 'value' link for pointing to a 'char' node.

If arg 2 is not a node of type 'charconn', error condition 'm1' is signalled. If a connector node of the list given as arg 2 includes a node to be used other than a 'charconn', error condition 'm 2' is signalled. If the given list of 'charconn's does not include enough of such nodes for returning the subname, error condition 'm 3' is signalled. If arg1 has no subname, the result is 'charconn end' and the given "free" list is not affected; otherwise, this function has the side-effect of removing those 'charconn' nodes which it uses from that "free" list.

Note this built-in function is available to a user program, but it is not used anywhere in the system other than one call in the interpreter to execute the function on behalf of the user's program.

The error messages for this function follow.

- m1: The second argument of a call on the builtin "name" is " ... " , which is not of type "charconn".
- m 2: The second argument of a call on the builtin "name" is a list beginning with " ... " which includes a node of type "charconn" whose "next" link points to " ... ", which is not of type "charconn".
- m 3: The second argument of a call on the builtin "name" is a list of nodes of type "charconn" beginning with " ... " which is too short to accommodate the subname of " ... " .

read_function (write)

This function is called with two tail arguments (arg1 and arg 2) and one head argument (arg 3) . It is used to define a reading function whose definition is arg 3. The link name which will later invoke that reading function is arg 2. arg1 is used as an indicator of the number and types of tail(s) which will be required to invoke that reading function. If there should be no restriction on the tails, arg1 should be 'flag general' . Otherwise,

arg1 must be a list where nodes of type 'cell' are the connectors. 'cell's are forwardly-linked by 'next' links, and 'cell end' terminates the list. Each other (if any) 'cell' node of the list has a 'value' link to point at the list element. Each given list element represents one tail argument of the reading function being defined. If the element is 'flag any' that particular tail argument may be any node. Otherwise, a list element must be a node of type 'type' to indicate the type of that particular tail argument which is required to invoke the reading function. If arg2 is 'cell end' an attempt is being made to define a reading function with no tail arguments.

If a definition is made when arg1 is not 'flag general' whose domain overlaps with that of a previous definition of the same link name, the newer definition overrides the previous one for the overlapped domain. This is discussed further in the section describing 'read_function (read)'.

arg3 must be either a node of type 'builtin' or 'rule'. If arg3 is 'builtin link', this is an attempt to define a link for reading; therefore, arg1 must be a list with exactly one element which is a 'type' node.

If arg3 is neither of type 'builtin' nor type 'rule', error condition 'drf1' is signalled. If arg1 is neither 'flag general' nor of type 'cell', error condition 'drf2' is signalled. If an element of the list given as arg1 is neither 'flag any' nor of type 'type', error condition 'drf3' is signalled. If a connector node of the list given as arg1 includes a node other than a 'cell', error condition 'drf4' is signalled.

If arg3 is 'builtin link', then this is an attempt to define a link for reading. If arg1 is not a list of one element, error condition 'drf5' is signalled. If another link cannot be defined for the given type because that would exceed the maximum number of links given in the hints, error condition 'drf6' is signalled.

In defining any reading function, if the system cannot accommodate that definition (according to hint variable 'defs_size'), an error condition 'drf7' is signalled. If the number of elements of the list given as arg1 exceeds the maximum number of arguments of a function (according to hint variable 'function_arguments'), then error condition 'drf8' is signalled.

The error messages for this function follow.

- drf1:** The head pointer of a write-call on the builtin "read_function" is " ... ", which is neither of type "builtin" nor of type "rule".
- drf2:** The first argument of a write-call on the builtin "read_function" is " ... ", which is neither the node "flag general" nor a node of type "cell".
- drf3:** The first argument of a write-call on the builtin "read_function" is a list beginning with " ... ", which includes a node of type "cell" whose "value" link points to " ... ", which is neither the node "flag any" nor a node of type "type".
- drf4:** The first argument of a write-call on the builtin "read_function" is a list beginning with " ... " which includes a node of type "cell" whose "next" link points to " ... ", which is not of type "cell".
- drf5:** A write-call on the builtin "read_function" is an attempt to define the link " ... ", and the first argument is " ... ", which is not a list of one node of type "type".
- drf6:** A write-call on the builtin "read_function" is an attempt to define the link " ... " on nodes of type " ... ", and another link cannot be defined for this type.
- drf7:** A write-call on the builtin "read_function" is causing an attempt to make a new definition, and that would exceed capacity.

drf8: The first argument of a write-call on the builtin "read_function" is a list beginning with " ... " whose length exceeds the maximum number of arguments allowed for a function definition.

write_function (write)

This function is called with two tail arguments (arg1 and arg2) and one head argument (arg3). It is used to define a writing function whose definition is arg3. The link name which will later invoke that writing function is arg2. arg1 is used as an indicator of the number and types of tail(s) which will be required to invoke that writing function. If there should be no restriction on the tails, arg1 should be 'flag general'. Otherwise, arg1 must be a list where nodes of type 'cell' are the connectors. 'cell's are forwardly-linked by 'next' links, and 'cell end' terminates the list. Each other (if any) 'cell' node of the list has a 'value' link to point at the list element. Each given list element represents one tail argument of the writing function being defined. If the element is 'flag any' that particular tail argument may be any node. Otherwise, a list element must be a node of type 'type' to indicate the type of that particular tail argument which is required to invoke the writing function. If arg2 is 'cell end' an attempt is being made to define a writing function with no tail arguments.

If a definition is made when arg1 is not 'flag general' whose domain overlaps with that of a previous definition of the same link name, the newer definition overrides the previous one for the overlapped domain. This is discussed further in the section describing 'write_function (read)'.

arg3 must be either a node of type 'builtin' or 'rule'. If arg3 is 'builtin link', this is an attempt to define a link for writing; therefore, arg1 must be a list with exactly one element which is a 'type' node.

If arg3 is neither of type 'builtin' nor type 'rule', error condition 'dwf1' is signalled. If arg1 is neither 'flag general' nor of type 'cell', error condition 'dwf2' is signalled. If an element of the list given as arg1 is neither 'flag any' nor of type 'type', error condition 'dwf3' is signalled. If a connector node of the list given as arg1 includes a node other than a 'cell', error

condition 'dwf4' is signalled.

If arg 3 is 'builtin link', then this is an attempt to define a link for writing. If arg1 is not a list of one element, error condition 'dwf5' is signalled. If another link cannot be defined for the given type because that would exceed the maximum number of links given in the hints, error condition 'dwf6' is signalled.

In defining any writing function, if the system cannot accommodate that definition (according to hint variable 'defns_size'), error condition 'dwf7' is signalled. If the number of elements of the list given as arg1 exceeds the maximum number of arguments of a function (according to hint variable 'function_arguments'), then error condition 'dwf8' is signalled.

The error messages for this function follow.

- dwf1: The head pointer of a write-call on the builtin
 "write_function" is " ... ", which is neither of
 type "builtin" nor of type "rule".
- dwf2: The first argument of a write-call on the builtin
 "write-function" is " ... ", which is neither the
 node "flag general" nor a node of type "cell".
- dwf3: The first argument of a write-call on the builtin
 "write_function" is a list beginning with " ... "
 which includes a node of type "cell" whose "value"
 link points to " ... ", which is neither the node
 "flag any" nor a node of type "type".
- dwf4: The first argument of a write-call on the builtin
 "write_function" is a list beginning with " ... "
 which includes a node of type "cell" whose "next"
 link points to " ... " , which is not of type "cell".

- dwf5: A write-call on the builtin "write_function" is an attempt to define the link " ... ", and the first argument is " ... ", which is not a list of one node of type "type".
- dwf6: A write-call on the builtin "write_function" is an attempt to define the link " ... " on nodes of type " ... ", and another link cannot be defined for this type.
- dwf7: A write-call on the builtin "write_function" is causing an attempt to make a new definition, and that would exceed capacity.
- dwf8: The first argument of a write-call on the builtin "write_function" is a list beginning with " ... " whose length exceeds the maximum number of arguments allowed for a function definition!

read_function (read)

This function is called with two tail arguments (arg1 and arg2) and one head result. It is used to determine the reading function which is defined to be called for a given link name, arg2, and with a given sequence of types of tail arguments. This sequence of types is represented as arg1 by a (possibly empty) list where nodes of type 'cell' are the connectors. 'cell's are forwardly-linked by 'next' links, and 'cell end' terminates the list. Each other (if any) 'cell' node of the list has a 'value' link pointing at a list element which is a node of type 'type'. Each given list element represents one tail of the link whose definition is sought. If there is no definition for the given link name and sequence of types, the result is 'builtin error'. Otherwise, this function's result is either another 'builtin' node or a 'rule' node.

Since overlapping domains might have been given for a particular link name on various write calls on 'read_function', we include the following

complicated description. For the given number of elements (including zero) of the list presented as arg1 , previous definitions are scanned for that number of elements. The scan is performed starting with the most recent definition and then working backwards towards the oldest definition. If a definition included 'flag any', that argument position matches any given type. If a match is found during the scan, no further scanning is performed and a result is returned. If the scan did not result in any match, then those definitions are scanned which were made for any general arguments as indicated by a first tail argument of 'flag general' on a previous write call on 'read_function'. As before, general definitions are scanned newest to oldest. There are no built-in general definitions. If a match is not found among the general definitions, a result of 'builtin error' is returned.

This function is called by the AMBIT/G interpreter for every reading link it processes to determine the definition of that link. It is not expected to be used within "normal" AMBIT/G programs.

If arg1 is not of type 'cell', error condition 'grf1' is signalled. If an element of the list given as arg1 is not of type 'type', error condition 'grf2' is signalled. If a connector node of the list given as arg1 includes a node other than a 'cell', error condition 'grf3' is signalled. If the number of elements of the list given as arg1 exceeds the maximum number of arguments of a function (according to hint variable 'function_arguments'), then error condition 'grf4' is signalled.

The error messages for this function follow.

- grf1: The first argument of a read-call on the builtin
 "read_function" is " ... ", which is not of type "cell".
- grf2: The first argument of a read-call on the builtin
 "read_function" is a list beginning with " ... "
 which includes a node of type "cell" whose "value"
 link points to " ... " , which is not of type "type".
- grf3: The first argument of a read-call on the builtin
 "read_function" is a list beginning with " ... "

which includes a node of type "cell" whose "next" link points to " ... " , which is not of type "cell".

grf4: The first argument of a read-call on the builtin "read_function" is a list beginning with " ... " whose length exceeds the maximum number of arguments allowed for a function definition.

write_function (read)

This function is called with two tail arguments (arg1 and arg2) and one head result. It is used to determine the writing function which is defined to be called for a given link name, arg2, and with a given sequence of types of tail arguments. This sequence of types is represented as arg1 by a (possibly empty) list where nodes of type 'cell' are the connectors. 'cell's are forwardly-linked by 'next' links, and 'cell end' terminates the list. Each other (if any) 'cell' node of the list has a 'value' link pointing at a list element which is a node of type 'type'. Each given list element represents one tail of the link whose definition is sought. If there is no definition for the given link name and sequence of types, the result is 'builtin error'. Otherwise, this function's result is either another 'builtin' node or a 'rule' node.

Since overlapping domains might have been given for a particular link name on various write calls on 'write_function', we include the following complicated description. For the given number of elements (including zero) of the list presented as arg1, previous definitions are scanned for that number of elements. The scan is performed starting with the most recent definition and the working backwards towards the oldest definition. If a definition included 'flag any', that argument position matches any given type. If a match is found during the scan, no further scanning is performed and a result is returned. If the scan did not result in any match, then those definitions are scanned which were made for any general arguments as indicated by a first tail argument of 'flag general' on a previous write call on 'write_function'. As before, general definitions are scanned newest to oldest. There are no builtin general definitions. If a match is not found among the general definitions,

a result of 'builtin error' is returned.

This function is called by the AMBIT/G interpreter for every writing link it processes to determine the definition of that link. It is not expected to be used within "normal" AMBIT/G programs.

If arg1 is not of type 'cell', error condition 'gwfl' is signalled. If an element of the list given as arg1 is not of type 'type', error condition 'gwfl2' is signalled. If a connector node of the list given as arg1 includes a node other than a 'cell', error condition 'gwfl3' is signalled. If the number of elements of the list given as arg1 exceeds the maximum number of arguments of a function (according to hint variable 'function_arguments'), then error condition 'gwfl4' is signalled.

The error messages for this function follow.

- gwfl: The first argument of a read-call on the builtin
 "write-function" is " ... ", which is not of type "cell".
- gwfl2: The first argument of a read-call on the builtin
 "write-function" is a list beginning with " ... "
 which includes a node of type "cell" whose "value"
 link points to " ... ", which is not of type "type".
- gwfl3: The first argument of a read-call on the builtin
 "write_function" is a list beginning with " ... "
 which includes a node of type "cell" whose "next"
 link points to " ... ", which is not of type "cell".
- gwfl4: The first argument of a read-call on the builtin
 "write_function" is a list beginning with " ... "
 whose length exceeds the maximum number of
 arguments allowed for a function definition.

char (read)

This function is called with one tail argument (arg1) and one head result. It is used to read (or input) one ASCII character from either the input stream of the user's typewriter terminal or from the normal input file for the AMBIT/G run (e.g., 'foo.ambitg'). The result of this function is always a node of type 'char', and it is one of the 128 built-in nodes of that type. The design of this function may be extended in the future as other input requirements present themselves; but presently only these two devices are available. If arg1 is the node 'char D' (for disk or data) the character is read from the file. If, however, this is an attempt to read beyond the end of the file, error condition 'rc1' is signalled. Any other value of arg1 causes a character to be read from the terminal. Actually, the Multics System "hands over" one line at a time to AMBIT/G , so a user must type a new line (carriage return) before any characters of that line are obtainable. If this function is called when the user has not typed any input, it waits for the input indefinitely with the AMBIT/G System in a dormant state with respect to Multics.

The AMBIT/G implementation on Multics passes through whatever ASCII character it finds. The Multics convention is every line ends in only the single character ASCII 'LF' (Multics terminology is NL for new line); the AMBIT/G programmer should not expect to find the ASCII 'CR' in the normal case.

This function is called by the AMBIT/G loader with an argument of 'char D' to read the input file.

The error messages for this function follow.

rc1: A read-call on the builtin "char" with a first argument of "char D" is an attempt to read beyond the last character of the input file.

char (write)

This function is called with one tail argument (arg1) and one head argument (arg2). It is used to write (or output) one ASCII character to the output stream of the user's typewriter terminal. The design of this function may be extended in the future as other output requirements present themselves; but presently only the one device is available. The character to be written is given as arg2 and it must be one of the 128 built-in nodes of type 'char'; otherwise, error condition 'wcl' is signalled.

This function normally buffers a line of characters at a time and flushes that buffer only when a 'LF' is written. The AMBIT/G implementation on Multics passes through whatever ASCII character it finds. The Multics convention is every line ends in only the single ASCII 'LF' (Multics terminology is NL for new line); the AMBIT/G programmer should not normally write the ASCII 'CR'. The line buffer can hold up to 130 characters and is flushed when that buffer is full.

To permit interactive programs written in AMBIT/G which have questions and answers, this function may be called with arg1 as 'char F' to flush the character buffer, including the current character given as arg2. Any other value of arg1 causes normal buffering to occur.

The error messages for this function follow.

wcl: The head pointer of a write-call on the builtin
 "char" is " ... ", which is not a node of type
 "char" which represents an ASCII character.

add (read)

This function is called with two tail arguments (arg1 and arg2) and one head result. It is used to produce as result the 'integer' node representing the sum of the integer represented by the 'integer' node arg1 and the integer represented by the 'integer' node arg2.

If arg1 is not of type 'integer', error condition 'add1' is signalled. If arg2 is not of type 'integer', error condition 'add2' is signalled. If the sum of the two arguments is an integer which is outside of the range of existing AMBIT/G integers (according to hint variables 'smallest_integer' and 'largest_integer'), error condition 'add3' is signalled.

The error messages for this function follow.

- add1: The first argument of a call on the builtin
 "add" is " ... ", which is not of type "integer".
- add2: The second argument of a call on the builtin
 "add" is " ... ", which is not of type "integer".
- add3: A call on the builtin "add" produced a sum of
 " ... ", which is outside of the range of integers.

subtract (read)

This function is called with two tail arguments (arg1 and arg2) and one head result. It is used to produce as result the 'integer' node representing the difference of the integer represented by the 'integer' node arg1 minus the integer represented by the 'integer' node arg2.

If arg1 is not of type 'integer', error condition 'sub1' is signalled. If arg2 is not of type 'integer', error condition 'sub2' is signalled. If the difference of the two arguments is an integer which is outside of the range of existing AMBIT/G integers (according to hint variables 'smallest_integer'

and 'largest_integer'), error condition 'sub3' is signalled.

The error messages for this function follow.

- sub1: The first argument of a call on the builtin
 "subtract" is " ... ", which is not of type "integer".
- sub2: The second argument of a call on the builtin
 "subtract" is " ... ", which is not of type "integer".
- sub3: A call on the builtin "subtract" produced a difference
 of " ... ", which is outside of the range of integers.

multiply (read)

This function is called with two tail arguments (arg1 and arg2) and one head result. It is used to produce as result the 'integer' node representing the product of the integer represented by the 'integer' node arg1 times the integer represented by the 'integer' node arg2.

If arg1 is not of type 'integer', error condition 'mul1' is signalled. If arg2 is not of type 'integer', error condition 'mul2' is signalled. If the product of the two arguments is an integer which is outside of the range of existing AMBIT/G integers (according to hint variables 'smallest_integer' and 'largest_integer'), error condition 'mul3' is signalled.

The error messages for this function follow.

- mul1: The first argument of a call on the builtin
 "multiply" is " ... ", which is not of type "integer".
- mul2: The second argument of a call on the builtin
 "multiply" is " ... ", which is not of type "integer".
- mul3: A call on the builtin "multiply" produced a product
 of " ... ", which is outside of the range of integers.

divide (read)

This function is called with two tail arguments (arg1 and arg2) and two head results (res1 and res2). It is used to produce: as result res1 the 'integer' node representing the quotient of the integer represented by the 'integer' node arg1 divided by the integer represented by the 'integer' node arg2; and as result res2 the 'integer' node representing the remainder of that division.

If arg1 is not of type 'integer', error condition 'div1' is signalled. If arg2 is not of type 'integer', error condition 'div2' is signalled. If arg2 is 'integer 0', error condition 'div3' is signalled. If the quotient of the division is an integer which is outside of the range of existing AMBIT/G integers (according to hint variables 'smallest_integer' and 'largest_integer'), error condition 'div4' is signalled. If the remainder of the division is an integer which is outside of the range of existing AMBIT/G integers, error condition 'div5' is signalled.

The error messages for this function follow.

- div1: The first argument of a call on the builtin
 "divide" is " ... ", which is not of type "integer".
- div2: The second argument of a call on the builtin
 "divide" is " ... ", which is not of type "integer".
- div3: A call on the builtin "divide" is an attempt to
 divide by zero.
- div4: A call on the builtin "divide" produced a quotient
 of " ... ", which is outside of the range of integers.
- div5: A call on the builtin "divide" produced a remainder
 of " ... ", which is outside of the range of integers.

sign (read)

This function is called with one tail argument (arg1) and one head result. It is used to produce as result either:

- a) 'integer -1' if arg1 is an 'integer' node representing an integer less than zero; or
- b) 'integer 0' if arg1 is 'integer 0'; or
- c) 'integer 1' if arg1 is an 'integer' node representing an integer greater than zero.

If arg1 is not of type 'integer', error condition 'sgn1' is signalled. If the integer result (-1, 0, or 1) is outside of the range of existing AMBIT/G integers (according to hint variables 'smallest_integer' and 'largest_integer'), error condition 'sgn2' is signalled.

The error messages for this function follow.

sgn1: The argument of a call on the builtin "sign"
 is " ... ", which is not of type integer".

sgn2: A call on the builtin "sign" produced a result
 of " ... ", which is outside the range of integers.

A SAMPLE ERROR

The following page is terminal output of an AMBIT/G run on Multics which causes an error condition (drfl). Following the listing of the run is a listing of the program which caused the error. The arrows added to the output indicate lines typed in by the user.

→ hmu

Multics 13.1, load 8.0/41.0; 7 users

r 1443 .147 0+2

→ ambitg foo
AMBIT/G

AMBIT/G Error: The head pointer of a write-call on the builtin
"read_function" is "cell x", which is neither of type
"builtin" nor of type "rule".

This error occurred while interpreting the rule "rule foo".

The interpreter was processing the link "(cell x, cell y) read_function! (cell x)".

r 1444 16.529 0+125

→ pr foo.ambitg

foo.ambitg 12/27/70 1444.7 est Sun

-page- hello to you

-rule- r1 foo

a1 cell x

b1 cell y

(a1, b1) read_function! (a1)

-endrule-

-start- foo

CHAPTER 7

THE DEBUGGING FACILITY

This chapter describes 'AMBIT/G DEBUG', henceforth called 'agd', which is an interactive debugging aid used in conjunction with the AMBIT/G Programming System implemented on Multics. This debugger gives the user the facility to examine the AMBIT/G Data Graph symbolically in a variety of ways and to alter links in the Data Graph on an individual basis.

The user invokes the debugger by typing the following command line to the console command monitor:

agd

The debugger should only be invoked any time after AMBIT/G data has been initialized (or restored). AMBIT/G initialization is complete after the second new line (carriage return) is issued by the AMBIT/G System when it is first started. If 'agd' does not detect any existing AMBIT/G data when it is started, it will type an informative error message and terminate. Otherwise, it issues a new line (carriage return) and waits for user input.

'agd' is used interactively at a typewriter terminal where the user types requests, and the debugger issues typed responses to those requests. It never takes the initiative of requesting something from the user. 'agd' continues to interpret the user's requests until it terminates as a result of the user's typing either:

- a) a '/'q' command to quit, or
- b) a '/'S' command to signal a system status save, or
- c) a '/'R' command to resume AMBIT/G interpretation.

LEXICAL CONVENTIONS

SPACES are treated as characters and are not normally optional in 'agd' . For example, a full node name must have at most one SPACE separating the type-name from the subname.

Since the names of AMBIT/G nodes may consist of any printing characters, the following conventions apply to all user-typed input to 'agd'.

The dollar sign '\$' serves as a protection character for whatever character immediately follows it. Unless preceded by a protective dollar sign, the following characters have special meaning:

<u>Character</u>	<u>Meaning</u>
carriage return	statement terminator
;	statement terminator
,	separator
/	separator and command indicator
&	internal name indicator
\$	protect the next character

When a carriage return is protected (because a line ended in an unprotected '\$'), both the '\$' and carriage return are ignored. Thus, this combination can be used to input a statement on any number of typed lines.

The use of an unprotected '\$' immediately preceding any of the other five special characters

; , / & \$

causes that character to be interpreted literally as a character in a name. Thus, a user must type

ab\$&\$cd

when referring to the node name 'ab&\$cd'.

When an unprotected '\$' immediately precedes any character other than the special ones, that '\$' is ignored.

The above conventions for user-typed input to 'agd' insure that a user may make unambiguous requests. These conventions only apply to input, but 'agd' output may be ambiguous in some rare cases. In these cases it should be possible for a user to type one or two requests whose responses will eliminate the ambiguity.

STATEMENTS

A statement is an individual request which a user types to 'agd'. Usually a user types a statement immediately followed by a carriage return. This causes 'agd' to scan the statement and make some response. Note, however, that 'agd' makes a null response to the null statement.

More than one statement may be included on a typed line by using an unprotected ';' as a statement terminator.

When 'agd' detects an error and responds to a statement without having scanned to the end of the statement, it aborts the scan for the remainder of the typed line and issues an extra carriage return to indicate this condition to the user.

Except for the null statement, a statement is either a command, or it is a request to examine part of the AMBIT/G Data Graph. Examination may result in a typeout of the origin, name, and destination of a particular link; or it may produce such a typeout for each link emanating from a particular node.

If 'agd' detects an error in a statement, it attempts to respond with an informative error diagnostic. There are so many possible diagnostics which may be issued that this writeup will not include a list of them. The sample session at the end of the chapter includes examples of these diagnostics.

Before presenting the various forms of statements, some definitions and conventions will be given which apply to many of the statements.

Node Names

AMBIT/G Data consists of nodes and links. Each node belongs to a class, where each member of that class has the same links defined. This membership is described as the type of a node. The type of any given node never changes, and contributes directly to the node's name.

The type or type-name is that character string which, together with a node's subname, makes up the full name of a node. However, some nodes have no subname (those located with a null name); such a node is unnamed. A node with a non-null subname is a named node.

Thus, each node in AMBIT/G Data has a type-name which consists of one or more printable ASCII characters (not including SPACE, TAB, etc.). Each named node also has a subname which consists of one or more printable ASCII characters (not including SPACE, TAB, etc.). The subnames of the nodes of a given type must be unique. The canonical graphical representation of a named node is a rectangular box with the type-name positioned above the subname within the box. In 'agd', a node name is typed as the type-name followed by one SPACE followed by the subname.

Each type in AMBIT/G data has an associated node of type 'type' with a subname corresponding to the associated type. For example, if 'integer 5' is a node name, its type or type-name is 'integer' and its subname is '5'. Furthermore, there must also be a node in the data whose name is 'type integer'. This implies there must be a node in the data whose name is 'type type', and this sequence of implications stops here.

Internal Subnames

For the purposes of using 'agd' to examine AMBIT/G Data which includes unnamed nodes, the convention has been established to associate an internal subname with every node. An internal subname begins with an unprotected '&', and that is followed directly by an unsigned or negative decimal integer. The integer part of an internal subname of a given node conveniently coincides with the integer representing that node in the PL/I representation of AMBIT/G Data in the Multics Implementation; this is particularly valuable for debugging the AMBIT/G System in conjunction with the Multics 'debug' command.

Note that the integer part of an internal subname may be any negative integer greater than some lower bound which depends upon the particular AMBIT/G run. There is a similar upper bound for those internal subnames whose integer parts are positive; however, within the positive range not all integers correspond to nodes. A node with n links ($n \geq 0$) will "use" n consecutive integers. A type node "uses" four consecutive integers.

A node with an internal subname whose integer part is negative is a terminal node; i.e., it has no links.

Typing Node Names

When 'agd' types out a node name, it types a type-name followed by a SPACE followed by a subname. For a named node, the subname typed is the subname of that node; and for an unnamed node, the subname typed is the internal subname of that node.

When a user wishes to type in a reference to a node by its node name he has some choices: the user may type either two names separated by one SPACE, or just one name. If he types two names, the first is taken as a type-name, and the second as a subname. 'agd' will allow the type-name, in this case, to be the internal subname of a 'type' node. The second name typed by a user may be either the subname of a node or an internal subname; either form is acceptable for a named node, but the former option could not be used for an unnamed node.

If the user types a node name consisting of just one name, that name is taken as a subname. 'agd' will accept that reference if the typed subname is the subname of only one node in the data. This will always be the case when an internal subname is typed.

Therefore, when a user refers to a node, he may type only its internal subname. If he precedes that by a type-name, 'agd' checks for consistency. On type out, 'agd' will not eliminate the type-name in a node name.

Link-Names

A link in AMBIT/G Data is basically a triplet of node names, where the elements are: origin, name, and destination. Although the 'name' of a link is just another node, a convention has been established for typing out and typing in link-names which is different from the conventions for typing node names.

When 'agd' types out a link-name it omits typing the type-name and separating SPACE when the type-name is "link"; otherwise, the node name for that link is typed. Since many names of links are of type 'link', this convention reduces type-out time.

This convention also applies to a user-typed link-name. Thus, if only one name is typed as a link-name, and if it is not an internal subname, then 'agd' assumes the user had also typed a type-name of "link". If the user types just an internal subname, the node which is uniquely named by that internal subname is taken as the link-name reference.

Link Type-Out

When 'agd' types out one or more links it first types out the node name of the origin of the link(s) followed by a colon. Then on separate lines, it types each link indented by one TAB. The link-name is typed followed by a '/' as separator. This is normally followed by the node name of the destination of the link. One special convention has been established to conserve type-out time and aid to readability: if the destination is the built-in undefined node, 'undef undef', the type out of that node name is omitted.

There is no similar convention for typing in.

STATEMENT FORMS

The most common 'agd' request is for the examination of a node or a link. The following syntactic forms may be used:

<u>Syntactic Form</u>	<u>Use</u>
<u>node-name</u>	to examine a node
<u>node-name</u> { / <u>link-name</u> } /	to examine a node
<u>node-name</u> { / <u>link-name</u> } / <u>link-name</u>	to examine a link

A matching pair of curly braces indicates they enclose a construct which may be repeated any number of times, including zero. The underlined strings are meta-variables (non-terminals); their definitions have already been given.

The only SPACES allowed in these statements are those which separate a type-name from a subname. A ',' may optionally be used in place of a '/' in these statements. Examination of a node means that 'agd' types out all links which emanate from the specified node. If just one node name is given, as in the first form above, that name refers to the specified node. A statement of the second form, however, permits the user to type a list of link-names to specify a "walk" through the AMBIT/G Data Graph to reach the specified node. Each given link-name is interpreted as a step out of the current node via that link to the node at the destination of that link; that destination node then becomes the current node.

A statement of the third form is used to examine just one link of a specified node. 'agd' interprets such a statement by first determining the origin in the same manner as it does for a statement of the second form. Then, the final link-name given determines the one link which 'agd' will type out.

As 'agd' scans a statement, interpreting a "walk", it checks for any error conditions and will report immediately on any error without continuing its scan.

Commands

The remaining statement forms are commands which begin with a '/' followed by a letter. The following commands are recognized by 'agd':

<u>Syntactic Form</u>	<u>Use</u>
/t <u>type-name</u>	type out <u>type</u> statistics
/l <u>type-name</u>	<u>list</u> all <u>links</u> of the type
/i <u>node-name</u>	type <u>internal</u> subname
/s <u>node-name/link-name/node-name</u>	<u>set</u> a link (same as /w)
/w <u>node-name/link-name/node-name</u>	<u>write</u> a link (same as /s)
/S	<u>Signal System Status Save</u> and <u>resume execution</u>
/R	<u>resume</u> AMBIT/G interpretation
/q	<u>quit</u>

The '/S', '/R', and '/q' commands must be given as just two characters. The other commands may have any number (including 0) of SPACES following the first two characters. The argument of a '/t' or '/l' command cannot be an internal subname.

The '/t' command causes 'agd' to type out the current and maximum numbers of links and nodes for the given type and also the internal subnames spanned by the current number of nodes.

The '/i' command causes 'agd' to type out the node name of the given node followed by a '=' followed by the internal subname of that node.

The '/s' or '/w' command alters the AMBIT/G Data and types out both the old and new values of the link.

The '/S' and '/R' commands provide user control of the AMBIT/G System which are not directly related to debugging. Their use is described elsewhere in the report.

The '/q' causes 'agd' to terminate and return to its caller.

A SAMPLE SESSION

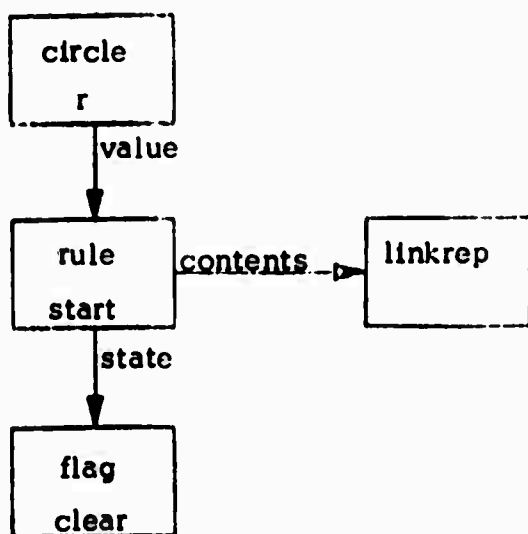
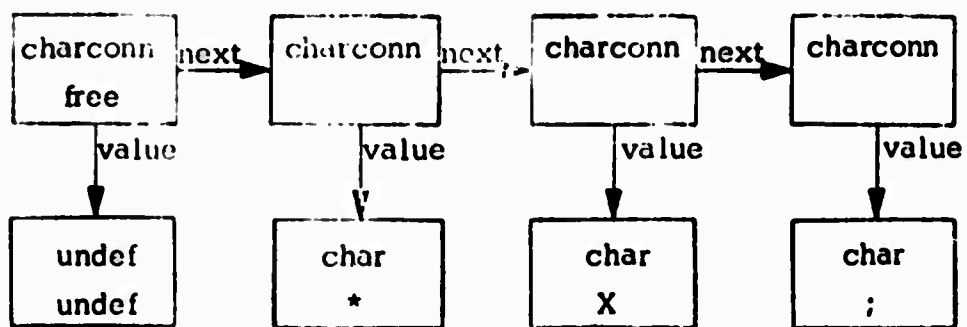
This description of 'agd' concludes with a sample session of using the debugger on a limited part of an AMBIT/G Data Graph. The listing included is a real session without any simulated typing. For convenience in the references to the listing, it has been annotated to the extent that every line typed in by the user has an associated integer; all other lines are typed out by the system.

On the next page is a portion of the AMBIT/G Data Graph which is referenced in the sample session. The Data Graph referenced also includes all initial (or built-in) data.

Following that is a listing of the session, and that is finally followed by an explanation of the session.

Other 'agd' sessions are included among the examples in Volume II.

A Portion of an AMBIT/G Data Graph



Listing of the Session

```
1  add
2  charconn free
   charconn free:
       next/charconn &5902
       value/
3  charconn &5902
   charconn &5902:
       next/charconn &5904
       value/char *
4  &5904
   charconn &5904:
       next/charconn &5906
       value/char X
5  &5905
   INTERNAL SUBNAME "&5905" DOES NOT HAVE A NODE
6  charconn free/next/next/next/
   charconn &5906:
       next/charconn &5908
       value/char ;
7  char ;
   THE NODE NAME "char" HAS BAD SYNTAX WRT SPACES
8  char ;
   char :: NO LINKS
9  free;/l char;/lcharconn
   "free" IS NOT UNIQUE: cell, pipe, circle, charconn
   NODES OF TYPE "char" HAVE NO LINKS
   NODES OF TYPE "charconn" HAVE LINKS:
       next
       value
10 /t charconn;/l charconn;/s charconn;/q charconn
    TYPE "charconn" HAS 2/2 LINKS AND 2003/2003 NODES: &5908 TO &5902
    type charconn = &1052
    INCOMPLETE "/s" COMMAND
11 /q charconn
    USE "/q" ALONE TO QUIT
12 charconn &1052
    INTERNAL SUBNAME "&1052" DOES NOT HAVE A NODE OF TYPE "charconn"
13 type &1052
    INTERNAL SUBNAME "&1.052" HAS BAD SYNTAX
14 type &1052
    type charconn: NO LINKS
15 foo;noo noo
    NO NODE HAS SUBNAME "foo"
    THE TYPE NODE "noo" IS UNDEFINED
16 type poo
    "poo" IS NOT A SUBNAME OF A NODE OF TYPE "type"
17 /s charconn &5906/value/char SP
    charconn &5908:
        value/char ;
    BECOMES value/char SP
```

```

18  circle r;circle r/link value
    circle r:
        value/rule start
    circle r:
        value/rule start
19  circle r,value,state
    rule start:
        state/flag clear
20  circle r/value/next
    NODE "rule start" DOES NOT HAVE LINK "link next"
21  circle r/value/state/
    flag clear: NO LINKS
22  circle r/value/state/state
    NODE "flag clear" DOES NOT HAVE LINK "link state"; IT HAS NONE
23  circle r/foo
    "foo" IS NOT A SUBNAME OF A NODE OF TYPE "link"
24  /q
    r 1822 9.807 256+651

```

Explanation of the Session

1. "agd" is the command the user typed to the Multics console command monitor which caused 'agd' to be invoked. Note the blank line issued by 'agd' to indicate it is ready to receive requests.
2. This is a request to examine a node. Note that 'agd' retyped "charconn free;" and then typed the two links emanating from that node. Since the names of both links are of type 'link', "link" is not typed in the link-names. Also, since the destination of the 'value' link is 'undef undef' nothing was typed there. The destination of the 'next' link is an unnamed node, and therefore an internal subname is typed.
3. This is a request to examine a node using a node name composed of a type-name and internal subname.
4. Just the internal subname was typed by the user to examine a node.
5. Just an internal subname was typed, but it does not name a node.
6. This is an examination of a node by specifying a "walk". Note the syntax of the destination of the 'value' link.
7. The user then tried to examine the node "char;", but the ';' acted as a statement terminator.
8. This is a successful examination of that node, demonstrating the use of the protection character.
9. Three statements are included on one typed line. First, there is an attempt to examine a node by typing only a subname, but that subname is not unique; note that 'cell free', 'pipe free', and 'circle free' are built-in nodes. Then there are two examples of the '/l' command.

10. This line includes four statements. The first is an example of the '/t' command, and the second is an example of the '/l' command. 'agd' detected an error in the third statement, and terminated its scan, which is indicated by its issuing the blank line.
11. The user now re-enters (for demonstration only) the command which was not previously scanned, but 'agd' detects an error and again terminates its scan in the middle of the line.
12. The user has typed a node name, where the type-name is not consistent with the type of the typed internal subname.
13. The typed input appears to be correct, but due to a terminal or communications problem an extra period was accidentally entered. (Incidentally, this was a spontaneous error which the author did not expect.)
14. This is an example of the examination of a 'type' node.
15. Two silly commands are typed by the user, and 'agd' types the two error diagnostics.
16. This is another example of an error diagnostic caused by an incorrect node name.
17. This is an example of setting a link.
18. This line includes both a node examination and a link examination which both produce identical results. The second statement includes a link-name which was typed with a type-name.
19. This is a link examination with a "walk", and also demonstrates the use of commas as separators.

20. This is a node examination which includes a "walk" ending in a non-existent link.
21. This is a successful node examination.
22. This is a link examination which includes a "walk" ending in a non-existent link on a terminal node.
23. This is an attempt to examine a link where the link-name is not even a node.
24. This is a successful end of the session. The final 'ready' message line was typed by the Multics system. It indicates that this sample session used 9.867 seconds of CPU time.

CHAPTER 8

THE IMPLEMENTATION

This chapter begins with credits and acknowledgements of those responsible for the implementation of the AMBIT/G System on Multics. The remainder of the chapter discusses various details of the implementation which serve as notes for a maintainer of the system and as a guide for using the system. Since at the time of this writing there is essentially no user community for this experimental implementation, we have not produced a polished user manual. The implementation was written in Multics PL/I and AMBIT/G (bootstrapped by hand-translation into PL/I).

CREDITS AND ACKNOWLEDGEMENTS

The AMBIT/G System described in this report has been implemented on the Multics (for Multiplexed Information and Computing Service) System at M.I.T., which is a general purpose time-shared computer utility implemented on a "General Electric" 645 computer system.

This development was carried out using a semi-portable Datel 30 type-writer terminal especially modified for Multics use. This terminal was used in the Wakefield office of Applied Data Research and in a home of one of the implementors. We regularly obtained line printer listings by ordering them (using the terminal) and then picking them up at M.I.T.

Michael S. Wolfberg was responsible for the implementation effort as a whole. Carlos Christensen, D. Austin Henderson, Michael J. Fischer, and M.S. Wolfberg worked together to produce the design of the language to be implemented and to resolve some difficult problems of the implementation. M.J. Fischer wrote the AMBIT/G program for the interpreter which appears in the third volume and wrote several of the example programs in the second volume. M.S. Wolfberg wrote the loader, the underlying foundation described in the current chapter, the debugging subsystem, and the remaining example programs with the assistance of Maynie Ho.

We thank Prof. J.C. R. Licklider, Director of Project MAC, M.I.T. who made important initial steps in arranging for our access to Multics. We also thank the staff of M.I.T. Information Processing Services and the M.I.T. Information Processing Center who administer Multics, especially Thomas Van Vleck (of I.P.C.) and Jerrold M. Grochow (Assistant to the Director of I.P.S.) for their helpful assistance and advice so courteously given.

AN INTERNAL VIEW OF THE MULTICS AMBIT/G SYSTEM

An AMBIT/G program exists on Multics as a pair of source files; for example, the program 'foo' would be represented by 'foo.hints' and 'foo.ambitg'. The file with secondary name 'hints' is usually rather small; it includes some information which causes a particular initialization of the AMBIT/G machine. Since this information is considered to be outside of the definition of the AMBIT/G language we call its contents "hint information" or just "hints". The syntax and semantics of the hints is given elsewhere.

The source file with secondary name 'ambitg' includes a string encodement of the AMBIT/G program in a format acceptable to the AMBIT/G loader.

The AMBIT/G System exists as eight executable segments: 'pribin', 'interpreter', 'loader', 'ambitg_error', 'intldr_error', 'agd', 'agsave', and 'agrestore'. At a minimum, the first three segments must be available for execution, and the next three segments should be available in case any error conditions arise. The remaining two segments are used only for saving and restoring the status of an AMBIT/G program. Although 'agd' is required for handling errors, it alone is required for performing interactive symbolic debugging of AMBIT/G data.

The AMBIT/G System can be invoked to run a program by the user's typing the console command "ambitg" followed by a space followed by the name of the program. Alternatively, the 'ambitg' procedure can be called from within a PL/I program with one character string argument as the name of the program. The 'ambitg' procedure is part of the 'pribin' bound archive, and the 'pribin' segment has 'ambitg' as an additional name.

Let us assume the user issued the console command "ambitg foo". We shall now trace through the initialization process.

Initialization

First, the system save switch is cleared. Then an attempt is made to "open" the source file 'foo.ambitg' in the current working directory; if there is an error an error message is typed and execution aborts. The length of this file is determined as a character count, and the file input counter is initialized to zero.

To overcome the inability of Multics PL/I to read a file whose name has been determined by a computation, a fixed name, 'ambitg_hints', is temporarily added to the hint file, 'foo.hints'. PL/I input statements are used to read the hint file. If the fixed name cannot be added to the hint file for any reason, an indicative error message is typed concerning the "opening" of the hint file and execution aborts.

Three segments are then created in the current process directory which are to contain nearly all data specific to this AMBIT/G run:

<u>Name</u>	<u>Use</u>
nodes_segment	all AMBIT/G 'type' nodes and non-terminal nodes
names_segment	the symbol table of all named nodes
defns_segment	all function definitions

If one of these segments is found to be already known to the process, it is used and initially truncated. Any error condition causes an indicative error message to be typed and execution aborts.

If all has gone well up to this point "AMBIT/G" is typed on the terminal as positive feedback to the user.

Default values are assigned to the hint variables and then a 'get file data' PL/I statement is used to read any overriding values for the hint variables. Consistency checks are performed for hint variables to meet various conditions. Any inconsistency causes an indicative error message to be typed and execution aborts.

After passing all consistency checks, the initializer reads any number of terminal node hints from the hint file. Each such hint consists of a non-null string and a non-negative integer. The end of these terminal node hints is signalled by a null string. The given hints are merged with built-in terminal node hints.

Then any number of non-terminal node hints are read from the hint file. Each such hint consists of a non-null string and an integer greater than zero and a non-negative integer. The end of these non-terminal node hints is signalled by a null string. The given hints are merged with built-in non-terminal node hints.

During the reading of hints various conditions may cause an error message to be typed and execution to be aborted. Otherwise, the hint file is "closed" by removing the temporary name 'ambitg_hints' from it. This too is prone to an error condition leading to execution aborting.

If all is well, the user is given more feedback by a blank line being typed.

At this point, the 'nodes_segment' is arranged, and all built-in nodes are created and their names are made known in the 'names_segment'. Then the 'defs_segment' is initialized to include definitions of all built-in functions. All built-in links are defined, and finally an initial data graph is created including various built-in rules in a 'clear' state.

A second blank line is typed to indicate to the user the end of initialization. The AMBIT/G interpreter is then invoked to begin interpretation at 'rule start', which is one of the built-in rules.

Interpretation

The initializer creates an AMBIT/G machine and data graph and finally calls the interpreter as a subroutine with one argument. In this case, that argument is 'rule start'. Thus the interpreter carries out an interpretation cycle on 'rule start'. Since that rule is in a 'clear' state, it is first compiled and then is interpreted. The contents of that rule causes a function call on the

AMBIT/G loader. The loader reads its input one character at a time from the source file 'foo.ambitg' by calling on the primitive 'read_char'. When a '-start-' statement is processed by the loader, it returns control to the interpreter which finishes its interpretation of 'rule start'. That interpretation causes the 'success' exit of 'rule start' to be the starting rule of 'foo'. So the interpreter will then carry out an interpretation cycle on that rule.

The interpreter then continues to execute the program 'foo' by interpreting its rules. It is possible that 'foo' includes rules which cause further reading of 'foo.ambitg' by either calling the loader again or by calling the read version of the builtin 'char'.

Interpretation continues indefinitely until one of the following occurs:

- a) The interpreter detects an error condition; this causes an error message to be typed and execution aborts.
- b) The interpreter calls a primitive which detects an error condition; this causes an error message to be typed and execution aborts.
- c) The interpreter interprets 'rule stop' at the top level (i.e., 'ptr ret' points to 'rule stop'); this causes the interpreter to return control to its caller. If the interpreter had been called by the initializer, the initializer then returns control to its caller which was probably the console command monitor. This is the standard ending of a complete AMBIT/G run. The AMBIT/G data is preserved in the process directory.
- d) The interpreter detects the system save switch is set when it begins a new cycle at its own 'rule start'; this causes the interpreter to call the 'agsave' procedure which saves the status of the AMBIT/G System in the current working directory. The 'agsave' procedure returns to its caller with a cleared system save switch if the user answers "yes" to

the question "CONTINUE EXECUTION?" typed on the terminal; this causes the interpreter to begin that new cycle from which it was interrupted. If the user had answered "no" to the question posed by the 'agsave' procedure, then it does not return to the interpreter, but instead calls the console command monitor. Thus this could be considered as the end of a partial AMBIT/G run. Note that the AMBIT/G data is preserved in the process directory in addition to the saved data in the current working directory. The rule which the interpreter was about to interpret is saved as the node pointed to by 'circle r'.

The system save switch is originally cleared as the first action performed by the initializer. The user can cause that switch to be set by temporarily interrupting his job and invoking the '/S' command of 'agd'.

- e) The user aborts the running of the job by depressing the "QUIT" button on his terminal and does not resume that job; this could be considered as the end of a partial AMBIT/G run. Note that AMBIT/G data is preserved in the process directory.
- f) The system crashes and automatically logs out the user; this destroys the process directory and thus all AMBIT/G data. If the user had caused any system saves to be done, the working directory at the time of that save contains the saved data. Thus if a user has invested in a rather long or costly run, he is advised to save the AMBIT/G system status at appropriate intervals during the run.

We have described the way in which the interpreter is called as a procedure by the initializer after all initialization is complete. The one other call upon the interpreter in the current implementation is in 'agrestore', the restoration procedure. This call can be invoked by various methods, but it finally calls upon the interpreter with an argument of the node pointed to by 'circle r'.

Loading

The AMBIT/G loader is a primitive of the system as are 'read_link' and 'read_type'. However, it differs from all other primitives in the method used to define and implement it. The loader was written in AMBIT/G and then hand-translated to PL/I in the same way the interpreter was implemented. Even though the loader is a primitive, it makes use of calling upon many other primitives, but then so do some of the other primitives.

If the loader is given a large source file to load it may require a significant amount of time. Thus if a user causes the setting of the system save switch during the interpretation of a rule which includes a call on the loader, he may have to wait a long time before the interpreter again begins another cycle when it causes the saving to occur.

The loader has no argument, but it is called as a function by the interpreter. The result is a 'rule' node specified in the 'start' statement which ends the loading process.

Although the loader detects and reports several error conditions resulting from an improper source file, there are some errors which show up in a primitive on which the loader calls. Unfortunately the error message in this case often does not include enough information for the user to determine the cause of the error. In this case, the user can invoke 'agd' and determine the current statement number by observing the destination of the 'value' link of 'circle page'; the name of the current page is a list of characters beginning at the destination of the 'value' link of 'circle line'. These two pieces of information are typed as part of any error message which the loader itself caused to be typed.

Errors

When the interpreter or loader detects an error condition it makes a procedure call on an error procedure in the object segment 'intldr_error'. Entry point names for errors detected by the interpreter are 'intldr_error\$intN' where N is a decimal integer. Similarly, loader errors use entry points names of the form 'intldr_error\$ldrN'. Depending on the type of error, some arguments are passed to the particular routine. Each error routine causes the

typing of an error message on the user's terminal. Arguments are converted to symbolic node names for more informative type-out by the error routine's calling 'agd\$get_name' which is a special entry point in the debugger.

Errors detected by the loader cause an additional line to be typed before the specific error message which indicates a statement number and the name of the current page being loaded. These are determined by following the 'value' link of 'circle page' to an integer and the 'value' link of 'circle line' to a list of characters.

Following the specific error message is a general dump of the status of the interpreter. First is the name of the rule being interpreted determined by following the 'value' link of 'circle r'. Then, if the 'value' link of 'circle l' points to a 'linkrep' node on the 'contents' list of the current rule, that 'linkrep' is dumped as completely as possible. Finally, if 'ptr ret' does not point to 'rule stop' a function call stack trace is performed until either 'rule stop' is encountered or a cycle is encountered or the list is found to include a node whose type is not 'rule'. The trace consists of typing sub-names of 'rule' nodes and perhaps ' ... ' to indicate a cycle, or a full node name if a node is found on the list whose type is not 'rule'.

After all of the typing, the error routine calls the console command monitor.

When a primitive other than the loader detects an error condition, it makes a procedure call on an error procedure in the object segment 'ambitg_error'. These various error routines operate just as those in 'intldr_error'.

Saving

The system save switch is a 'fixed binary external static' variable which is cleared at the beginning of AMBIT/G initialization. It can be set by the user's temporarily interrupting his job by depressing the "QUIT" button on his terminal and then invoking the '/S' command of 'agd'. Such an invocation may be carried out in one of two ways:

- a) The user may enter the 'agd' debugger by typing the console command "agd". Then he may use various features of the debugger. When he wishes to signal a system save and continue execution, he types the 'agd' command "/S".
- b) The user may type the console command "agd\$S", which is equivalent to typing "agd" and then "/S".

Either of these operations simply causes the system save switch to be set and then interrupted AMBIT/G execution to be resumed by calling the 'start' procedure of the Multics Standard Service System.

Actual saving will then occur when the interpreter is about to begin its next interpretation cycle. Before the interpreter's 'rule start' is executed, if the system save switch is set, a call is made on 'agsave'. The 'agsave' procedure saves the status of the AMBIT/G System in the current working directory as detailed later. After saving is complete, the procedure types a question to the user: "CONTINUE EXECUTION? ". If he answers "yes" the 'agsave' procedure clears the system save switch and returns to its caller; this causes the interpreter to proceed with its interpretation cycle which was interrupted. If the user answers "no", 'agsave' calls the console command monitor.

The method of saving just described preserves the interpretability of an AMBIT/G program and we expect it to be the common method for saving. However, the user is also permitted to invoke 'agsave' as a console command at any time. If saving is done at a time when AMBIT/G is in a strange state, later restoration and resumption of execution may not be reliable. Saving such as this is acceptable if the user expects only to look at the AMBIT/G data with 'agd' after a future restoration.

When 'agsave' is invoked as a console command it still asks the user whether execution should continue. In this case, the question is meaningless and either "yes" or "no" is acceptable since either answer ultimately causes control to proceed to the console command monitor. (The 'agsave' routine should be altered to eliminate the typing of the question in this case.)

We shall now describe some details of the actions of 'agsave'. It begins by typing an informative message on the terminal which includes the name of the program and the full name of the destination of the 'value' link of 'circle r'. For example:

SAVING AMBIT/G STATUS OF "foo" AT "rule x"

Next, fifteen external static variables which reflect system status are copied as an extension to the 'nodes_segment'. Then the three segments in the process directory are copied into the current working directory with the following naming convention (assuming 'foo' is the program name):

<u>Original</u>	<u>Saved Copy</u>
nodes_segment	foo.nodes.save
names_segment	foo.names.save
defs_segment	foo.defs.save

If any such files already existed in the current working directory they are overwritten. Thus it is possible to save only one version of a given AMBIT/G program in a given directory without using Multics file manipulation commands such as 'rename'.

It is often the case that the 'nodes_segment' is not densely utilized, and thus the Multics 'copy_seg_' procedure causes the typing of a verbose warning message which indicates the current length does not match the current block count. The user should ignore this warning.

If 'agsave' detects any error condition, it types an indicative error message and calls the console command monitor.

After the three segments are properly copied 'agsave' asks its question, which has been previously documented.

Restoration

Restoration is logically the reverse operation of saving, but since most external static variables can be reconstructed from the saved 'names_segment' they are not saved. Thus restoration takes significantly longer than saving.

The restoration procedure is 'agrestore' and it accepts a single argument which is the character string name of the program to be restored. The common method of invoking restoration is the user's typing a command to the console command monitor such as "agrestore foo". This would cause restoration of the saved AMBIT/G program 'foo' from the current working directory to the process directory; thus any existing AMBIT/G data would be overwritten. For a restoration to be properly completed the current working directory must contain:

- a) foo.nodes.save
- b) foo.names.save
- c) foo.defns.save
- d) foo.ambitg

Note that the hints file ('foo.hints') was read by the initializer, but it is never needed again. Even if the restored AMBIT/G program 'foo' does not read any more from 'foo.ambitg', that file (or any file with that name) must be in the current working directory.

The restoration process begins by typing an indicative message such as:

RESTORING AMBIT/G STATUS OF "foo"

Then the three saved segments are copied to the process directory. As in saving, it is likely that a verbose warning message will be issued due to a mismatch of the current length and current block count of the saved 'nodes_segment'. The user should ignore such a warning.

After the three segments are properly copied and the source file is initiated, the fifteen saved external static variables are initialized and then all other external static variables are initialized. Much of this is done by calls on the 'locate_prime' function, the alternate form of the 'locate' primitive.

If 'agrestore' detects any error condition, it types an indicative error message and calls the console command monitor.

Finally, restoration is complete and 'agrestore' then causes execution to resume by calling the interpreter with an argument of the node at the destination of the 'value' link of 'circle r'. Before calling the interpreter the system save switch is cleared and a message is typed to the user such as:

EXECUTION CONTINUES AT "rule x"

The restoration procedure has two alternate entry points which implement separately the distinct operations of restoration and resumption of execution. Invoking the command 'agrestore\$noex foo' causes restoration of the program 'foo', but when restoration is complete control returns to (its caller) the console command monitor. Invoking the command 'agrestore \$ resume' (with no argument) causes execution to resume as described previously. The '/R' command of 'agd' or the alternate entry 'agd \$ R' causes a call on 'agrestore \$ resume'. Such resumption of execution may be appropriate after certain error conditions leading to termination. In some cases, the user can use 'agd' to alter one or more links and then issue '/R' to try again. If a correctable error occurred during the interpretation of a rule, it may be necessary to alter the destination of the 'state' link of that rule.

Debugging

The interactive symbolic debugger 'agd' (for AMBIT/G Debugger) can be successfully invoked only after AMBIT/G data has been initialized or restored. When 'agd' begins, if it does not detect any existing AMBIT/G data, it types an informative error message and terminates. Otherwise, 'agd' issues a carriage return (new line) and waits for user input. There are no other errors which can cause termination.

The debugger makes use of several external static variables and the 'nodes_segment' and the 'names_segment'. It is self-contained and includes its own routines for reading types and links, etc.

Once it is properly started, it is terminated only by a '/q', '/S', or '/R' command. 'agd' is not written to account for the program interrupt condition.

One procedure is defined within the debugger for use by other parts of the AMBIT/G System: 'agd \$ get_name' determines a best symbolic node name as two character strings given the node address of a node. This procedure is called from 'intldr_error', 'ambitg_error', 'agsave', and 'agrestore'.

The debugger has two more alternate entry points which provide abbreviated forms of user commands:

- a) 'agd \$ S' can be issued as an abbreviation for the user's typing "agd" and then "/S". This command is described in the earlier section on saving.
- b) 'agd \$ R' can be issued as an abbreviation for the user's typing "agd" and then "/R". This command is described in the previous section on restoration.

Primitives

The loader has already been described since it is such a different sort of primitive. There are nine other PL/I procedures which include the 16 other primitives. These relationships are described in the section on the files of the implementation, but repeated here for convenience. An asterisk following an entry point name indicates it is called as a function.

<u>Read/Write</u>	<u>Name</u>	<u>Procedure</u>	<u>Entry Point</u>
read	type	read_type	read_type*
read	locate	locate	locate*
read	link (link')	read_link	read_link*
write	link (link')	read_link	write_link
read	name	read_name	read_name*
read	uchar	read_char	read_char*
write	char	write_char	write_char
read	read_function	get_read_function	get_read_function*
read	write_function	get_read_function	get_write_function*
write	read_function	define_read_function	define_read_function
write	write_function	define_read_function	define_write_function
read	add	agadd	agadd*
read	subtract	agadd	agsubtract*
read	multiply	agadd	agmultiply*
read	divide	agadd	agdivide
read	sign	agadd	agsign*

There are two additional entry points within these procedures used by the implementation:

'ravt' in procedure 'read_link' is a function used by the interpreter and loader to read a link and verify the type of the destination.

'locate_prime' in procedure 'locate' is a function used by the initialiser and 'agrestore'; it does the same as 'locate' except its second argument is a PL/I character string.

An internal description of each primitive is not included in this report; the user's description of each primitive should suffice. There are, however, some general statements we can make about implementation of the primitives. Many are called as functions, as indicated above, and the others are called as procedures. Arguments and results are always node addresses (indices into the virtual 'nodes_segment'. Note that since 'agdivide' has two results it is called as a procedure.

Most primitives perform extensive checking for error conditions and upon detecting an error make a call upon an error procedure within the object segment 'ambitq_error'.

The various primitives employ local variables for temporary storage and argument/ result passing. Otherwise they use various external static variables, the 'nodes_segment', 'names_segment', and 'defs_segment'.

Some primitives call upon others. The 'read_link' and 'write_link' primitives use their own local version of 'read_type' for efficiency. Otherwise, such use of primitives follows:

<u>Primitive (Entry Point)</u>	<u>Calls Upon (Entry Point)</u>
read_type	---
locate	read_type, read_link
read_link	---
write_link	---
read_name	read_type, read_link, write_link
read_char	---
write_char	---
get_read_function	read_type, read_link
get_write_function	read_type, read_link
define_read_function	read_type, read_link
define_write_function	read_type, read_link
agadd	---
agsubtract	---
agmultiply	---
agdivide	---
agsign	---

FILES OF THE MULTICS AMBIT/G SYSTEM

On January 1, 1971 the Multics AMBIT/G System was frozen in the following state. Directory ' >udd>Ambit/g>Wolfberg' contains the following executable segments which constitute the running system:

<u>Name</u>	<u>Use</u>
pribin	bound archive with initializer and all primitives except the loader; this has 19 additional names for proper linking
interpreter	AMBIT/G interpreter
loader	AMBIT/G loader
ambitg_error	handles all errors detected by primitives
intldr_error	handles errors detected by the interpreter and loader
agd	AMBIT/G DEBUG interactive debugger
agsave	handles saving of system status
agrestore	handles restoration of system status

In addition, this directory includes the following segments:

<u>Name</u>	<u>Use</u>
pribin.archive	archive of object segments of the initializer and all primitives except the loader
fact.defns.save fact.names.save fact.nodes.save	saved system status after execution of an AMBIT/ G program named 'fact'

The object segments 'interpreter' and 'loader' were created with the 'table' option of the PL/I compiler, which produces a symbol table useful for symbolic debugging. All other segments do not have symbol tables. When

more confidence is developed in the interpreter and loader they should be re-compiled without symbol tables.

Ideally, the interpreter and loader should be part of the bound archive; however, the binder currently has a limitation in the size of a table ('reference' array) which causes the inclusion of either 'interpreter' or 'loader' to exceed its capacity. When a new binder is released for Multics at least these two segments, and perhaps others, should be included in the bound archive. Since the detection of an error currently aborts execution, it is of little value to include 'ambitg_error' and 'intldr_error' in the bound archive.

When the PL/I compiler is altered to produce faster procedure calls all PL/I components of the system should be re-compiled.

The 'Wolfberg' directory has one sub-directory named 'source' which contains two archive segments:

<u>Name</u>	<u>Use</u>
primits.archive	archive of source segments of all primitives except the loader
sys.archive	archive of all other source segments for the system plus source segments of all AMBIT/G programs

There are 10 distinct PL/I procedures which cover all 17 primitives as follows:

<u>Read/Write</u>	<u>Name</u>	<u>Procedure</u>	<u>Entry Point</u>
read	type	read_type	read_type
read	locate	locate	locate
read	link (link')	read_link	read_link
write	link (link')	read_link	write_link
read	name	read_name	read_name
read	char	read_char	read_char
write	char	write_char	write_char
read	read_function	get_read_function	get_read_function

read	write_function	get_read_function	get_write_function
write	read_function	define_read_function	define_read_function
write	write_function	define_read_function	define_write_function
read	load	loader	loader
read	add	agadd	agadd
read	subtract	agadd	agsubtract
read	multiply	agadd	agmultiply
read	divide	agadd	agdivide
read	sign	agadd	agsign

There are two additional entry points within these procedures used by the implementation:

'revt' in procedure 'read_link' is used by the interpreter and loader to read a link and verify the type of the destination.

'locate_prime' in procedure 'locate' is used by the initializer and 'agrestore'; it does the same as 'locate' except its second argument is a PL/I character string.

Each of the 11 AMBIT/G programs in 'sys.archive' consists of two source segments. For the program 'octdec', for example, there are 'octdec.hints' and 'octdec.ambitg'. The following is a list of these programs.

<u>Name</u>	<u>Use</u>
msw2	check-out program in data loading form only
msw3	same as msw2, but using rule loading
msw5	demonstration of generality of function calls
reverse1	short program to reverse a list, method 1
reverse2	short program to reverse a list, method 2
reverse3	short program to reverse a list, method 3
octdec	interactive octal to decimal converter
quicksort	routine to sort a list of integers
mfgarb	Michael Fischer's garbage collector
lispgc	LISP garbage collector from Christensen's AMBIT/G paper
fact	factorial routine with recursion package

The following are directory listings and archive tables of contents of the frozen state of the Multics AMBIT/G System. Also included is a map for the 'pribin' bound segment. This indicates those segments in 'pribin.archive'.

'Wolfberg' Directory

Segments= 13, Records= 116.

rowa	1	fact.defns.save
rowa	2	fact.names.save
rowa	11	fact.nodes.save
re	6	agd
re	7	intldr_error
re	7	ambitg_error
re	16	interpreter
re	8	agrustore
re	2	agsave
re	19	loader
r wa	0	mailbox
r wa	19	pribin.archive
re	18	pribin
		ambitg
		define_write_function
		define_read_function
		ravt
		agsign
		agdivide
		agmultiply
		agsubtract
		agadd
		locate
		locate_prime
		read_type
		read_link
		write_link
		read_char
		get_read_function
		get_write_function
		read_name
		write_char

Directories= 1, Records= 1.

rowa	1	source
------	---	--------

Links= 0.

'Wolfberg>source' Directory

Segments= 2, Records= 60.

r wa 61 sys.archive
r wa 7 prints.archive

Directories= 0.

Links= 0.

sys.archive

name	updated	mode	modified	length
agsave.pll	12/21/70	r wa	12/21/70	39169
ambitg.pll	12/26/70	r wa	12/26/70	329274
msw3.hints	12/19/70	rewa	12/18/70	243
msw3.ambitg	12/20/70	r wa	12/20/70	8181
msw2.hints	12/19/70	r wa	12/18/70	243
msw2.ambitg	12/19/70	r wa	12/18/70	37044
msw5.hints	12/30/70	r wa	12/29/70	198
msw5.ambitg	12/30/70	r wa	12/30/70	18801
agrestore.pll	12/26/70	r wa	12/26/70	143937
mfgarb.hints	12/22/70	r wa	12/21/70	594
mfgarb.ambitg	12/22/70	r wa	12/21/70	14292
ambitg_error.pll	12/28/70	rewa	12/28/70	109002
lispgc.hints	12/27/70	r wa	12/23/70	837
lispgc.ambitg	12/31/70	r wa	12/31/70	37998
loader.pll	12/27/70	r wa	12/27/70	409215
fact.ambitg	12/31/70	r wa	12/31/70	93285
fact.hints	12/31/70	r wa	12/31/70	1008
intldr_error.pll	12/29/70	rewa	12/29/70	143469
interpreter.pll	12/29/70	r wa	12/29/70	390529
agd.pll	12/29/70	r wa	12/29/70	197955
reverse1.hints	12/30/70	r wa	12/30/70	306
reverse1.ambitg	12/30/70	r wa	12/30/70	4185
reverse2.hints	12/30/70	r wa	12/30/70	216
reverse2.ambitg	12/30/70	rewa	12/30/70	4563
reverse3.hints	12/30/70	rewa	12/30/70	210
reverse3.ambitg	12/30/70	r wa	12/30/70	10449
quicksort.hints	12/30/70	r wa	12/30/70	342
quicksort.ambitg	12/30/70	r wa	12/30/70	31284
octdec.hints	12/31/70	r wa	12/30/70	351
octdec.ambitg	12/31/70	r wa	12/31/70	30277

primits.archive

name	updated	mode	modified	length
read_type.pll	12/23/70	1408.0 r wa	12/23/70	1330.3
get_read_function.pll	12/23/70	1408.0 r wa	12/23/70	1350.9
read_char.pll	12/23/70	1408.0 r wa	12/23/70	1337.0
locate.pll	12/23/70	1408.0 r wa	12/23/70	1404.9
define_read_function.pll	12/23/70	1408.0 r wa	12/23/70	1402.2
write_char.pll	12/23/70	1408.5 r wa	12/23/70	1404.9
read_name.pll	12/23/70	1408.5 r wa	12/23/70	1408.0
agadd.pll	12/22/70	2307.6 r wa	12/22/70	2302.8
read_link.pll	12/23/70	1408.5 r wa	12/23/70	1408.5
				8217
				23130
				14562
				50805
				39681
				11025
				29493
				30726
				22374

pribin.map

12/27/70 1337.3 est Sun

Map for bound segment
Found in directory
Created

pribin
>user_dir_dir>Ambit/g>Wolfgang
12/26/70 1443.1 est Sat

Component segment

	Text		Linkage	
	Start	Length	Start	Length
read_type	0	220	0	46
read_link	220	645	46	64
read_char	1066	413	132	102
read_name	1502	1126	234	130
agadd	2630	1165	364	116
locate	4016	1651	502	156
write_char	5670	355	660	62
get_read_function	6246	747	742	126
define_read_function	7216	1360	1070	176
ambitg	10576	23712	1266	1442
Total		34510		2730

of internal references = 62.
of collapsed external links =78(1387 references to 346 links).
of collapsed entry links =9(10 references to 10 links).

The 'mailbox' segment in directory 'Wolfberg' is a receptacle for receiving mail from other users of Multics.

The additional names are required on the bound archive 'pribin' since procedures external to this segment make procedure calls using those names. In particular, the interpreter includes calls on every primitive. Eventually, when more of the AMBIT/G System can be commonly bound, most of these additional names can be removed.

The use of the sub-directory 'scurce' was established to make easier the updating of the procedures of the system. It is advantageous for PL/I compilations of primitives to be done in the 'source' directory since the 'pribin' bound archive includes names of all primitive object segments. Thus, for example, if 'read_type' were to be compiled in the 'Wolfberg' directory, then the object segment produced by the compiler would replace 'pribin' alias 'read_type'. Such use of the file system may lead to undesirable results if the maintainer is not exceedingly careful.

PL/I DATA FORMATS

We present here the arrangements of PL/I data used in the Multics implementation of the AMIBT/G System to represent the AMBIT/G data base including nodes, links, names, and function definitions. The reader who is interested in further details should consult PL/I listings of the system. This discussion will concentrate on the formats of the 'nodes_segment', 'names_segment', and 'defns_segment'. In addition to these, however, the implementation makes extensive use of 'external static' variables for global use throughout the procedures of the implementation. Only a small number of these reflect system status -- those which are saved during a system save.

nodes_segment

This segment is simply a long one-dimensional 'nodes' array with limits 'lb' to 'ub' declared as a PL/I structure:


```

dc1 1 nodes_segment based(nodes_ptr_local) aligned,
    2 nodes(lb:ub) fixed bin;

```

The 'nodes' array actually contains only representations of 'type' nodes and all non-terminal nodes; however, we usually think of a virtual 'nodes' array which also includes terminals. In this virtual array all nodes of a given type are represented contiguously. A node address is an index into the virtual nodes array. A terminal node occupies one index number below the actual array. The node address of the first 'type' node is the value of 'lb', which is one more than the value of the hint variable 'largest_integer'. Thus the node address of each 'integer' node is the integer itself. This implies all other terminal nodes have an associated node address less than the value of the hint variable 'smallest_integer'. Although a 'type' node is considered to be a terminal, this exceptional case occupies four index numbers, and the node address of such a node is the algebraically smallest of the four numbers. Each non-terminal node of n links (maximum) occupies n index numbers and its node address is the algebraically smallest of those numbers. An initial set of the n entries of a non-terminal node contain node addresses of the destinations of the node's defined links.

The four array entries occupied by a 'type' node are used for the following information:

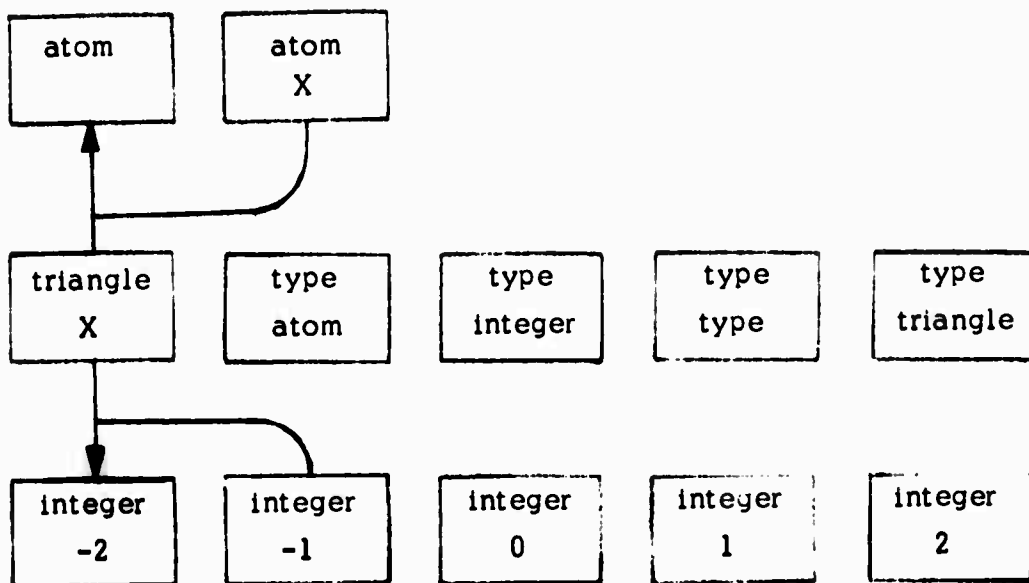
- a) first: an index into the 'nodes' array indicating the starting place of nodes of this type. For terminal nodes, this is the algebraically smallest node address of all nodes of this type. For a non-terminal type of n links (maximum), this is n less than the algebraically smallest node address of all nodes of this type. These extra n entries are used to contain the node addresses of link names of the given type.
- b) second: a positive integer indicating the current number of links defined for this type. This integer remains zero for terminal types; it may increase during an AMBIT/G run for non-terminal types from zero up to the maximum number of links given in the next entry.

- c) third: a positive integer indicating the maximum number of links which may be defined for this type. This entry remains constant throughout an AMBIT/G run. It is the integer zero for terminal types.
- d) fourth: an index into the 'nodes' array indicating the node address of the next node of this type which is available for freshly locating. If no nodes of this type have been located, it is the algebraically smallest node address of all nodes of this type. If all of the nodes of this type have been located, it is an index beyond the algebraically largest node address of all nodes of this type.

To conclude this description, an example will clarify the representation of data graph in the 'nodes_segment'. The example is strictly hypothetical since it does not include the vast built-in data which is initialized in every real AMBIT/G run. Instead, we represent only the following snapshot of a data graph during a hypothetical AMBIT/G run:

<u>Node Type</u>	<u>Current Number of Nodes</u>	<u>Maximum Number of Nodes</u>	<u>Current Number of Links</u>	<u>Maximum Number of Links</u>
atom	2	3	0	0
integer	5	5	0	0
type	4	4	0	0
triangle	1	2	2	3

The small data graph outlined above has 12 nodes. Although the 'nodes_segment' includes no references to names of nodes we will include them here to aid in the discussion. The following is a diagram of the data graph:



On the next page we present a listing which shows the 'nodes_segment' representing the hypothetical data graph. The horizontal lines separate node representations. An asterisk next to an index number indicates that it is also a node address of a located node. Although indices range from -5 to 32, the actual 'nodes' array of the 'nodes_segment' occupies entries with indices 3 to 31. Note the two defined links of 'triangle's and the two links of 'triangle X'. Note how the nodes of type 'type' are arranged in the same order as the entire 'nodes_segment' and thus finding the type of a given node (given its node address) can be accomplished by a binary search. Furthermore, the maximum number of nodes of a given type is implicitly available by looking at the first word of the next 'type' node. This arrangement requires an extra dummy node to end the group of 'type' nodes.

<u>Index</u>	<u>Entry</u>	<u>Subname</u>	<u>Representing</u>
-5*		X	nodes of type 'atom'
-4*			
-3			
-2*		-2	nodes of type 'integer'
-1*		-1	
0*		0	
1*		1	
2*		2	nodes of type 'type'
3*	-5	atom	
4	0		
5	0		
6	-3		
7*	-2	integer	
8	0		
9	0		
10	3		
11*	3	type	
12	0		dummy node to end 'type's
13	0		
14	23		
15*	23	triangle	
16	2		
17	3		
18	29		
19	32		defined links for nodes of type 'triangle'
20	0		
21	0		
22	0		nodes of type 'triangle'
23	-5		
24	-1		
25			
26*	-4	X	nodes of type 'triangle'
27	-2		
28			
29			
30			
31			
32			

names_segment

This segment is a structure defined by the following PL/I declaration:

```
dc1 1 names_segment based(names_ptr_local) aligned,  
    2 max_chars fixed bin,  
    2 names_next fixed bin,  
    2 names_size fixed bin,  
    2 names(hint_names_size),  
        3 named_node fixed bin,  
        3 node_name char(hint_name_length);
```

The value of the first entry of the structure, 'max_chars', is the value of the hint variable 'name_length'. 'names_next' contains an index to the next available entry of the 'names' array described below. The value of 'names_size' is the value of the hint variable 'names_size'. The 'names_segment' ends with a one-dimensional array named 'names' of length equal to the value of 'names_size'. Each element of the array holds a pair of quantities:

- a) a node address representing a named node, and
- b) a character string (of length up to the value of 'names_size') representing the subname of the named node.

Since the 'names' array contains only subnames, the full name of a node is represented not only by its subname, but also by the subname of the 'type' node representing its type.

The 'names' array of the 'names_segment' contains all named nodes except nodes of type 'integer' and built-in nodes of type 'char'. The names of these nodes are handled implicitly in the primitives 'read_name' and 'locate'. A given node has at most one subname. There is no order to the entries of the 'names' array.

We complete this description with a listing below showing the arrangement of the hypothetical 'names_segment' which would correspond to the hypothetical 'nodes_segment'. The values of hint variables 'name_length' and 'names_size' used in this example are the default values used in a real AMBIT/G run.

```

max_chars      = 25
names_next     = 7
names_size     = 1000
names(1)       = 11 , "type"
names(2)       = 3 , "atom"
names(3)       = 7 , "integer"
names(4)       = -5 , "X"
names(5)       = 15 , "triangle"
names(6)       = 26 , "X"

```

defns_segment

This segment is a structure defined by the following declaration:

```

dcl 1 defns_segment based(defns_ptr_local) aligned,
      2 max_args fixed bin,
      2 defns_next fixed bin,
      2 defns_size fixed bin,
      2 defns(hint_defns_size) fixed bin;

```

The value of the first entry of the structure, 'max_args', is the value of the hint variable 'function_arguments'. 'defns_next' contains an index to the next available entry of the 'defns' array described below. The value of 'defns_size' is the value of the hint variable 'defns_size'. The 'defns_segment' ends with a one-dimensional array named 'defns' of length equal to the value of 'defns_size'. Each element is either an index back into the 'defns' array or a node address (index into the virtual 'names' array).

Let M be the value of 'max_args', i.e., the maximum number of arguments a function may have for a particular AMBIT/G run. Then the first $2*M+4$ entries of the 'defns' array are used to contain pointers to various lists as follows:

<u>Entry Name</u>	<u>Contains Pointer to List of Definitions of:</u>
defns(1)	general reading functions
defns(2)	general writing functions
defns(3)	reading functions with 0 arguments
defns(4)	writing functions with 0 arguments
defns(5)	reading functions with 1 argument
defns(6)	writing functions with 1 argument
defns(7)	reading functions with 2 arguments
.	.
.	.
.	.
defns(2*M+4)	writing functions with <u>M</u> arguments

A list pointer either contains a 0 to indicate the list contains no entries, or it contains an index into the 'defns' array to the first entry of a block occupying several entries. One block represents one function definition. Each block on the general reading or general writing list consists of 3 consecutive entries. Each block on an N argument list consists of N+3 entries arranged as follows:

<u>Entry within Block</u>	<u>Use</u>
1	pointer to next block on this list
2	link name
3	type of tail 1
.	.
.	.
.	.
<u>N</u> + 2	type of tail <u>N</u>
<u>N</u> + 3	definition ('rule' or 'builtin')

The first entry of a block is a pointer used to singly-link the particular list of blocks. The last block on a list contains an indicative pointer of 0. When a primitive adds a definition to one of the lists it is pushed down at the head of the list. The remaining entries within a block are the arguments received by a write call on either the 'read_function' or 'write_function' built-in functions. Thus the "type of tail" may either be the node address of a 'type' node or of 'flag any'.

Blocks on the general lists and 0-argument lists consist of three entries arranged as follows:

<u>Entry within a Block</u>	<u>Use</u>
1	pointer to next block on this list
2	link name
3	definition ('rule' or 'builtin')

The blocks entered into the 'defns' array are added at the next available entry, and thus an initial portion of the array is always in use densely packed. Blocks are never removed, and therefore, the array grows longer with each processed definition. Definitions are not even merged. The variable 'defns_next' indicates the extent of use of the 'defns' array at any time.

When a link is defined for either reading or writing, a four-word block is entered on the one-argument reading or one-argument writing list. Furthermore, the link definition is recorded in the 'nodes_segment' in the place used to contain the link names for a given type.

We conclude with an example of what the 'defns' array would be if the value of hint variable 'function_arguments' were 2, and if two links were defined for reading and writing on nodes of type 'triangle', as used in the example we have been employing throughout this section on data formats. In this example, the value of 'defns_next' is 25.

<u>Index</u>	<u>Entry</u>	<u>Representing</u>
1	0	general read
2	0	general write
3	0	0-read
4	0	0-write
5	17	1-read
6	21	1-write
7	0	2-read
8	0	2-write
9	0	block defining 'atom X'
10	-5	as a link for reading
11	15	on nodes of type 'triangle'
12	NABL*	
13	0	block defining 'atom X'
14	-5	as a link for writing
15	15	on nodes of type 'triangle'
16	NABL*	
17	9	block defining 'integer -1'
18	-1	as a link for reading
19	15	on nodes of type
20	NABL*	'triangle'
21	13	block defining 'integer -1'
22	-1	as a link for writing
23	15	on nodes of type
24	NABL*	'triangle'
25		next available entry

* "NABL" means "node address of 'builtin link'". Note that our simple example which ignores built-in data is too simple to represent these definitions completely.

PL/I IMPLEMENTATION OF THE INTERPRETER AND LOADER

Both the AMBIT/G interpreter and AMBIT/G loader were designed and programmed as AMBIT/G programs. However, neither one has been executed in the same manner as a user AMBIT/G program; in fact, neither has been encoded in loader input form. Instead, both have been hand-translated from AMBIT/G into a stylized PL/I form which is a vast succession of procedure and function calls.

AMBIT/G functions were implemented as PL/I procedures with tails and heads passed through the argument list. The translation is done one rule at a time starting with the following labelled statement:

L:

call rule(S,F);

where L is the label of this rule corresponding to the rule's name, S is the label of the rule at the success exit, and F is the label of the rule at the fail exit. Rules which were unnamed in the listing are given labels by a simple algorithm. If a rule in the listing did not include a fail exit, a label 'imp' (for "impossible") is used. If control ever reaches 'imp', error condition 'int4' or 'ldr4' is signalled. Note that this encodement is possible since neither the interpreter nor loader includes any rules which attempt to modify 'success' or 'fail' links of one of their own rules.

The contents of a rule is translated into calls on primitives, calls on functions of the program, and calls on special functions and procedures derived from primitives used only for this stylized form. Within each rule, dummy variables 'd1', 'd2', ... are used for matching dummy nodes to the datn. Finally, the translation of a rule ends with:

call endrule;

We now present examples of function and procedure calls which may be used in the translation of a rule contents. We begin with the calling sequences of primitives:

```

head1 = read_link(tail1,tail2);
call write_link(tail1,tail2,head1);
head1 = read_type(tail1);
head1 = locate(tail1,tail2);
head1 = read_name(tail1,tail2);
head1 = read_char(tail1);
call write_char(tail1,head1);
head1 = get_read_function(tail1,tail2);
head1 = get_write_function(tail1,tail2);
call define_read_function(tail1,tail2,head1);
call define_write_function(tail1,tail2,head1);
head1 = agadd(tail1,tail2);
head1 = agsubtract(tail1,tail2);
head1 = agmultiply(tail1,tail2);
call agdivide(tail1,tail2,head1,head2);
head1 = agsign(tail1);
head1 = loader;

```

The arguments and results which are passed are 'fixed binary' integers representing node addresses. Now we present calling sequences and explanations of the derived procedures and function which may be included in the translation of a rule contents:

Calling Sequence

```

call test(node1,node2);

call test_link(origin,name,dest);

call verify(node1,node2);

call verify_link(origin,name,dest);

```

Use

```

if the two arguments are not
the same, take the fail exit
of the rule.

if the link given by origin and
name does not point to dest,
take the fail exit of the rule.

if the two arguments are not
the same, signal error condition
'int1' or 'ldr1'.

if the link given by origin and
name does not point to dest,
signal error condition 'int2' or
'ldr2'.

```

call verify_type(node,type);

if the type of node is not type, signal error condition 'int3' or 'ldr3'.

dest = ravn(origin,name,type);

this means "read and verify type". If the destination of the link given by origin and name is of type type, return it as the result; otherwise, signal error condition 'int37' or 'ldri3'.

When the 'rule' procedure is called to identify the beginning of a rule, the two given exits are saved (logically, on a stack). If during the execution of the rule a rule failure is detected, the given fail exit is taken immediately. When the 'endrule' procedure is called, it causes the transfer of control to the success exit.

To easily implement the interpreter and loader in PL/I, the AMBIT/G initializer was designed to set up an 'external static fixed binary' variable for each built-in named node mentioned in either the interpreter or loader. The initializer uses a form of the 'locate' primitive to initialize each of these variables with the node address of the node it represents. For example, the variable 'diamond_end' contains the node address of the built-in node 'diamond end'. Thus the encodement of a named node in a rule contents amounts to using the variable which names the node; the variable name is the same as the node name, except an underbar separates the type from the subname.

We conclude by presenting the PL/I statements used to represent one of the rules in the AMBIT/G interpreter. The name of the rule is 'do_rf_r', and it appears on the page of the interpreter listing (in Vol.III) entitled '10'. We purposefully omit discussion of the handling of the error condition.

```

/* 10 */
do_rf_r:
    call rule(b1_return_r,error_do_rf_r);
    d1=ravt(circle_1,link_value,type_linkrep);
    d2=ravt(d1,link_orv,type_diamond);
    d3=ravt(d2,link_next,type_diamond);
    d4=ravt(d1,link_dest,type_diamond);
    d5=ravt(d2,link_value,type_noderap);
    d6=ravt(d3,link_value,type_noderap);
    d7=ravt(d4,link_value,type_noderap);
    d8=read_link(d5,link_rep);
    d9=read_link(d6,link_rep);
    d10=get_read_function(d8,d9);
    call test_link(d3,link_next,diamond_en1);
    call test_link(d4,link_next,diamond_en1);
    call vset(d7,d8,d10);
    go to endrule;
error_do_rf_r:
    call iatldr_error$int12;

```

CHAPTER 3

FURTHER WORK

In the short run, the necessary work to make the AMBIT/G System publically available through Multics should be done. This will entail cleaning up some rough spots in the design and attempting to improve the apparently slow speed of the system. Some minor bugs can be fixed.

Greater public interest in AMBIT/G and a large improvement in its usefulness would result from building a graphical interface for its use. One of the reasons we implemented on Multics is for its support of the ARDS storage tube display terminal and the potential of using the ARPA network in this medium.

In the longer run, we seek the means of providing an AMBIT/G System as a simple and practical tool to a software programmer. Currently, this goal is being pursued with the AMBIT/L Programming System implemented on a D.E.C. PDP-10/50 time-sharing computer.

We know of several deficiencies in the current design and implementation of AMBIT/G. In this chapter we mention a few of these and discuss some of the possible solutions which have occurred to us. It should be borne in mind, however, that our thinking is not yet complete on most of these issues.

Self Interpretation

The AMBIT/G version of the interpreter is, so far as we know, a legitimate AMBIT/G program. We qualify this statement because it has never been tested; only the PL/I program obtained from it has actually been run. A real test of the AMBIT/G version of the interpreter would be to place it, together with a test program, into the PL/I implementation and see if the PL/I interpreter interpreting the AMBIT/G interpreter could successfully interpret the test program.

In fact, we know that such a test would fail, for the three programs would interfere with each other in the use of storage. It seems clear that some sort of block structure is needed to make this work, but we do not know just how it should look.

Error Handling

In the present implementation, all errors are terminal; there is no way for a user's program to regain control after the occurrence of an error. Since the interpreter is written using the same conventions as a user's program, it likewise cannot regain control after a built-in function detects an error.

Some sort of an interrupt facility should probably be added to allow for error recovery. There are actually two examples of interrupts built into the interpreter: the branches to 'rule go' and to 'rule help'.

Explicit Representation of Primitives

In the present implementation, the primitive routines were hand-coded in PL/I and no formal definition of them exists. It is certainly possible to define a representation of arbitrary AMBIT/G data in terms of a fixed collection of shapes. One could then write AMBIT/G programs which manipulate these shapes so as to implement the primitives in much the same way as the AMBIT/G version of the interpreter defines the rules for program execution.

Additional Primitives

The primitive functions 'locate', 'read_function' and 'write_function' currently have no inverses. At least for the sake of completeness, it would seem that there should be a function 'lose' which returns an unused node to the environment and a function 'forget' which removes a definition of a read or write function.

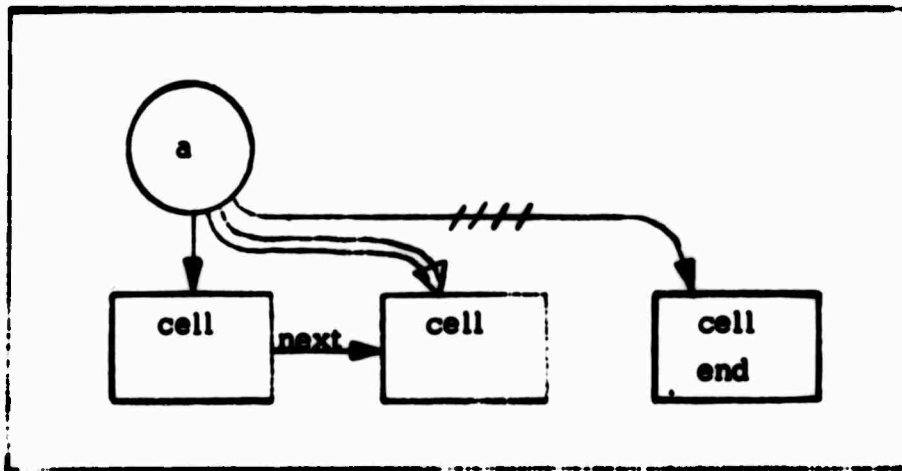
Non-primitive Built-ins

It is likely that, as AMBIT/G evolves, the need for more built-in functions will arise. One would like to be able to take an ordinary AMBIT/G function and, without translating it into PL/I, install it in the interpreter as a 'built-in' in a way that looks to the user exactly as if it were actually a PL/I routine. It would be easy to modify the interpreter to simulate a user-function call whenever a particular built-in is recognized. But this doesn't

have quite the desired effect, for then the user's 'rule' node rather than the interpreter becomes the caller, and of course this difference is detectable by the user (although in any practical sense it is probably not too important).

Other Link Modes

There are many cases in which it would be nice to have a link in a rule with opposite effect of a 'test' link, that is, the rule would fail only if the data graph did match the link. Diagramming such a link by slashing the arrow, we could then write the rule



which would advance the pointer 'circle a' only if it had not previously reached 'cell end'. A disadvantage of such a "link", of course, is that it lessens the gestalt feeling of the language, but so do function calls.

Regardless of the merits of such a link, perhaps a mechanism should be provided whereby the user could extend the number of available link modes and define interpretations for the new ones. How to do this is still very much an open problem.

We have generalized links in rules to the point where any link can denote any operation and the particular operation is determined dynamically. However, we at present do not allow a different function to be used for mode 'frame' as is used for mode 'test'. One could generalize the 'test' link to say that both the tails and the heads are passed to a function which then decides, in an arbitrary manner, whether or not the test succeeds. However,

this would require (at the least) that we provide some way for a function to return a failure indication.

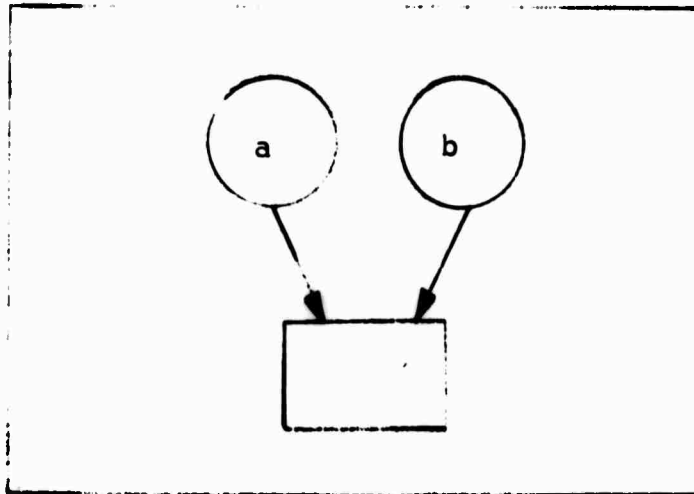
Order of Execution and Flow Links

When functions have side effects, the order in which they are called often makes a difference. It is sometimes convenient to be able to call two functions from the same rule and to specify separately which is to be executed first. Flow links, as in AMBIT/L, are one device which permits such a specification and perhaps should be added to AMBIT/G.

A more common and also more subtle problem arises when one wishes to make a statement of the form: "if a certain portion of the data satisfies some condition, then another portion of the data will be as indicated and should be modified accordingly." Another way of saying the same thing is, "if a certain portion of the data does not satisfy the condition, quit immediately and do not attempt to match the frame to the remainder." More generally, one might wish to intermix quite thoroughly the order of execution of the three modes of links: first establish a little bit of frame, then check some condition, then some more frame, another condition, perhaps now a modification, then some more frame, and so on. Flow links, of course, could also specify such an order, but it is not clear to what extent this is desirable. Moreover, modifications can cause very subtle complications, although so can side-effects of other functions.

Constraints

The only constraints that the present implementation will accept and enforce (other than hints) are those resulting from redundancy in the 'frame' portion of a rule. For example, the rule



is essentially a constraint that says, when it is executed, 'circle a' and 'circle b' point to the same thing.

Many other constraints have been proposed, and the intention of eventually providing for a considerable variety of constraints had a strong influence on the design of the language, tending to make us less concerned over inefficiency resulting from increased generality on the belief that, at the very least, the generality could be constrained away and the efficiency restored. Of course, actually doing so would require an implementation able not only to check the validity of constraints but also to take advantage of them wherever possible.

Continuing research must be done in the ways constraints can be incorporated into the language. We seek to expand the vocabulary of constraints to extend the ability of AMBIT/G to model machine language software.

Adding constraints and building a compiler to utilize them is probably the most important and most difficult remaining task.

CHAPTER 10

PROJECT BIBLIOGRAPHY

This chapter is composed of a list of papers and a list of implementations. An item appears in these lists because it was produced as part of the project (in which case it is marked with a star, *) or because it pertains directly to AMBIT/G. Only immediately relevant and generally available papers are mentioned. The list of implementations may be incomplete, since we include only those of which we have direct knowledge.

- P1. Christensen, Carlos. "An example of the manipulation of directed graphs in the AMBIT/G programming language." In Klerer and Reinfelds, eds., Interactive Systems for Experimental Applied Mathematics. Academic Press, New York, 1968.

This is the first paper on AMBIT/G. It remains useful because it has a complete listing and careful explanation of the link-bending garbage collection program (for LISP) written in AMBIT/G. This program has, to our knowledge, been run as a test case on every implementation of AMBIT/G. It is a good example of AMBIT/G because it is quite short but decidedly non-trivial.

- P2. Cheatham, T.E., Jr. "The theory and construction of compilers." Massachusetts Computer Associates, Wakefield, Mass., June 1967 (to be published as a book).

This textbook on compiler-writing makes very successful use of AMBIT/G data structures to describe and explain a variety of algorithms for syntactic analysis.

- P3.* Henderson, D. Austin, "A description and definition of simple AMBIT/G -- a graphical programming language." Massachusetts Computer Associates, Wakefield, Mass., April 1969.

The paper consists of two descriptions of simple AMBIT/G: the first is in English and is quite informal; the second is in mathematical notation and constitutes a formal definition of the language expressed in predicate calculus.

- P4. Rovner, Paul D. and Henderson, D. Austin, "On the implementation of AMBIT/G: a graphical programming language." Presented at the AFIPS/ACM International Conference on Artificial Intelligence, Washington, D. C., May 1969.

This paper describes an interactive AMBIT/G system, with input through a graphics tablet and output on a computer driven display. The implementation is on the TX-2 at M.I.T. Lincoln Laboratory.

- P5.* Jorrand, Philippe. "Some aspects of BASEL, the base language for an extensible language facility." Proceedings of the Extensible Languages Symposium, Boston, May 1969, published as the August 1969 edition of SIGPLAN Notices.

BASEL was designed as the base language component for an extensible language facility called ELF. ELF was intended to have several components: one for syntactic extension, one for definition of communications with a given kind of environment, and some others. It follows that, on the one hand, BASEL must have a very simple syntax and, on the other, it must be a very "powerful" language.

- P6.* Hammer, Michael M. and Jorrand, Philippe. "The formal definition of BASEL." (in three volumes). Massachusetts Computer Associates, Wakefield, Mass. August 1969.

The purpose of this document is three-fold: to discuss in some detail the subtler features of the BASEL language; to indicate the issues involved in defining a language in terms of the two-dimensional, machine-independent programming language AMBIT/G; and to provide the actual programs for the BASEL compiler and interpreter written in AMBIT/G. A basic familiarity with the concepts of BASEL and AMBIT/G is assumed. Part 1, "Introduction", describes in an informal and instructive way the AMBIT/G data which is used in the compiler and interpreter, and then proceeds to a discussion of the defining programs themselves. Part 2, "Compiler", is an AMBIT/G program which converts the output of a conventional parsing routine into a form suitable for interpretation. Part 3, "Interpreter", is an AMBIT/G program which interprets the compiled form of a BASEL program.

- P7.* Ledeen, Kenneth S. "A character recognizer." Massachusetts Computer Associates, Wakefield, Mass., August 1969.

A real-time character recognition scheme, that is, an algorithm for associating a sequence of pen movements with a character code and display form, was designed and implemented for the Harvard University PDP-1 computer with Grafacon tablet and CRT display. The program allows the user to "train" the recognition program to recognize his individual printing style, and to draft display characters of his own design. The original intention, not realized, was to use this Character Recognition System as the input mechanism of the AMBIT/G implementation.

- P8.* Wolfberg, Michael S. "A user's view of the character recognition program." Massachusetts Computer Associates, Wakefield, Mass., August 1969.

The paper first presents a user's view of the Character Recognition System (see Ledeen, above) in completely verbal terms. Next an informal two-dimensional notation is introduced and is used to document the program from the user's point of view. Finally, a series of photographs of the screen is presented which documents for the reader a sample session of using the Character Recognition System.

- P9.*** Wolfberg, Michael S. "An interactive graph theory system." Preprint of a paper presented at the Computer Graphics 70 International Symposium, Brunel University, England, April 1970. Massachusetts Computer Associates, Wakefield, Mass., March 1970.

This paper describes an interactive graphics system for solving graph theoretic problems. The system is implemented on a remote graphics terminal with processing power connected by voice-grade telephone line to a central computer. The potential of using the terminal as a programmable subsystem has been exploited, and computing power is appropriately divided between the two machines. In order to express interactive graph theoretic algorithms, the central computer may be programmed in an algorithmic language which includes data structure and associative operations. Examples of system use and programming are presented. The writing of this paper (but not the work described in the paper) was performed as part of the AMBIT/G project.

- P10.*** Christensen, Carlos and Wolfberg Michael S. "AMBIT/G as an implementation language." To appear in the Convention Digest of the IEEE International Convention and Exposition, New York, March 1971.

This is a summary of a talk to be given in the session "Manufacturing Software, the Case for High Level Languages", chaired by J.W. Poduska. It is our most recent version of a concise, introductory description of our work on AMBIT/G.

- P11.*** Christensen, Carlos. "An introduction to AMBIT/L, a diagrammatic language for list processing." To appear in the proceedings of SYMSAM/2, the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March 1971.

AMBIT/L is a list-processing programming system. The system grew directly out of AMBIT/G and achieves practical value by accepting limitations on the generality of AMBIT/G. 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 accommodate 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.

- P12.*** First semi-annual technical report for the project Research in Machine-Independent Software Programming. Massachusetts Computer Associates, Wakefield, Mass., February 1969.

This semi-annual report provides a detailed discussion of the background, basic approach, and research plan of the project.

- Pl3.*** Second semi-annual technical report for the project Research in Machine-Independent Software Programming. Massachusetts Computer Associates, Wakefield, Mass., August 1969.

This report describes our work on the PDP-1 implementation of AMBIT/G (which was not completed), on the modelling of BASEL in AMBIT/G, and on the general design of AMBIT/G.

- Pl4.*** Third semi-annual technical report part I (covering task area I) for the project Research in Machine-Independent Software Programming. Massachusetts Computer Associates, Wakefield, Mass., February 1970.

This semi-annual report gives a complete description of AMBIT/G. It is especially recommended for its discussion of constraints. It contains explanations and examples of the concept of constraint which have not appeared elsewhere.

- Pl5.*** Christensen, Carlos; Wolfberg, Michael S.; and Fischer, Michael J. "A report on AMBIT/G", final report -- task area I (in four volumes) for the project Research in Machine-Independent Software Programming, Massachusetts Computer Associates, Wakefield, Mass., February 1971.

An AMBIT/G system has been implemented on the Multics System at M.I.T. The implementation is ostensive and is intended for experiments in the use of AMBIT/G. It is written partly in AMBIT/G and partly in PL/I. This report begins with fundamental concepts and then proceeds to describe the implementation in great detail. The AMBIT/G programs for the AMBIT/G interpreter and the AMBIT/G loader are described and then displayed in full. Instructions for the input, execution, and debugging of a user program are given. Many examples are included, carefully chosen to illustrate and teach important features of AMBIT/G.

IMPLEMENTATIONS

- 11.** Moskovites, Peter. An AMBIT/G Compiler implemented on the SDS-940 at Harvard University by Massachusetts Computer Associates, completed August 1967.
- 12.** Rovner, Paul; Henderson, D. Austin; and Greenberg, Martha. An Interactive AMBIT/G System with Graphic I/O, implemented on the TX-2 at M.I.T. Lincoln Laboratory, initial system completed April 1968, and revised system completed Fall 1968.
- 13.*** Wolfberg, Michael S.; Supnik, R.; and Ledeen, K.S. A Character Recognition System, implemented on the PDP-1 at Harvard University by Massachusetts Computer Associates, completed August 1969.
- 14.** An Experimental Implementation of AMBIT/G, implemented on the ATLAS computer, Cambridge University, England, Summer 1969.

15. **Christensen, Carlos; Muntz, Charles; and others. A Complete AMBIT/L Programming System, implemented on the Applied Data Research PDP-10 in Princeton, N.J. by Massachusetts Computer Associates, completed September 1969.**
- 16.* **Wolfberg, Michael S.; Fischer, Michael J.; and Ho, Maynie. An AMBIT/G System, implemented on the Multics System at M.I.T. by Massachusetts Computer Associates, completed December 1970.**