Massachusetts
COMPUTER ASSOCIATES, Inc. / Lakeside Office Park • Wakefield, Massachusetts 01880 • Area Code 617 245-9540

AD 683362

FIRST SEMI-ANNUAL TECHNICAL REPORT

(21 June 1968 - 21 December 1968)

FOR THE PROJECT

"RESEARCH IN MACHINE-INDEPENDENT SOFTWARE PROGRAMMING"

Principal Investigators:

T.E. Cheatham, Jr. Phone (617) 245-9540
Carlos Christensen Phone (617) 245-9540

Project Manager:

Carlos Christensen Phone (617) 245-9540

ARPA Order Number ARPA 1228
Program Code Number 8D30

Contractor: Massachusetts Computer Associates, Inc.
Contract No.: DAHC04 68 C 0043
Effective Date: 21 June 1968
Expiration Date: 21 December 1969
Amount: $124,530.00 (for initial 12-month period)

DDC
RECEIVED
MAR 5 1969
B

Sponsored by
Advanced Research Project Agency
ARPA Order No. 1228

This document has been approved for public release and sale; its distribution is unlimited

37

Reproduced by the
CLEARINGHOUSE
for Federal Scientific & Technical
Information Springfield Va. 22151

SUMMARY

This document constitutes the first Semi-Annual Technical Report for the project "Research in Machine-Independent Software Programming". Since it is the first formal report of work on the project, it includes an introduction to the project, a discussion of the background on which the project is based, a statement of our basic approach to the problem addressed, and an outline of the research to be performed under this project. This expository information is followed by a description of the work which has been done during the first six months of the project. This description is divided into four parts each describing a major area of project activity. The document concludes with a list of project reports which are currently in preparation and a bibliography of related papers.

CONTENTS

Introduction	1
Background	3
Basic Approach	7
Research Plan	9
Current Results	18
Reports in Preparation	30
Bibliography	33

INTRODUCTION

During the past decade the "credibility gap" in software specification and construction has widened alarmingly, while that in hardware specification and construction has narrowed until the time and cost of specifying and constructing rather major hardware systems now can be confidently predicted. The lessons learned from the software "failure" of many large-scale command and control systems, of major commercial and university time-shared and batch operating systems, and of attempts to provide "complete" language facilities such as promised by PL/I have not yet resulted in the development of a basic program construction technology which permits the prediction and control of software costs and facilities.

The cost of construction of programs of large size or complexity on a "one-shot" basis is difficult to predict in any case. Of more importance, however, is the fact that even when a large program finally operates in an acceptable fashion the program is completely un-transferrable. To obtain a software facility which is functionally the same on different equipment usually requires an effort of the same order of magnitude as the original implementation. In this era, when computer hardware and communication facilities have developed to the point that we have the possibility of actually sharing facilities among a large number of systems by means of a network of large scale time-shared computer systems, our current difficulties with software will doubtless preclude

anything much more advanced then sharing of data for some time to come. The hope held out by the advocates of the so-called higher level languages for system construction, notably JOVIAL and PL/I, has not been realized. While these languages have been useful on several large scale programming tasks, they do not in any sense "solve the problem". It has been found that these languages are inadequate for the construction of large scale software (including such relatively minor components as their own compilers) because they do not provide facilities for the introduction and manipulation of appropriate data structures and they rather strictly adhere to imperative forms which mirror conventional "engineering computations". As a result, the programmer has little or no control over the linguistic representation of his data and algorithms, and thus often finds the representation he is forced to utilize completely alien to him.

The object of the current project is to develop solutions for these problems of software programming technology. The project activities consist of research into machine-independent techniques for programming computer software, of the design and implementation of certain specific language and system facilities, and of experimentation with the machine-independent specification and implementation of typical software components.

BACKGROUND

We have been concerned with machine-independent programming for many years. The three sections which follow trace the progress of our activity in this area beginning with major work in both translator-writing systems and extensible operating systems, continuing to the development of several languages based on pattern-matching, and leading to our current work on the AMBIT/G language.

Machine-Independent Programming

For nearly ten years we have been active in research, development, and implementation of translator-writing systems. The work began with the Compiler Generator System, CGS [2,18,19,21], and continues with the Translator Generator System, TGS. In the course of this work we have invented, re-invented, and evaluated many techniques of machine-independence, including bootstrapping, syntax-directed compiling, and automatic program optimization. We have had ample opportunity to observe the difficulties attendant on the construction of a machine-independent compiler. We have noted that efforts to achieve machine-independent programs often conclude with a very heavy and serious dependence on the operating system in which the program is used.

Over the same period, we have been concerned with the design of extensible and transportable operating systems. This work began with the CL-I and CL-II Systems [1,22], continued with the design of the Automatic Operating and Scheduling Program, AOSP [17], and with research on extensible systems [15,16]. This work has a special importance, because

although many people have developed translator-writing systems, rather few people have given attention to the basic problems of operating system design. Our objective in this work has been to eliminate explicit limitations with respect to the availability of processors and memory, and to thus remove a fundamental source of machine dependence from computer programming.

Probably the greatest value to us of the work described above has been our experience with attempts to actually implement our designs for large and complex hardware configurations. Many seemingly elegant notions collapse under this kind of implementation test, and we have developed a familiarity with the areas in which weakness develops in the implementation of a large software system.

In the past few years, our work on translator-writing systems has evolved into the consideration of language extension [4,5]. Here we seek to replace the increasingly large set of high-level languages with a single language system in which the user may introduce new constructions to suit his particular needs and may modify or delete those which are not appropriate.

Our activity in compiler design and construction has been accompanied by work on methods of program optimization. For some time, this work produced rather specialized devices for the processing of common sub-expressions, the efficient allocation of registers, and so on. Recently, however, important progress has been made toward developing a more general model for program optimization [20]. This model provides a single representation for a class of algorithms which differ in their sequencing of operations but which all perform the same input-to-output mapping. It has provided us with important guidance in formulating our basic approach to the development of AMBIT/G.

Pattern-Directed Languages

In the course of the work just described, a vigorous interest in pattern-directed languages has developed. This work is based on the notion that if data is assigned plausible structure (such as a fully parenthesized string of tokens), then the programmer can access and modify this data very conveniently by writing a structure which may match (or may fail to match) a sub-structure of his data.

We began our work with pattern-directed languages with AMBIT/S, a language for algebraic symbol manipulation [7,8,9] and the Translator Generator System, TGS. The AMBIT/S language assumes that its data is a parenthesized string of tokens to which access is obtained by means of pointers into the string. It has been used to program algorithms for algebraic symbol manipulation and is the basis for a complete interactive system for algebraic symbol manipulation. The Translator Generator System assumes the data to be a special set of stacks and trees appropriate for compiling, and includes a variety of facilities for pattern definition. It has been used to write several compilers, one of which was a very large compiler for an immediate predecessor of PL/I [10].

Our work with pattern-directed languages has convinced us of the value of this approach to the writing of structure-manipulating programs. But it also exposed the weakness of attempting to fix, once and for all, the data structure on which all users of the language must operate. AMBIT/G was developed to correct this weakness.

The AMBIT/G Language

The invention of AMBIT/G was preceded by experiments with the informal description of syntactic analysis. The data for syntactic analysis can conveniently

be thought of as a tree; and in these experiments we made this explicit by writing both the data and the transformation to be applied to the data as diagrams of trees rather than as some linear encodement of trees.

The invention of the AMBIT/G programming language [11] occurred rather abruptly in February 1967. It was recognized that it was both legitimate and practical to view any structured data as actually existing in the form of a diagram -- specifically, a directed graph. And, further, a program could be composed almost entirely of transformations -- so-called "modification rules" -- which were themselves diagrams of the data. Graphic consoles have already reached the point at which it is feasible to compose and execute such a diagrammatic program by means of tablet and scope, and a version of AMBIT/G is currently being implemented on such a console.

AMBIT/G has been used in two areas in which programming is notoriously difficult. First, the algorithms for syntactic analysis mentioned above have been expressed in AMBIT/G and have been used in this form for teaching syntactic analysis [3]. Second, a large and complicated language processor has been modelled entirely in AMBIT/G. The latter effort has established that AMBIT/G can be used with great effectiveness for the machine-independent programming of computer software.

BASIC APPROACH

The purpose of a computer program in general, and of a computer software program in particular, is to define an input-to-output mapping. Any one such mapping can be defined in many ways; that is, there is a large family of algorithms each of which produces the same mapping. These algorithms are distinguished from one another not by the mapping they perform, but by the variety of methods they use to perform the mapping.

As an ideal, we can conceive of a programming environment in which it would be possible to express in a single program all of the algorithms by which a particular mapping could be obtained. If such a program were, at the same time, readable and easily understood by any programmer, then it would be very valuable. By a process of qualification, it could be specialized to contain successively fewer algorithms for the desired mapping until only one -- the most efficient for a given hardware configuration -- remained.

In designing such an environment we must, of course, make some restrictions. The first restriction is to include only algorithms which might reasonably be performed efficiently by an automatic computer. Although there does not exist any useable body of techniques for deriving this restriction objectively, rather good estimates of 'standard computer power' can be obtained intuitively. Indeed, the designs of FORTRAN, COBOL, ALGOL, etc. are all

based on such estimates for their intended application areas; and such weakness as do exist in these languages are not, for the most part, the result of misjudgment of computer power.

Given a restriction on algorithms to those which are appropriate for efficient automatic execution, the problem remains to find a way to embody the remaining family of algorithms for a given input-to-output mapping into a single program which is easy to read and to modify in its original form and is also easy to specialize for execution on particular hardware.

AMBIT/G is a language which opens important new possibilities for such a machine-independent specification of a program. In the current project we are pursuing a program of research which will lead to new facilities for producing machine-independent software through the development and use of AMBIT/G.

RESEARCH PLAN

In this section we discuss three specific areas of research, each of which will make a contribution toward the development of machine-independent software. The first area is the investigation of the properties of the large family of pattern-directed languages which currently exist and of which AMBIT/G is a member. The second area is the design and implementation of the AMBIT/G language. The third area is the application of AMBIT/G to the production of machine-independent software.

An Investigation of Pattern-Directed Languages

There exists a large family of languages which have as their central feature the notion of pattern matching. These include, on the one hand, the mathematical models represented by the varieties of Markov algorithm; and, on the other hand, the programming languages such as AMBIT/G, ASP, Formula ALGOL, AMBIT/S, and SNOBOL. The members of this family of languages have many advantages and problems in common, and we propose to investigate these common factors. The following research descriptions are intended to be typical of the work we plan to do under this heading, although the details of this plan will undoubtedly change during the course of the project:

-- An Analysis of the Family of Pattern-Directed Languages

As noted in the preceding paragraph, there are a rather large number of pattern-directed languages - indeed systems - which have been developed. Recently we have been interested in trying to analyze and

relate these various languages from two points of view. On the one hand, it is interesting to view the string manipulating languages as extensions of or variations on Markov Normal Algorithms (MNAs). Since MNAs are equivalent to such formalisms as Turing Machines, it is clear that they can provide a basis for effective computability. Thus, the demonstration that various extensions are, in fact, equivalent to MNAs provides an assurance of their inherent power; the difficulty or size of an MNA equivalent to, say, some PANON or AMBIT/S construct is, in some sense, a measure of the linguistic or representational power of these latter facilities as contrasted with the rather primitive facilities in pure MNAs.

On the other hand, it has proved most interesting and revealing to take a language like AMBIT/G as a basis and to model MNAs and their extensions in terms of AMBIT/G. It is clear from our preliminary investigation that AMBIT/G is at once more "primitive" than an MNA but at the same time is at a level which is rather more comprehensible. Further, it seems clear that mirroring such string (i.e. one-dimensional data) processes as syntactic analysis, algebraic simplification, and the like in the higher dimensionality inherent in AMBIT/G may lead to some very interesting results.

-- <u>The Scoping and Protection of the Data of a Pattern-Directed Program</u>

In a programming language such as ALGOL in which the structure of the data is pre-defined, the scoping and protection of the data is built into the system. The danger of inadvertantly modifying one value in the course of modifying another is small because of the discipline which is inherent in the system.

On the other hand, in a pattern-directed language the tendency is to think of all of the data as being organized as a single data structure. This data structure may indeed contain items which function as names for other data, and it will contain delimiters to separate or group substructures of data. But the programmer has the capacity to modify the data structure as a whole and, in particular, to manipulate names and delimiters within the data. Thus it becomes possible to inadvertently upset large portions of the data as the result of a programming error.

We intend to investigate the apparent conflict between the advantages of the scoping and protection of data which results from a built-in data structure and the advantage of flexibility and extensibility which results from complete programmer control over the structure of his data. We anticipate a result which will permit the programmer to dynamically define and modify the rules by which scoping and protection of data are achieved.

The Communication of Difficult Algorithms by Means of Pattern-Directed Languages

There exists a collection of processes which are felt to be conceptually simple and basic but which have been found to be difficult to communicate. The difficulties in communicating these algorithms appears to be fundamental to the extent that they lie in person-to-person communication rather than in person-to-computer communication. These algorithms are often characterized by the presence of "bookkeeping" which a human being somehow does not require (at least consciously) in performing the corresponding process.

Syntactic analysis is an excellent example of a process which is conceptually simple but difficult to program. The process produces the analysis of a given sentence according to a given grammar; and after studying a few examples of the desired result, many people can perform the process rapidly and accurately. But these same people will be surprised and baffled by the complexity of a written algorithm for the process.

There are, of course, a considerable number of distinct algorithms for the process of syntactic analysis -- top-down, bottom-up, reductions, and so on. We have already experimented extensively with the writing of these algorithms in AMBIT/G. We have found that much of the troublesome bookkeeping can be eliminated by choosing a data structure which is designed for the process of syntactic analysis.

We intend to continue our investigation of the application of AMBIT/G to the programming of small but difficult algorithms, with particular emphasis on the collection of algorithms for syntactic analysis. Because these algorithms are small, it will be possible to study their characteristics in detail and to use them as a test-bed for proposed modifications to the design of AMBIT/G.

The Design and Implementation of AMBIT/G

The statements of an AMBIT/G program are of two kinds: imperatives and declaratives. The imperatives of an AMBIT/G program fully define a single input-to-output mapping; but they do not define many of the details of the algorithm which produces the required output from the given input. That is, the imperatives represent a class of algorithms, each member of which performs the same input-to-output mapping. On the other hand, the declaratives

of an AMBIT/G program specify the details of the algorithm to be used, and thus restrict the meaning of the AMBIT/G program. That is, the declaratives eliminate members of the class of algorithms represented by the imperative part.

A central feature of AMBIT/G is the clear and effective distinction between the imperative language and the declarative language. Although the immediate appeal of AMBIT/G lies in its use of diagrams to represent structured data, we believe that the underlying power of the language lies with the distinction between imperatives and declaratives. In the design of AMBIT/G, we will continue to strengthen this distinction.

In the following two sections we consider separately our objectives in the design of the imperative and declarative language of AMBIT/G.

-- The Imperative Language

AMBIT/G data is a single directed graph, the data graph, with nodes and shapes chosen by the programmer to be most convenient for the task which is to be programmed. The basic imperative of the AMBIT/G language is the modification rule, which is a representation of the data graph before and after a proposed modification has been performed.

AMBIT/G is deliberately ambiguous in two respects. First, it assumes that the data is actually laid out as a directed graph, and leaves out of consideration the layout of tables, lines, fields, and bits which may ultimately be used to represent this data in a computer memory. Second, the modification rule does not specify a particular method by which its pattern is to be matched to the data graph. As a

result of these ambiguities, an AMBIT/G program actually represents not a single algorithm but rather a class of algorithms which differ in the representation of data and the details of pattern matching.

Our continued work on the design of AMBIT/G imperatives will be directed largely toward the development of other forms of ambiguity. An example of such a development is the introduction of reversibility of program flow, which will permit the programming on Non-Deterministic Algorithms as discussed by Floyd [12]. A second example is the introduction of notations for parallel flow based on the ideas developed by Shapiro [20].

-- The Declarative Language

The design of the declaratives of AMBIT/G is a challenging problem. For any set of imperatives which define an input-to-output map and for any hardware configuration there must exist a set of declaratives which restrict the imperatives to the given hardware configuration. These declaratives must restrict the program to the point at which it is a single algorithm for a specific data base on the given computer.

Beyond this, it will be necessary to use a declarative with a variety of scope over program and data. On the one hand, the user may wish to use a declarative to restrict a small detail of his program; on the other hand he may wish to use a single declarative to impose a restriction on all of the imperatives in the program.

The design of the declaratives for AMBIT/G will be an important area of research within the project.

Other work on the design of AMBIT/G will be directed toward the development of extension facilities for the language. It is clear that a facility comparable to the macro-expander of a conventional language is required; but it remains to be seen exactly how this concept can be transferred to the directed-graph diagrams of an AMBIT/G program.

The implementation of AMBIT/G will serve three purposes within the project as a whole. First, it will permit us to experiment with and evaluate new design features in the language. Second, it will be the basis for testing the feasibility of specializing a program by means of declaratives to produce efficient code for a given target machine. Finally, it will permit us to experiment with the machine-independent programming in AMBIT/G of actual software components.

Machine-Independent Computer Software

At present there have been several occasions on which AMBIT/G has been used in a non-trivial way in designing and/or specifying (i.e. "programming") software components. These include:

-- Semi-formal Model of a Programming Language

The development of a semi-formal model for a new (but rather simple) programming language. This model utilizes an augmented context free grammar to specify the syntax and the semantics of the language; the augments to the grammar are in AMBIT/G and the result of analyzing a source string in the language is an AMBIT/G data structure. A program executer, which, given the data structure which would result from analyzing a source program, will then carry out the program is also written in AMBIT/G. A preliminary draft of a report describing this model is attached.

-- Advanced Language Design

We have been involved in designing an "extensible" or "enhanceable" programming language for several months. Briefly the language is to be (through extensions) of at least the power of ALGOL-68 or PL/I but, hopefully, without the anomalies and excessive complexity (in the sense of constructing compilers for them) of these languages. Recently AMBIT/G has been used in the representation of declarations and subsequent processing of expressions involving new data types defined by a user; this area is generally "messy" in an ordinary compiler and with a language which permits declaration of new data types and new operations on these data types finding an appropriate means for specification of the semantics was formidable indeed. AMBIT/G has proved extremely valuable here, and in two ways. First, it seems a very appropriate language for specifying the semantics in that it is "readable" and in that it is reasonably free of representational commitment. Secondly, we have found that AMBIT/G programs are sufficiently readable that asymmetries in the language which did not show up in the grammar or in various examples were often directly observable in the AMBIT/G semantics of various declarations and statements. That is, AMBIT/G in this instance is having a very useful effect in the language design as well as in the specification of a language translation. Unfortunately, the results here are not in reportable form at this time.

-- AMBIT/G in AMBIT/G

Several students who are implementing a preliminary AMBIT/G system on the TX-2 have found that AMBIT/G is an excellent vehicle for describing its own translator, and, perhaps, more importantly, for specifying the program which will produce output of AMBIT/G data structures on a CRT.

Under the current project we intend to continue this work. In particular, we will actually implement one or more compilers described in AMBIT/G (using the AMBIT/G implementation discussed earlier) and then experiment with various strategies for representing AMBIT/G constructs on different target hardware. We might here hope for a machine-independent (but complete) compiler model which, demonstrably, could enjoy efficient representation on different equipments.

We will also investigate the specification of operating system components with AMBIT/G; it is very possible that here the AMBIT/G freedom from representational commitment may yield methods for machine-independent specification which are even more dramatic than the compiler models discussed above.

CURRENT RESULTS.

In this section we describe the results of project activity during the first six months of the project, that is, for the period 21 June 68 through 21 December 68. These results are described under headings which represent the four tasks into which the project has been divided. The information given here is a summary of the project results; a much more detailed statement will be given in several Project Reports which are now in preparation. Abstracts of these reports are given elsewhere in this document.

Task 1: An Investigation of Pattern-Directed Languages.

The object of this task is to perform research into the properties of pattern-directed languages, with attention to the advantages and disadvantages peculiar to these languages, and devise general improvement and extensions to these languages.

Work on this task has proceeded to two areas: the programming of syntactic analysis in a diagrammatic programming language and the consideration of design problems which are peculiar to pattern-directed languages. These areas are discussed in the following paragraphs.

Applications to Syntactic Analysis. The application of AMBIT/G to the representation of algorithms for performing syntactic analysis has been pursued during

the first six months of the project and we have at present developed a technique for describing analysis which has been documented (see [P1]) and also used quite successfully in a classroom context (Applied Mathematics 295, Theory and Construction of Compilers, Fall Term 1968-9, Harvard University).

In brief, the technique is as follows: We first introduce the notions of non-deterministic algorithms (see [12]) and of context free grammars, syntax trees, and so on. We next pose the problem of syntactic analysis of some source text alleged to be generated by some given context free grammar as that of constructing the analysis tree. We then observe that this, in turn, can be viewed as a game, not unlike the childrens game of dominoes, in which one has "syntactic" dominoes corresponding to the various syntax rules of the grammar as well as the symbols of the source text and the "goal" (or root of the analysis tree). The construction of the analysis tree can now be viewed as a game; the construction of the analysis tree corresponds to choosing and placing syntactic dominoes so as, in the end, to have all edges abutted. The various methods of syntactic analysis (i.e. various ways of constructing the analysis tree) can be viewed as strategies of play of the game.

We then develop "top-down", "left-corner-bottom-up" and "direct reductions" strategies of play; these strategies are very easily comprehended and, when comprehended, ensure that the student really understands the basic mechanisms required for the three kinds of analysis. Following this we formalize the dominoes and the strategies as AMBIT/G data objects and programs; this has the very distinct advantage that the two-dimensional aspect (the representation of trees and so on) is indirectly modelled. Following this we look at "tricky" ways of encoding the same logic using conventional programming language techniques (i.e. integers to represent symbols, sectors and matrices of integers to represent strings, etc.) and finally look to adding selectivity (i.e. reducing the amount of choice which has to be made non-deterministically). The result is a program for each of the three basic methods which is a more-or-less efficient analyzer but which has the obscurity normal to such programs. The student then has representations which range from the "game" which is informal but very readily understood to an AMBIT/G program which is formal but still readily understandable, and, on to a program (in, essentially, ALGOL) which is efficient but rather obscure.

Design Problems of Pattern-Directed Languages. Work in this area has been oriented toward the design of AMBIT/G. This work will eventually be relevant to all pattern-directed languages. We have not yet developed these more general results. Accordingly, the current work on design problems is reported under Task 2, below.

Task 2: Design of the Programming System.

The objective of this task is to design a complete programming system for the machine-independent implementation of computer software. This design work will be based on the results of the research performed under Task 1 and on the already-existing design for the AMBIT/G programming language [11].

During the past six months, design work has proceeded in parallel in three almost independent areas: the design of the basic AMBIT/G language, the design of a specialized language called AMBIT/L, and the design of the input-output graphics interface for AMBIT/G. This work is described in the following paragraphs.

Basic AMBIT/G. We have given extensive consideration to several fundamental problems language design which arise in AMBIT/G and, to an extent, in all pattern-directed language. In seeking solutions to these problems, we have often turned for guidance to the BASEL Formal Model, a large scale application of AMBIT/G programming. We have thus been able to resolve design problems with the aid or practical experience with the language as well as by means of abstract principles of language design.

We have assumed the published description of AMBIT/G [11] as a point of departure. In this context, the following issues have arisen:

1. Node Type and Link Name. In the published description of AMBIT/G, the type of a node could only be indicated by the shape of the node boundary, and the name of a link could only be indicated by the position (middle of left side, lower-right corner, etc.) of the origin of the link on the node boundary. It is apparent, however, that when a data base is large and complex it is

often preferable to use explicit mnemonic identifiers to specify the type of a node boundary or the name of a link. Accordingly, the design of the language should include explicit typing of nodes and naming of links as an option to be used where it contributes to clarity.

2. Identifier Scoping. The introduction of a block structure into the language is necessary to scope identifiers, especially in connection with the definition of a function. The ALGOL 60 block structure when expressed in a suitable diagrammatic form is appropriate for this purpose, and will be adopted in AMBIT/G. We note, however, that although scoping of identifiers is useful in AMBIT/G, the AMBIT/G programmer will use relatively fewer identifiers than the ALGOL 60 programmer. This is because an AMBIT/G structure makes feasible and even encourages the use of links to pass from one point in the data to the other and thereby eliminate the maintainance of temporary pointers which would be used in an ALGOL 60 data structure.

3. Functions and Procedures. We have experimented extensively with representations of function-calls and procedure-calls in AMBIT/G. A function-call is represented as a special node with special links to obtain arguments and deliver results; it is used within a modification rule as an integral part of the program. A procedure-call is a more conventional notation, and closely resembles the procedure call of ALGOL 60; that one of these two facilities should be viewed as fundamental and the other as derivative. We have not made a final decision in this matter, but it is probable that the function call, with its facility for nesting within a larger rule, will become fundamental.

4. Arithmetic. The "built-in" facilities for arithmetic and related operations which must eventually appear in any language can be satisfactorily obtained in AMBIT/G without making any fundamental changes in the language. The facilities require, at a minimum, certain reserved node-boundary shapes (for integers, reals, etc.) and the usual collection of built-in arithmetic operations expressed as AMBIT/G functions. Beyond this, a good macro facility could

take responsibility for supplying convenient notations for algebraic expressions. We feel that it is not appropriate to introduce an arithmetic facility by, for example, introducing ALGOL 60 as an embedded or supplemental language to AMBIT/G.

5. Diagrammatic Macros. The introduction of a facility for two-dimensional macros operating over an AMBIT/G program is certainly the most difficult design problem confronting us. Conventional macro-expanders rely heavily on the structural simplicity of a symbol sequence, and restrict their operations to rather simply manipulations of that symbol sequence. An AMBIT/G macro-expander must operate on diagrams whose parts do not have a clear sequential ordering, and the way in which these operations should be restricted are not intuitively obvious. Since AMBIT/G is, itself, a diagram manipulation language, it might appear that an AMBIT/G macro should simply be a well-scoped AMBIT/G program which operates on the remainder of the program in which it appears; but such a scheme could easily loose the simplicity and clarity which conventional macro-expanders achieve precisely because their facilities are limited to simple manipulations of the program. This issue is under study at the present time.

The current state of the AMBIT/G design will be reported in a forthcoming Project Report [P2]. This report will contain a definition of AMBIT/G; but more important, it will try to establish a consistent set of design objectives which are the basis of the definition.

AMBIT/L. The AMBIT/L programming language is a specialization of AMBIT/G in which the node types are pre-defined rather than specified by the programmer. Specifically, AMBIT/L permits the use of the following ten types of nodes:

= The cell, which has two links and is never named. A cell node is used to structure the data; it is the only node with more than one link.

= The pointer, which has one link and always has a name. A pointer node is the usual means for access to the data.

= The basic symbol, which has no links and always has a single typographical symbol as its name. A basic symbol node is the usual means for representing the semantics (content) of the data.

= The mark, which is similar to the basic symbol except that its name is "internal"; that is, known only to the program. It is used to introduce semantics which cannot be confused with the semantics of the input data.

= The integer and the real, which each have no links and an integer number or real number as name. These nodes are used as convenience- and efficiency-oriented abbreviations for basic-symbol strings which represent numbers.

= The string and the token, which each have one link and a symbol string or a general (parenthesized) structure as name. These nodes are used in the construction of symbol tables (in the sense of compiler technology), and depend on concealed hashing techniques. They are to symbol table management what integer and real are to arithmetic.

= The function and label, which each have no links and a name identical to a function-name or label within the program. These nodes are used for indirect control references, and are the basis for a facility equivalent to the use of function and label variables.

The restriction of the data base to this set of node types is a rather straightforward modification to AMBIT/G; but the implications of the restriction, both for the user of the language and the implementer are important. AMBIT/L is a language with a fixed data base; it is therefore amenable to the development of a "school" of programming (techniques and disciplines) and to highly specialized optimization of implementation.

Work on AMBIT/L has become an integral part of the development of AMBIT/G. AMBIT/L is a diagrammatic pattern-replacement language like AMBIT/G; but it has a fixed data base like IPL-V, LISP, and SLIP. Thus AMBIT/L fills the gap between AMBIT/G and the well-known symbol manipulation languages. Where possible, design issues arising in AMBIT/G are first studied in the more conventional context of AMBIT/L.

The Graphics Interface for AMBIT/G. The design of the input-output graphics interface of AMBIT/G has made extensive use of transition diagrams for the definition of the interaction between the user and the graphic input-output devices. These transition diagrams will be included in a forthcoming Project Report [P5].

Task 3: The Implementation of the Programming System.

The objective of this task is to implement and document an experimental AMBIT/G programming system. This implementation will determine the feasibility of the design concepts of the language and will permit the application of the language to selected practical applications.

Both an AMBIT/G program and its data are represented in a diagrammatic form, a form which is a slight elaboration of the "directed graph" of graph theory. This form of representation contrasts with that of conventional high-level language, in which both program and data are represented as character strings. In order to accomodate the diagrammatic representation of AMBIT/G, the programming system must make use of graphic input-output facilities.

The user will supply input to the system by drawing a free-hand diagram directly on a RAND Tablet while he watches a CRT display on which the system maintains a version of the input which has been cleaned up and squared off. The use of conventional character-string input devices, such as teletype, will be avoided; and even the use of target button on the display will be avoided except where its effectiveness is superior to that of free-hand input. As nearly as possible, the user will sketch an AMBIT/G diagram as he would sketch it on paper and the system feedback on the CRT will return the same diagram in a diagram of publication quality.

Ideally, the output of a program would be controlled entirely by the program, and well-formatted, readable diagrams would appear on the CRT display automatically. This would correspond to the behavior of conventional programming systems, in which the composition of output is almost always controlled by format routines within the program.

In practice, however, it will often be necessary for the user to interact closely with the system to obtain acceptable output diagrams. Specifically, the result of running a program will often produce data in which some information about the composition of the diagrammatic representation of the data is absent. In such cases, the user will be called upon to supply the missing information about composition before the diagram can be presented on the CRT display. As a special but important case, the program output will contain no specification of composition at all, and the user will determine the entire layout of the output diagram by intensive interaction with the system.

This form of diagrammatic input-output requires work in some important new areas of computing. For input, we must have a character-recognizer of a rather general kind to correctly interpret the free-hand Rand Tablet strokes supplied by the user. For output, we must supply some facilities for automatically providing acceptable composition for output diagrams; and, where these fail, we must provide for user-system interaction to supply corrective and supplementary composition information.

These input-output requirements are typical of many applications of graphic computing. AMBIT/G diagrams are particularly appropriate objects for experimental development of graphic techniques because they embody complexity of structure with a minimum of irrelevant detail. An input-output system which can cope with the composition problems of AMBIT/G diagrams will include the basic facilities required for processing many other types of diagrams

We have started the implementation of AMBIT/G with the construction of a character recognizer for use with RAND Tablet input. This recognizer is now operational on the PDP-1 at Harvard University and will be described in two forthcoming Project Reports [P4, P5].

Once the recognizer has been initialized, it will accept a <u>sketched symbol</u> (a set of free-hand strokes anywhere on the RAND Tablet) as input and will respond with a corresponding <u>standard symbol</u> (a set of straight-line segments on the CRT display) as output. The initialization of the recognizer requires, first, the drafting

of the set of standard symbols, and second, the training of the recognizer until it responds to a sketched symbol with the correct standard symbol.

In drafting mode, the recognizer is used to build up a set of separate diagrams each of which is to be a standard symbol. The user is given a coarse grid on a scale much larger than that at which the standard symbol will ultimately be displayed, and he enters straight line segments which each connect two intersections of the grid. For example, the user might enter:

= three lines arranged to be a crude representation of the letter 'U'; or,

= 23 lines arranged to be a publication-quality representation of a 'U', with short lines to negotiate the curve, doubled-lines to widen the left arm, and additional lines to supply a serif at the top of each arm; or,

= four lines arranged in a rectangle of proporations 3 x 4 to be used as an AMBIT/G node boundary.

Because of the enlarged scale of the drafting grid, the user can easily enter portions of a symbol which will appear as fine details when the symbol is displayed at normal scale.

In training mode, the recognizer is taught by the user to associate with each sketched symbol a unique drafted symbol. This training is achieved by a sequence of executions of a training cycle. The following is an abbreviated description of the training cycle:

1. The system requests input of a sketched symbol;

2. The user draws on the RAND Tablet a sketched symbol which corresponds to some drafted symbol which he has (mentally) chosen as the subject of this training cycle;

3. The system consults its record of previous training cycles, outputs on the CRT display a standard symbol or a "don't know" message, and then requests evaluation of its input.

4. The user accepts or rejects the recognition by entering one of two sketched symbols reserved for this purpose;

5. If the user has accepted the recognition, the system records its success and goes to Step 1; otherwise, the system displays the complete set of drafted symbols and asks the user to select the symbol which is the correct recognition;

6. The user selects a symbol by pointing at it (via the RAND Tablet);

7. The system records its new lesson and goes to Step 1.

This description omits some interesting details of the recognizer which are important in practive if not in principle; a detailed description of the recognition strategy will be given in [P4].

The recognizer has commands for filing and accessing both sets of drafted symbols and the recognition data resulting from a training session. It is not necessary to start with an empty set of drafted nodes; the user may rather modity an existing set to suit his current needs and then train the recognizer to recognize his sketched symbols for the new drafted symbols.

The recognizer just described is now being incorporated in the general input processor for AMBIT/G. This processor will permit the user to draw in a diagram of an AMBIT/G rule or of AMBIT/G data in free-hand style, with no significant use of buttons or targets. Node boundaries and the individual characters of node names will be sketched in and immediately replaced by drafted symbols by the recognizer; links will be sketched in by the user and immediately smoothed and adjusted by a special purpose recognizer.

Task 4: Programming of Machine-Independent Software.

The objective of this task is to apply AMBIT/G to the programming of typical software components, such as those found in compilers and operating systems. This application of the language will test the effectiveness of the programming system resulting from the research, design, and implementation performed under this project.

During the first six months of the project, AMBIT/G has been used to program a formal executer for a large and complex language, BASEL; to program a portion of an interactive graphics system for the drafting of display character; and to program a variety of small but interesting algorithms. These applications are described in the following paragraphs.

The BASEL Definition. The BASEL Programming Language was developed as a part of the Extensible Language Project; it is the base language (BASEL) for an extensible language facility. To the extent that the "size" of a programming language can be discussed, BASEL is about the size of ALGOL 60; however, because it represents a concentration of the most complex and deeply embedded features of a high-level language and omits "syntactic sugaring", it is a complex and powerful language.

BASEL is of interest to the project under discussion here as a language whose definition presents serious notational difficulties. The data base of a BASEL program is not the simple association of variable names with scalars and arrays which is required by ALGOL 60; rather, it is a rather general form of directed-graph structure. Accordingly, the construction of a formal model for the BASEL programming system was adopted as the first major programming application of AMBIT/G.

The BASEL formal model consists of a compiler, which translates a BASEL program into an intermediate representation, and an interpreter which executes the intermediate representation of the program. In its present form, the compiler assumes that the program has been parsed according to a given BNF and contains an AMBIT/G

program which compiles any given phrase of BASEL into the intermediate representation. The interpreter is written entirely in AMBIT/G.

The BASEL formal model will be given in a forthcoming Project Report [P3]. This report has appropriate annotation of the AMBIT/G program and is a good example of large-scale AMBIT/G programming.

The Character Drafter Program. The character drafter program is an interactive graphics program which permits a user to enter a set of straight line segments which, collectively represent a picture of a single character or symbol. This diagram is filled and it later displayed, on a reduced scale, as the character recognizers response to a hand-drawn character.

The documentation of this program in AMBIT/G has provided us with an opportunity to apply AMBIT/G in the area of computer graphics. Since the character drafter will, itself, be a part of the final AMBIT/G implementation, it also contributes to the documentation of that system. As is often the case with documentation after implementation, the process of writing the AMBIT/G program revealed a variety of inconsistencies and redundancies in the design of the implementation.

The AMBIT/G program for the character drafter will appear as a part of the forthcoming Project Report [P5].

Other Programming Applications. Small scale applications of AMBIT/G have been made in the areas of sorting, storage management, multiple precision arithmetic, and list-processing utility subroutines. This collection of applications will be augmented as the project proceeds and finally published as a collection of short examples of AMBIT/G programming.

REPORTS IN PREPARATION

The following reports are currently being prepared for publication as Project Reports. Titles and abstracts are tentative and are subject to minor changes.

P1. Cheatham, T.E.,Jr., "Non-Deterministic Models for Syntactic Analysis"

Abstract. This paper is essentially Chapter IV of the set of notes for the course "Theory and Construction of Compilers" offered as Applied Mathematics 295 at Harvard University.

The intent of this Chapter is to first introduce three basic methods of syntactic analysis by using informal non-deterministic algorithms and choosing representations of the data and processing of it which results in the construction of a syntactic analysis tree which enhance understanding rather than suggest (machine) efficiency. We then formalize the data structures and manipulations as AMBIT/G data and programs. Following this we develop representations of the nodes and links as integer arrays and give an ALGOL-like program for performing the analysis.

P2. Christensen, Carlos. "A Description of AMBIT/G"

Abstract. This paper represents a working definition of the AMBIT/G programming language. The definition is presented in two forms: that of intuitive pattern matching and that of a formal matching algorithm.

The intuitive pattern matching corresponds to the gestalt phenomenon which occurs when a reader recognizes that two directed graphs are equivalent without having systematically compared them node by node and link by link. Since this phenomenon is the basis for the appeal of AMBIT/G, a definition in these terms cannot be omitted from a description of the language.

The formal matching algorithm represents the process which must be used when a pattern is too large or complex to permit the reliable application of intuitive pattern matching. It is a step-by-step comparison of the parts of two directed graphs to determine their equivalence. Since the more complicated facilities of AMBIT/G tend to loose the simplicity of pure pattern-matching, the formal matching algorithm is also essential to the description of AMBIT/G.

P3. Jorrand, Philippe. "The Formal Definition of BASEL"

Abstract. This document is a complete formal definition of the BASEL programming language. It is presented in the form of a program executer which consists of a compiler and an interpreter.

The compiler inputs the BASEL source program and translates it into a tree-like AMBIT/G data structure called the execution tree. The compiler consists of a program syntax written in a BNF-like notation and of a program written in AMBIT/G. A syntax analyzer is assumed which parses the BASEL program according to the program syntax and delivers phrases, one at a time, to the AMBIT/G program. The AMBIT/G program translates each phrase into an appropriate addition to the execution tree which it is building.

The interpreter inputs the execution tree and walks this tree performing the operations indicated at the nodes of the tree; this constitutes execution of the given program. The interpreter is written entirely in AMBIT/G.

In this way, the formal definition completely specifies the context-free aspects of BASEL (in the program syntax), the context sensitive aspects of BASEL (in the AMBIT/G phrase - translator) and the semantic aspects of BASEL (in the AMBIT/G interpreter).

P4. Ledeen, Ken. "An Adaptable Character Recognizer"

Abstract. The increasing availability of interactive computing systems and stylus devices has created a demand for a flexible and natural means of inputting non-standard graphics and two-dimensional forms.

A real-time character recognition scheme, that is, an algorithm for associating pen movements with a character code and a display form, has been designed and implemented for the Harvard University PDP-1 computer, Grafacon tablet, and CRT display. The tablet provides high resolution pen position information upon demand from the computer. This information is then processed by the recognition program. The program is "trained" by the user to recognize his individual printing style, and to display characters of his own design.

The character recognition scheme will be used to provide input to the AMBIT/G programming system, and has already been used to provide input to several other programs which are briefly described in this paper.

P5. Moskovites, Peter. "Two Notes on the AMBIT/G Character Recognizer"

Abstract. This paper describes two separate aspects of the character recognizer, one on the user/system interaction of the recognizer and the other on the implementation of that portion of the recognizer, the "drafter", which is used by the operator to design a new display character.

The user/system interaction of the recognizer is described largely by three transition diagrams. These represent the overall control of the recognizer, the control during the training of the recognizer, and the control during the drafting of new symbols. Also included are photographs of the CRT display at various typical states of the use of the recognizer.

The description of the implementation of the drafter is given in the form of an AMBIT/G program. This is of interest because it not only defines the implementation of the drafter but also represents a programming application of AMBIT/G.

BIBLIOGRAPHY

The following papers report on work done at Computer Associates, Inc. which is related to, but not part, of the current project.

1. Cheatham, T.E.,Jr. and Leonard, Gene F. "An Introduction to the CL-II Programming System", Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6311-0111, November, 1963.
   Also in:
   Rosen, Saul (Ed.) Programming Systems and Languages. New York: McGraw-Hill, 1967.

2. Cheatham, T.E., Jr. and Sattley, Kirk. "Syntax Directed Compiling", Proceedings of the AFIPS Spring Joing Computer Conference, Washington, D.C., April, 1964. Vol. 25, Baltimore: Spartan, 1964. pp. 31-57.
   Also in:
   Rosen, Saul (Ed.) Programming Systems and Languages. New York: Mc-Graw-Hill, 1967.

3. Cheatham, T.E., Jr. "The Theory and Construction of Compilers", Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6606-0111, June, 1966.

4. Cheatham, T.E., Jr., "The Introduction of Definitional Facilities Into Higher Level Programming Languages", Second Edition. Proceedings of the AFIPS Fall Joint Computer Conference, San Francisco, November 1966. Vol. 29, Washington, D.C.: Spartan, 1966. pp. 623-637.

5. Cheatham, T.E., Jr., "The Introduction of Definitional Facilities into Higher Level Programming Languages", A Talk Presented at the 1966 Fall Joint Computer Conference. Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6612-1512, December, 1966.

6. Cheatham, T.E., Jr., Fischer, Alice E. and Jorrand, Philippe. "On the Basis for ELF: An Extensible Language Facility", Proceedings of the AFIPS Fall Joint Computer Conference, San Francisco, California, December 1968. Washington, D.C.: Thompson, 1968. pp. 937-948.

7. Christensen, Carlos. "AMBIT: A Programming Language for Algebraic Symbol Manipulation", Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6410-1511, October, 1964.

8. Christensen, Carlos. "Examples of Symbol Manipulation in the AMBIT Programming Language", Proceedings of the ACM 20th National Conference, Cleveland, Ohio, August, 1965. New York: ACM, 1965. pp. 247-269.

9. Christensen, Carlos. "On the Implementation of AMBIT, A Language for Symbol Manipulation", Comm. ACM, 9 (August, 1966) pp. 570-573.

10. Christensen, Carlos and Mitchell, Robert W. "Reference Manual for the NICOL II Programming Language", First Edition. Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6701-2611, January, 1967.

11. Christensen, Carlos. "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language".
In:
Klerer, M. and Reinfelds, J. (Eds.) Interactive Systems for Experimental Applied Mathematics. New York: Academic Press, 1968.

12. Floyd, Robert W. "Non-Deterministic Algorithms", J. ACM, 14 (October, 1967) pp. 636-644.

13. Holt, Anatol W., Shapiro, Robert M., Saint, Harry and Warshall, Stephen. "Final Report for the Information System Theory Project", Applied Data Research, Inc., Princeton, New Jersey, February 1968.
Prepared for Rome Air Development Center, Rome, New York, Griffiss Air Force Base, New York, Contract AF30(602)-4211.

14. Jorrand, Philippe. "A Grammar for BASEL: An Example of the Use of the Interactive System Processor", Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6811-2111, November, 1968.

15. Leonard, Gene F. and Goodroe, John R. "An Environment for an Operating System", Proceedings of the ACM 19th National Conference, Philadelphia, Pennsylvania, 1964. New York: ACM, 1964. pp. E2.3-1 - E2.3-11.

16. Leonard, Gene F. and Goodroe, John R. "More on Extensible Machines", Comm. ACM, 9 (March, 1966) pp. 183-188.

17. Sattley, Kirk and Warshall, Stephen. "Specifications for an Automatic Operating and Scheduline Program", Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6201-0111, January, 1962.

18. Shapiro, Robert M. and Zand, Louis J. "A Description of the Compiler Generator System", Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6306-0112, June, 1963.

19. Shapiro, Robert M. and Warshall, Stephen. "A General-Purpose-Table-Driven Compiler", Proceedings of the AFIPS Spring Joint Computer Conference, Washington, D.C., April, 1964. Baltimore: Spartan, 1964 pp. 59-65.

20. Shapiro, Robert M. and Saint, Harry. "A New Approach to Optimization of Sequencing Decisions", Applied Data Research, Inc., Princeton, New Jersey, CA-6803-2411, March, 1968.

21. Warshall, Stephen. "A Syntax Directed Generator", Proceedings of the Eastern Joint Computer Conference, Washington, D.C., December, 1961, New York: Macmillan, 1961, pp. 295-306.

22. Warshall, Stephen. "Some Remarks on the Design of Multi-Processing Computer Systems", Massachusetts Computer Associates, Inc., Wakefield, Mass., CA-6304-0111, April, 1963.