NAS 12-563
CR 86301

Massachusetts
COMPUTER ASSOCIATES, Inc.

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

ON THE BASIS FOR ELF - AN EXTENSIBLE LANGUAGE FACILITY

by

T.E. Cheatham, Jr.

CA-6806-1311

June 13, 1968

The work reported in this paper was supported, in part, by the National Aeronautics and Space Administration under contract No. 12-563 and by the Department of the Army under contract No. DAAB 03-C-68-0064.

ABSTRACT

This paper provides a philosophical basis and motivation for ELF - an extensible language facility.

We introduce the subject by discussing the need for a variety of programming languages and exploring several alternative ways of providing for this variety. Following this, we present and discuss the overall design criteria which have provided the sources of constraint in the development of the extensible language facility. We then consider the facility from three points of view: as a language, from the point of view of compiling programs in the language, and from the point of view of the interface between the language and the system.

The remainder of the paper might best be characterized as an essay on programming languages. We consider a number of features of programming languages such as the basic data types and operations, control features, and so on. By contrasting the handling of these features in several current languages and by suggesting several generalizations as well as noting a number of lacunae, we motivate the particular handling of them in ELF.

PREFACE

For some time we have been working toward the development of a programming language facility which can be easily tailored to the needs of any particular group of users. The Extensible Language Facility - ELF for short - is the result of this work.

This paper is not, however, an introduction to ELF as such. Rather, our concern here is with providing a philosophical basis for ELF and for motivating the notions and mechanisms proposed for the programming language component of ELF. A companion paper [F1] provides an introduction to the programming language component of ELF. Programming languages and, particularly, complete programming systems have become ever more complex; ELF is no exception, although we do submit that it does not have the apparent complexity of PL/I or ALGOL-68. It is our hope that this paper will provide sufficient background and framework that the papers and documents which are concerned with the details of ELF as such are more easily read and comprehended.

There have been many sources of inspiration for ELF. The GPL language of Garwick (see [G1], [G2]), the ALGOL D language of Perlis and Galler (see [G3]), and the recent draft report on ALGOL-68 (see [V1]) have all had considerable influence on our work. We would also note that many of our colleagues have made significant contributions.

INTRODUCTION

There are two basic premises which underly the development of ELF. The first of these is that there exists a need for a wide variety of programming languages; indeed, our progress in the understanding and application of computers will demand an ever widening variety of languages. There are, in fact, "scientific" problems, "data processing" problems, "information retrieval" problems, "symbol manipulation" problems, "text handling" problems, and so on. From the point of view of a computer user who is working in one or more of these areas there are certain units of data with which he would like to transact and there are certain unit operations which he would like to perform on these data. The user will be able to make effective use of a computer only when the language facilities provided allow him to work toward a desired result in terms of data and operations which he chooses as being a natural representation of his conception of the problem solution. That is, it is not enough to have a language facility which is formally sufficient to allow the user to solve his problem; indeed, most available programming languages are, to within certain size limitations, universal languages. Rather, the facility must be natural for him to use in the solution of his particular problem.

The second basic premise underlying our work is that the environment in which programs are prepared, debugged, operated, documented, and maintained is changing and that the language facilities currently available do not properly reflect these changes. We are speaking, of course, of the advent of interactive computer systems which provide the user with computer-based files and which can provide for a much more intimate involvement with the development, debugging, and execution of a program than is possible with a batch system. A modern language must be developed with this kind of programming and operating environment in mind; although not everyone will be able to interact with a reasonable programming system in the immediate future, the trend is clear.

Let us now explore briefly the implications of these two premises and examine some alternative approaches to providing an appropriate language facility.

The "classical" approach to providing a large variety of languages has been that of developing languages and their translators - and often even their operating environments - independently. However, it seems clear that the cost of creating and maintaining an ever increasing number of language systems is not tolerable. Somehow we must both provide the variety of facilities but, at the same time, also reduce the number of different systems. It would seem that there are two extreme approaches to the problem of developing a language facility which provides all things to all men. We will refer to these as the shell approach and the core approach. The shell approach calls for the construction of one universal language which contains all the facilities required for every class of users. PL/I with the "compile-time" facility is probably the best current example of a shell language. In contrast, the core approach calls for the development of a small "core" language which, by itself, is probably not appropriate for any class of user, but which contains facilities for self-extension. A particular class of users then extends the core language to create a language which is appropriate for their problems. There are, to our knowledge, three current languages which are, to some extent, core languages: ALGOL-D, GPL, and ALGOL-68.

The shell approach does have a certain appeal. Just as the telephone company assures us that they have the potential of connecting us to any appropriately interfaced computer in the world within a few minutes for a modest fee, it would be a comfort to feel that all the language facilities we might desire were available and required only our calling them up and getting connected to them. Nonetheless, it is our opinion that the shell approach must be rejected. The problem is not as simple as that of connecting a few telephone lines. First, the overhead inherent in utilizing a shell language is rather large. That is, a user must carry as overhead all the mechanisms for language facilities which he is not using, implying an overhead cost measured both in time (for example, in looking for cases which don't occur in a program) and space (keeping the whole translator available "just in case"). Perhaps a more important difficulty is that if the PL/I philosophy of prescribing some meaning for most any construction is followed, the language designer is forced to devise numerous default conventions plus giving interpretations for all manner of operations on incompatible operands. In PL/I this requirement has led to such anomalies as: both of the boolean expressions $5 < 6 < 7$ and $7 < 6 < 5$, are true; the interpretation A*B where A and B are matrices is the matrix whose $(i,j)^{th}$ element is the product of the $(i,j)^{th}$ elements of A and B. It is not that these kinds of interpretations are "bad" - the point is that they are built-in and unchangeable. No matter what meaning one might like for $7 < 6 < 5$ (I like <u>false</u>) or for $(i,j)^{th}$ element of A*B, (I like the inner product of the i^{th} row of A and of the j^{th} column of B), that meaning provided by the designers is now fixed. One must revert to procedures if he wishes to introduce new operators, or to detour around the built-in operators when he needs to vary the meaning of those originally provided. And this becomes even more cumbersome when, as in PL/I, procedures can produce only scalar results. We would maintain that our reasons for rejecting the shell approach are not based on speculation; the difficulties currently being experienced with the implementation and utilization of <u>full</u> PL/I provide ample evidence.

Thus it is our contention that the most reasonable approach to providing the desired variety of language facilities is that of providing an extensible language supported by an appropriate compiling system. We do not, however, suggest that we can now devise a single universal core language which will adequately provide for the needs of the whole programming community; the diversity in "styles" of languages and translation mechanisms will probably always be sufficient to encourage several language facilities. ELF, which is the subject of this paper, provides a facility in the "style" of such languages as ALGOL-60, PL/I, and COBOL.

Now let us discuss the second premise, concerning the environment in which we envision programming being done. Our basic assumption here is that the programmer does not approach the computer with a deck of cards or magnetic tape which constitute a complete and independent run: a "run" deck which would commence with control cards, followed by his problem and then by his data, and which would result in the system accepting these, compiling his problem, running it against his data, and finally burying him in dumps or some other visible output. Rather, the programmer's unit transactions should be thought of as acts of updating some file. He might insert a few corrections to his program text, might call for some incremental change to some executable form of his program, and then might let his program run, all the while maintaining an intimate control over the proceedings, responding to messages as they occur instead of having to wait for the final results before he can exert any control.

We do not suggest that the ELF is a solution to the problem of providing a language for the effective use of a modern time-shared* system with permanent users' files. Indeed, there is really very little experience now accumulated in using such facilities, as most of the language facilities now in use on the available systems were developed as "batch" languages. It is to be hoped that work such as that now underway at Carnegie-Mellon under Perlis' direction will provide some guidance in this area [P2].

We do suggest, however, that we can now devise an extensible language facility in such a manner that it is cognizant of an available filing system and provides for interactive control; we will discuss our point of view on the relation of the language to the system in a later section.

---

* That is, interactive; how the intimacy between the user and the system is arranged does not concern us.

The remainder of this paper is divided into four sections. In the next section we will discuss the overall design criteria which have guided the development of the language. Following this, we will present an overview of the language with the object of providing the reader with a general feeling for the language as well as for the translating and executing mechanisms which we envision. Following this we will discuss the kinds of features and facilities which will be in the language; the purpose of this section is to justify and motivate the kinds of constructions proposed for ELF rather than to describe ELF as such. The final section is devoted to a summary and conclusion.

DESIGN CRITERIA

Perhaps the most eloquent defense of the overall design criteria to which we have tried to adhere was given in the 1966 Turing lecture by A.J. Perlis [P1]. There Perlis framed the problem as that of providing for systematic variability in a language. All acceptable languages provide for constant as well as for variable operand values. However, a great deal more variability must be provided if a language is to be extensible. There must be means of providing for variability in the types of quantities with which we deal, in the operations on these quantities, in programs or procedures, in the syntax of programs, in regimes of control, in the binding of programs to other programs and data, in the means of accessing data, in the employment of the various storage and input/output resources afforded by the system, and so on. However, we must provide for this variability very carefully so that we retain the necessary control over the efficiency of use of the computer, or else our result will be a purely academic exercise.

In our design of ELF we have looked to a number of "users" as sources of constraint; unless the language facility is properly matched to its users, it will not be an effective tool. These "users" include the programmers who will read and write in the language, the computer which will execute programs, the compiler or translator which will prepare executable programs, and the operating system which will provide the environment for the preparation and execution of programs. In addition, we feel that there are two other important sources of constraint: the traditions established by current languages, and the practicality of the language. Let us now briefly discuss the nature of the constraints which each of these various sources imposes.

Programmers

People have to learn and use the language. Indeed, we hope that people will even read programs in the language in addition to writing them. However, we find that different people have rather different ideas about the form in which a program should be cast. Most serious programmers adhere to the basic expression forms where these forms are appropriate - using the infix, prefix, and postfix operators plus parentheses which have resulted from the years of development of mathematical notation. The form of program text which is not inherently "expression-like" is, of course, not so well established. We note here, however, that the usual "out" for introducing new operations into a language - the use of functions or procedures - does not provide an adequate notation for the majority of operations. If the number of arguments required exceeds three or four, the user has difficulty in associating the "meaning" of an argument with its position in the argument list and he might be considerably better off with some keywords to help him focus on what is what. Also, if the nesting of function calls gets to be more than two or three deep, the "LISP-unreadability" problem becomes serious. We would also note that, for the user, an important criterion is that he should not have to introduce and deal with constructs which are unnecessary to the solution of his problems. The arithmetic expression form provides a facility which is both natural to a large class of users, and which also very effectively hides the setting up of temporary storage for intermediate results. Similarly the various renderings of McCarthy's conditional expressions as well as the iteration or looping facilities which appear in many programming languages have, as a secondary effect, that of eliminating the needs of introducing temporaries or lables which are used only once (see [D1] and [L2] for interesting discussions of this point).

Computers

The abilities of current and projected computers must also be viewed as a source of constraint. That is, we should try to "match" the basic types and operations in the language with those available in "standard" computers (and here we have reference to CPUs, not the whole "system"). Thus, for example, although our mathematical natures might encourage us to define only integer quantities and operations as primitive, and obtain floating point quantities and operations by extension, this would surely be foolish when we are faced with computers which by-and-large have floating point quantities and operations as primitives. Similarly, we reject the notion of quantities and operations drawn from set theory as primitive in the language because of the wide variety of implementation strategies which might be employed in providing for these. Such quantities and operations should be introduced via extensions. We must presume that the facilities available in current computers mirror, to some extent, the basic facilities which the users require.

Compilers

The past several years have witnessed the emergence of a considerable body of experience and technology in compiling programs. Unfortunately, most recent language developments seem to ignore this technology and demand new and ever more difficult and expensive translating mechanisms. We have attempted to reverse this trend and to adhere rather strictly to the technology available - to provide a language which can be effectively and efficiently translated and for which the known techniques of code generation and optimization will apply.

Operating Systems

The constraints which might be imposed by the peripheral devices and operating systems which are to be used must be noted. Thus, the means for encoding messages to and from the computer are rather strictly dependent upon the devices (and software) which are available; our adherence to a conventional string language with reasonably conventional characters is dictated by this consideration. The control structure inherent in modern computers must also be kept in mind; for example, the notion of "interrupt" is basic in most computer systems and our language facilities should reflect this. Further, the availability of various kinds of storage having varying degrees of accessibility plus the needs of the operating system to allocate the storage and other resources of the system must not be ignored.

Tradition

The "tradition" which has been established by such languages as ALGOL-60, PL/I, COBOL, and LISP and which is being established by ALGOL-68, GPL, and ALGOL-D should be considered as a source of constraint. That is, it does not seem reasonable to re-invent and re-cast the facilities available in those languages just to be different. Our departures from the facilities available there should be well thought out and well justified. It will be clear that we have in fact departed in more-or-less significant ways from all these languages; we hope that our arguments for doing this are convincing.

Practicability

The final source of constraint which we have tried to observe is that of practicability. It is our intention that the language be as efficient and useable as any of the conventional programming languages. In adhering to this constraint we have failed in many ways to reach all the goals of variability which Perlis prescribed. Thus, ELF provides a language which has the kinds of variability which we can imagine being handled with reasonable efficiency. Another generation of language development will be desirable when we better understand other kinds of variability and can devise mechanisms for handling them efficiently.

OVERVIEW OF THE EXTENSIBLE LANGUAGE

There are a number of vantage points from which one can view the extensible language. In this section we will look at the language from three points of view. First, we will take the conventional view, looking at the language as providing various types of data and operations, and various ways of writing about these. Following this we will consider it from the point of view of compiling programs written in it. We feel that an understanding of the language from this point of view is helpful if one is going to understand the various extension mechanisms which are available in the language. The final point of view we will take is that of the interface between the language and the operating environment.

Conventional View

We can think of the language as containing a "base" component and an "extension" component. The base component is rather similar to conventional programming languages in that it provides for the declaration of operands of a number of basic data types and the construction of ordinary expressions over the various types of operands. In addition there are the usual assignment and control statements, including <u>go</u> <u>to</u> and iteration statements plus conditional expressions, conditional variable references, and conditional statements. The declarations and imperatives are all imbedded in a block structure which is similar to that of ALGOL-60. The differences between the base component and, say, ALGOL-60 are not dramatic save in certain notations used and the fact that there is an operating system and a filing facility in the background.

It is the extension component which departs from most conventional programming languages; there are, however, some strong similarities here with ALGOL-D, GPL, and ALGOL-68. Basically, one can introduce new data types, new operations, and new forms of writing (that is, new syntactic structure) into the language. New operations can be defined over new data types in terms of previously defined operations over the components of these types. New forms of writing may be introduced either by introducing new formats for operators or by what is essentially a combined lexical and syntactic macro facility (see [C1]). The new data types are created by the operations of constructing n-tuples (with named components), constructing rows (with numbered components), and constructing pointers, all recursively. Further, one can gain control over the interpretations of certain constructions which effectively permit rather more context than a single operation to be taken into account in determining the way in which some particular operation is to be "coded". As with ALGOL-D this facility permits one to carefully control the employment of temporaries and other such space/time trade-offs when generating code for such things as matrix operations.

In the next section we will discuss the elements of the language in somewhat more detail; for the moment it will suffice to think of the language as similar to ALGOL-60 but with provisions for new data types and operations (or, if ALGOL-68 is familiar to you, similar to ALGOL-68).

Compilation View

The second point of view we want to explore is that of the compiling mechanism which we have in mind. Although one does not conventionally talk about compiling techniques in describing a language, we believe that it is rather important in this case. That is, we have been strongly influenced in our choice of language constructs, notations, and mechanisms by what we feel can be readily handled by the current compiling technology and thus understanding our view of the kinds of compiling mechanisms envisioned is rather important. For present purposes we want to think of the compiler for the language as consisting of several "components", including: a lexical analyzer, a syntactic analyzer, a parse interpreter, and a user controlled optimizer, plus other components for generating machine code and filing it, or for interpretively executing some "internal" representation of the program text, and so on. We shall have no particular interest in these latter components here; let us now consider the other components.

Lexical Analyzer

The lexical analyzer will be responsible for isolating, identifying, and appropriately converting the source input (e.g. typed characters) thus producing a stream of "token descriptors" representing constructs at the level of "identifier", "literal", "operator", "delimiter", and so on. We anticipate that, although the lexical analyzer will be "table driven" by tables derived from a grammar which specifies the structure of the tokens, this component will not be changed or extended by the average user and we will thus think of it as fixed.

Syntactic Analyzer

We intend that the syntactic analyzer be essentially an operator precedence analyzer. An operator precedence analyzer is, of course, one of the simplest and most efficient kinds of syntactic analyzers available. Operator precedence analysis works only on a rather restricted set of languages. However, as Floyd demonstrated in his original paper on this method [F2], ALGOL-60 is close to being an operator precedence language; further, those changes Floyd proposed to the original syntax rules for ALGOL-60 and to certain constructions in the language in order to make it operator precedence did no real violence to the language but actually made it cleaner and more symmetric. Thus, a language does not necessarily suffer in richness of style because it was designed with this method of analysis in mind. Another important reason for the choice of this method is that those properties of the operators which, properly encoded, are required to "drive" such an analyzer are exactly the properties which the user has in mind when he specifies an operator, namely, the precedence, in the sense of order of evaluation of that operator relative to other operators.

It will be convenient to think of the operators available in the language as including binary infix (e.g. '+' or '<'), unary prefix (e.g. '-'), unary suffix (e.g. '!'), unary outfix (e.g. '| ... |'), n-ary "distributed" (e.g. '<u>if</u> ... <u>then</u> ... <u>else</u>' or '<u>increment</u> ... <u>by</u> ...'), and "functional" (e.g. 'MAX (...,...)' or 'SIN( )'). Each operator (actually each fixed "part" of each operator) will enjoy one of four relations with respect to all other operators (or parts of operators), namely: takes precedence, yields precedence, has equal precedence, or none. The user will introduce a new operator (syntactically) by specifying the precedence of each of its parts relative to the precedence of operators already available. It will generally be the case that a given operator will be defined for operands of a variety of data types; the "syntactic analyzer" will isolate a phrase - an operator plus its operands - by using the precedence relations, and then the <u>parse interpreter</u> will then determine the "meaning" or "interpretation" of the phrase in accordance with specifications which are either built-in (e.g. with '+' operating on two integers) or supplied as extensions by the user (e.g. with '+' operating on two quaternions). One of the "dispositions" which the parse interpreter might make of some phrase is to place the operands of that phrase into some previously given (macro) "skeleton" and re-submit the resulting text for syntactic analysis. This will provide what are essentially the "lexical macro" and "syntactic macro" facilities proposed in [C1].

There are certain operators which require a larger context than the phrase in which they occur for their interpretation, particularly if one has a goal of producing optimal coding and either does not have, or prefers not to overburden, a code optimizer. An example here would be the coding of the multiplication of two conformable matrices in the three contexts:

A * C   (A + B) * C   (A + B) * (C + D)

Thus, it may be that one might desire an algorithm for matrix multiplication which required only one temporary scalar for the first case, a temporary row for the second, and a full temporary matrix only for the third. On the other hand, one might use temporary rows for all three cases, giving up storage efficiency in the first case and computation efficiency in the third. The point is that there are cases in which the determination of the appropriate means for performing some operation depends upon some context. The <u>user controlled optimization</u> phase provides for this. It would also be in this stage that the user would have the ability to tinker with such things as the allocation of storage, the means of access (e.g. via some hardware or software "paging" scheme) to certain quantities, and so on.

Briefly, we think of the parse interpreter as constructing what in effect is a computation tree representation of the analyzed program text; each node of this tree would be labelled with the data type of the value it represents. The <u>user controlled optimizer</u> may then be thought of as a mechanism which "walks" over this computation tree, inspects context as appropriate, and re-organizes and re-constructs portions of this tree. The mechanism has certain similarities with those proposed in ALGOL-D, but with a control and sequencing strategy similar to that of the GSL component of the CGS system (see [S1], [S2], and [W2]).

Interface View

Now let us briefly consider the language from the point of view of its interaction with the environment in which programs are constructed, debugged, and executed. First, we want to emphasize that we would expect the language to include the means for the kinds of communication with the operating environment which are typically handled via "control" or "job" statements as well as the kinds of communications which have to do with "editing". We presume that there is a filing system which contains such things as: (1) program text which we might want to incorporate into the input stream to the compiler during some run, (2) specifications of modes of programs or procedures which our current program might want to reference, (3) modifications or extensions to the compiler which we might want incorporated for processing our current program text, (4) an "executor" which can execute programs as they are represented following the interpretation of the parse and user-controlled optimization, (5) data which has been previously input or generated and then filed, (6) and so on. Clearly there must also be means for placing any of these items in the filing system. Thus, a "run" or "session" might be one in which we input a number of extensions to the language including new data types and operations over them, with the result that they are filed in such a fashion that we can later call the compiler, mentioning that it is to include these extensions. Another type of session might be the input of program statements with the expectation that they be executed directly. Another might be the input and editing of a program with the expectation of filing it for later execution. That is, a "unit transaction" within the ELF system will typically utilize material previously developed and filed, and result in material to be filed. There will be a number of forms which this material might take, ranging from text at one extreme to modifications to compiler tables or certain programs within the compiler at the other. So long as this philosophy of operation is understood, we will not go into further details here; we intend to spell out more details of the linguistic forms and possible system mechanisms to attend to problems in this area in a subsequent paper.

THE BASE LANGUAGE

In this section we will discuss a number of concepts and mechanisms which are relevant in a general purpose programming language and to attempt to motivate the particular realizations which have been made of them in ELF. It is not our purpose to discuss ELF as such, and certainly not to provide an introduction or primer for the language. Rather, we hope that our discussion here will, as it were, "soften the blow" and make the specifics of ELF which might otherwise appear to be rather strange departures from convention appear to be more reasonable (or, perhaps, less capricious). Thus, our hope is to suggest the kinds and means of variability which ELF will possess; our method will be to discuss inadequacies in current languages and, mainly by example, to show how they might be removed. One must turn to [F1] for the introduction to ELF as such. We might also note that we are not here attempting to give a complete justification for all the concepts and notions in the language. Indeed, perhaps the best view of the sequel is as a series of remarks on languages and language design.

One of the first issues we would like to discuss is related to the ambiguity of a variable name in most programming languages. That is, it requires a certain amount of context to specify the meanings of the three occurrences of 'X' in the ALGOL-60 fragments

```
X := X + 1          and          F (..., X, ...)
```

Of course we all know that the first and second 'X's mean the location in which a value is stored, and the value stored at that location, respectively. The ':=' and the '+' provide sufficient context to determine which of the two possible meanings is to be taken. The third 'X' is more troublesome. Without the description of the procedure named 'F' we simple have no idea whether the place or the value is meant. There is a further source of difficulty with "names" and "values" when we wish to pass to a procedure what amounts to the address of some quantity so that the procedure might store a new value into the location associated with it as well as to fetch and use its current value. Of course ALGOL-60 has the provision for name and value parameters of call; the trouble arises from the fact that ALGOL-60 really has no provision for passage of an address only, thus, for example, denying any direct and efficient general procedure which exchanges two values.

ALGOL-68 deals with this problem rather directly. In ALGOL-68 one declares the referential level" of each identifier, as for example in

```
real PI = 3.14159265;
ref real X;
ref ref real P:
proc ( ) ref real N;
```

Here PI is declared as a literal (real) value (and, like any literal, cannot appear on the left hand side of an assignment); X is declared as a reference to a real, that is, a variable - a place to store a real; P is declared as a reference to a reference to a real; a place to store the address of (or a pointer to, etc.) a real variable; and, N is declared to be a procedure which takes no arguments but produces as value a place to store a real. Passing arguments of the types real, ref real, and proc ( ) ref real are equivalent, in ALGOL-68, to what is usually called value, address, and name call. ALGOL-68 goes on to include as a basic feature of the language the automatic adjustment of referential levels. That is, appearances of X and P in arithmetic expressions and assignments would be attended by the equivalent of the insertion of the prefix operator 'val', interpreted as "take the value of", whenever the referential levels of quantities under various operations do not match. Thus, given the assignment

```
P := X + PI;
```

ALGOL-68 would automatically adjust this to

```
val P := val X + PI;
```

and, given

```
P := X;
```

would provide no adjustment since none is required; the variable P would receive as value the address of X. One of our more serious objections to ALGOL-68 is this automatic adjustment of referential levels. An alternative way to handle the adjustment of referential levels would be to insist that the user specify all such adjustments by extensions of the meanings of various operations.

Another feature of ALGOL-68 which we find controversial is its handling of the allocation of space. In the above example, no space would be allocated for PI, a space appropriate for storing a real would be allocated for X, and two space, one appropriate for storing a real and one appropriate for storing the address of the first space would be allocated for P.

It might be more desirable to arrange that there be no automatic allocation of space save that required to accomodate the "highest level" portion of a quantity declared, so that declaration of variables of data types "real variable" and "pointer to a real variable" would cause allocation of space adequate for storing a real and space adequate for storing a pointer to a real, respectively, leaving it to the user to obtain any further space he desired. We note that one good feature of ALGOL-68 in distinct contrast to PL/I, is that ALGOL-68 does not generally deal with pointers or addresses which point to (address) entities of unknown types; rather, one knows exactly what is pointed to in each case, an advantage which avoids many of the anomalies of pointers in PL/I.

It is possible to define several primitive data types or modes from which all other data types or modes are constructed via extensions. One set which corresponds well with conventional usage and conventional computing machines is the following:

Mode	Examples		Meaning
int	1	2	integer value
real	1.5E6	3.14159265	real (or, floating) value
bool	true		boolean value
char	'a'		character value
label	L		label value
mode	real		mode (or, data type) value

It might also be worthwhile to introduce a further primitive mode, say 'none', to describe those constructions, such as control transfers, which have no "value" in the conventional sense. Certainly one wants procedures which deliver no value. However, we would not want to specify that a procedure take as an argument a "quantity" with no value; we might argue that compile time insertion of a statement in place of some parameter would be useful, but at run-time one should expect to pass the name of a procedure rather than the procedure itself as an argument since one would not want the procedure carried out as the arguments of call are processed, but rather, when it is referenced in the body. That is, the mode 'none' is unlike the other (value) modes in that one cannot specify that an argument to a procedure have mode 'none'.

There is one further construct which might be introduced as a primitive; that is the tuple. It is clear that we want as a primitive notion that of an argument list for a function which the programmer wishes to reference by the conventional "functional notation".

When talking about the various quantities which can be manipulated in a language it will be convenient to speak of them as a triple - the name (designation, etc.), the mode (data type, etc.) of the quantity, and the meaning (interpretation, referent ,etc.); herein we will often display such triples in the form {name | mode | meaning} as for example:

{1 | int | 1}

{pi | real | 3.14159265}

{L | label | some particular point in a program}

{int | mode | built-in notion of integer value}

{X | loc real | a location in which a real can be stored, a real variable}

We might note here that we use the operator 'loc' in much the say way 'ref' is understood in ALGOL-68; we use the word loc rather than ref to emphasize the fact that no automatic adjustment of referential level is implied as it is in ALGOL-68.

The notion of expressions which have values is included in most languages. In general, these are constructed by using various operators (binary infix, unary suffix and prefix, n-ary distributed, and functional form) and their associated operands; the order of evaluation is determined by the relative precedence of the operators and further controlled by the use of parentheses for grouping. Some examples are

Expression	Mode
(1 + 2) * 3	int
(1 < 2) ∧ true	bool
sin (.5)	real
go to L;	none

The operators which we might view as primitive would include the ordinary arithmetic operations over ints and reals, the boolean operators over bools, and the relations which take ints or reals or chars and produce bools.

In most languages, there is no notion of mode valued expression or mode operator except that which is implicit in such constructions as

```
array HENRY [1:N]          of ALGOL-60
DCL 1 A,
      2 A1  INTEGER,       of PL/I
      2 A2  FLOAT,
        etc.
```

It would be much more consistent to treat modes as values - albeit "values" which indicate structure or meaning rather than the more conventional view of value as represented by numbers, characters, and so on. Indeed, one can define various operations which take mode valued operands and result in mode values, as well as define procedures which deliver modes as results. A completely symmetric treatment of modes as values would include a notion of mode valued variables. However, our adherence to a principle of "practicability" dictates a very careful control of the kinds of variability permitted in the mode of a quantity. Thus, as is the case with most programming languages, it is in this area that we would propose to carefully restrict variability.

There are three basic mode operators which we might remark upon briefly. The first of these was hinted at above; this is 'loc' operator, a unary prefix operator which takes a mode expression as operand and is interpreted as "create a space of the appropriate size and shape to store a ...". A mode expression of the form

loc ...

in the context of providing for the declaration of some quantity would indicate that the quantity is an address of (pointer to, etc.) a '...' .

The second operator is one which would permit the construction of a homogeneous collection of somethings. It is a binary operator whose operands are a mode and an integer value. The integer value is the number of somethings and the mode value is the mode of each something. The binding of the value of the integer operand could, of course, be expected to be accomplished at several different times including compile time, block entry time, or even dynamically. The use of this kind of mode operator would permit (by recursive use) the modelling of arrays, lists, and the like.

The third mode operator would be an n-ary operator which would permit the specification of n-tuples of (generally) non-homogeneous components.

With the "row" and "n-tuple" operators we would also have to provide some means for selecting the components as well as for constructing instances of structures created using these operators. The selection of a component of a row would doubtless be via its number (subscript); it might be more convenient to name rather than number the components of an n-tuple, however.

With the use of these three mode operators we might then construct things like the following:

{complex | mode | a pair of reals named 'r' and 'i'}

{vector | mode | a row of 10 ints named '1'. '2', ..., '10'}

{U | loc complex | the location of a place of the appropriate shape and size to store a pair of reals; the two components of U might be accessed via the names 'U.r' and 'U.i' or by the forms 'r of U' and 'i of U', etc.}

{V | loc vector | the location of a place of the appropriate shape and size to store 10 ints; the 1st, 2nd, etc. components of V might be accessed by the names 'V.1', 'V.2', etc. or the forms '1 of V', '2 of V', etc. or the form 'V[1]', 'V[2]', etc.}

As for constructing instances we might imagine such things as

```
complex (1.0, 2.0)
row (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

which would permit assignment statements like

```
complex (1,0, 2.0)  →  U ;
row (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  →  V  ;
```

Most programming languages have some form of 'if ... then ... else' or "conditional" operator. ALGOL-60 permits both conditional expressions and conditional statements, but does not permit conditional references (of, if you prefer, conditional "left-hand sides"). We submit that the conditional operator ought to be included in a language and to permit then and else clauses of any allowable mode, so long as the then clause and the else clause have the same mode (unless one has extended the meaning to cover the case of certain different modes).

There are some questions concerning the assignment operator and its interpretation in a programming language. Most programming languages use right to left substitution and there is considerable variation in the treatment of multiple assignment and imbedded assignments. We submit, first, that a left to right substitution is more "natural" and avoids the usual difficulties of explaining the order of evaluation of left- and right-hand sides. Thus, we might employ the operator '→' and write assignments like

```
0.0  →  X
pi * sin(.5)  →  X
I  →  V[I]
     etc.
```

An interesting interpretation of '→' as an operator is that it has a value, which is the value of its left-hand side. One might then view the ';' as a suffix operator which has the effect of "throwing away a value". This would clarify such constructions as

```
0.0 → V[1 → I];
```

Whether or not one prefers left-to-right or right-to-left assignments it does seem reasonable to take the assignment operator as having a value and to define the operator as "built-in" when occurring between an operand which is a value and an operand which is a location of a size and shape appropriate to store that value; any other interpretations could be entered via extensions. For example, in ALGOL-68, the assignment

```
P := X
```

where P is a ref ref real and X is a ref real is defined to result in P "pointing to" X; a case could be made, however, for interpreting the meaning as causing the place P points to to receive the value currently in the location corresponding to X. Our point is that user control over the interpretation should be possible.

Most modern programming languages have a notion of compound statement and/or block. Usually a "block" is a compound statement with declarations; the identifiers introduced are local to the block and allocation and freeing of whatever space is required by virtue of the declarations is arranged upon entry to and exit from the block. However, few have any notion of a compound or block expression, except for the case where the block constitutes a procedure body. It seems to us an obvious generalization to permit blocks and/or compound statements in general to have as value a value of any mode and to permit their inclusion wherever such values might be permitted. Taking curly brackets as "compound expression delimiters" we might have an expression like

```
{S(L), L-1 → L;} * {S(L), L-1 → L;} → {L+1 → L; S(L)}
```

which does something like popping two values off a stack, multiplying them, and pushing the result onto the stack.

We would also note that tying all allocation and freeing of space to block entry and exit does not generally provide the user sufficient control over his allocation and that something like the AUTOMATIC, CONTROLLED, and STATIC allocation attribute of PL/I is required.

There are a number of difficulties with the way in which procedures are handled in most programming languages. A procedure is nothing but a parametrically dependent value. That is, it requires certain parameters or arguments, each of a certain mode, and produces a value of a certain mode. The mode of a procedure might be indicated as

proc (real) real	procedure taking a real argument and delivering a real result, such as a trigonometric function.
proc (int, int) mode	a procedure taking two int arguments and delivering a mode valued result, such as, for example, 'row M of row N of real'.

The definition of a particular procedure requires that (1) the mode of the procedure (in the above sense) be established, and that (2) one indicate the formal parameters which, in the body of the procedure, act as place-holders for the actual parameters which will be delivered upon the call of the procedures, and (3) one provide the body of the procedure. The body generally is an expression involving the formal parameters plus, perhaps, other data such as literals, variables local to the procedure, and variables in some containing block. A possible notation for this defining a procedure, somewhat reminiscent of the λ-calculus is suggested by:

proc (int i, int j) $((i + j) \uparrow 2)$

which is taken to be the definition of a procedure which takes two int valued arguments and (by virtue of the fact that the expression following the argument list has an int value), returns an int value, the square of their sum. There are two contexts in which we might find a procedure body.

First, being essentially a λ-expression it could be applied to two arguments, viz:

```
proc (int i, int j)  ((i + j) ↑ 2)  [1,2]
```

resulting in ' 9 '. A more conventional (in programming languages) use would be in the context of a procedure name declaration like

```
let P be proc (int i, int j)  ((i + j) ↑ 2);
```

which establishes P as the name of a procedure which takes two int arguments and returns the square of their sum as its int result. By insisting upon explicit indication of the mode of a procdure (P would have the mode proc (int, int) int) one can separate the declaration of a procedure name and the assignment of a value to it, as suggested by:

```
let TRIG be a loc proc (real) real;
      :
      SIN  →   TRIG;
      :
(val TRIG) [.5]  →  X;
```

The assignment of SIN to TRIG is clearly acceptable since SIN has mode proc (real) real and TRIG was declared to be a loc proc (real) real, that is, a location of the shape and size appropriate for storing a proc (real) real. The value of a loc anything is that anything, here the SIN procedure which duly takes its argument and produces SIN (.5).

There is another area of confusion as regards procedures in most conventional programming languages. Generally, the user is permitted to introduce and use procedures only by adhering to the "procedure name followed by parenthesized argument list" form. We would argue that other forms should be permitted, particularly those in which the procedure name looks like an operator and is employed in suffix, infix, or prefix fashion, or, like the "if ... then ... else ..." operator, with its "name" spread out among its operands. So long as we can parse program text and relate an operator and its operands, the "free form" should surely be permitted and treated just like any procedure call.

Again it is generally not possible in most programming languages to utilize the same procedure form with different modes of arguments except in some special cases like the arithmetic operators which generally accept either <u>int</u> or <u>real</u> operands, or in the case of PL/I where "anything goes" and conversion to some canonical mode is arranged for automatically. However, the mechanism to permit the same procedure form to enjoy different interpretations depending on the data type of its operands is really quite simple; if one is to be permitted new data types he should surely be permitted the pleasure of defining meaning for old friends like '+' for these new modes of operands.

While we are on the subject of operators and procedures, we would note that typically one can control the order of execution of his operations through the precedence of each operator, plus the use of parentheses to force precedence. In general one cannot, however, say anything about the order of evaluation of the operands of an operator; that is built in as left-to-right, or as being in parallel, and so on, and is not subject to change or control by the user. Faced with the newer computing machines and/or operating systems which have some provision for parallel operations at the arithmetic, storage access, or program execution level, it would seem that we ought to provide language features which take some advantage of these. One way to do this would be to permit the user to specify for each operator the permissible order of evaluation of its operands. Thus, one might indicate that with most of the arithmetic operators parallel evaluation of their operands is permitted, that the operands of '→' or ':=' are left-to-right or right-to-left, and so on. However, if left-to-right order of evaluation was important, as it would be in an expression with "side effects" (such as is the case with stack popping and pushing) then one could stipulate that the left-to-right order be preserved. For operators at the "statement-level" such as ';' or '<u>begin</u>' and so on, such control would amount to indicating sequential or parallel operation of parts of programs.

Most current programming languages have some kind of provision for iteration or repeated execution of a compound statement. Typically one can specify that some (integer) variable takes on some sequence of values specified either by displaying the sequence of values or by indicating an initial and final value and an increment. Further, there is sometimes a provision for stopping the iteration at any cycle via a while clause which tests for the truth of some boolean expression. We would like to discuss several extensions of the conventional iteration statement which we think are justifiable constructions in a basic language (i.e. not obtained via extensions). First, if we are to have some notion like that of row discussed above and some means for constructing a row whose number of components might vary then we need some kind of language feature which will permit this. The generalization of iteration facilities suggested by the following might be considered.

for i = 1 to N keep i

Here we hope to suggest that N int values, specifically (1, 2, 3, ..., N) are constructed. These could then be combined into a row by use of a row-constructing function.

There are a couple of other generalizations of iterations which, if they were to be defined as extensions, would require introducing either unwanted labels or temporary storage, both of which are objectionable since it takes a fairly clever optimizer to detect the superfluity of such constructs. First, we might envision adding a variant of the while clause; the while clause of course simply stops the iteration when it becomes false. An interesting variant which has deep roots in mathematical usage might be called a such that clause, a boolean expression which simply acts like a filter, letting some values through and "skipping" the cycle for others. Second, we might note that with the 'do' and 'keep' variations we can iterate statements and we can produce a collection of values; the one thing we can't do simply is to produce a single value, for instance the bool desirable from a programming language equivalent of a predicate like

$$\forall\ i\ (A_i = B_i)$$

That is, we want to cycle through some index set making a test; the success or failure of that test might trigger both stopping the iteration (which can be handled by a while clause) and returning a value. One possible construction would be a clause which we might call the exit value expression, which would be associated with some part of the iteration specification: if that part of the iteration specification caused termination of the loop, then the value of the exit value expression would be taken as the value of the iteration statement as a whole. The predicate above might then be rendered as:

for i from 1 to N exit value (true) while ($A_i = B_i$) exit value (false)

That is, the construction will have the value true if the loop terminates because it has been done N times and false if the loop stops because the while clause fails.

In general, the facilities available in most languages for any kind of control operations other than simple go to statements, conditionals, and loops is rather meager. Indeed, most languages do not permit the use of the control features available on most machines such as interrupt, interrogation of the status of various devices, and so on. PL/I, of course, made a significant step in this direction by introducing tasks and the ON facility. We submit that considerably more must be provided in this area. One must be able to call procedures with a variety of relationships - subroutine, co-routine, independent task, and so on - and also to obtain and make use of various kinds of status information.

The final question we would like to discuss is that of the binding of various quantities such as the extents of arrays the values of variables, and so on. In ALGOL-60 like languages it has been customary to think of binding as occurring at one of four times: compile time, load time, block entry time, or execute time. There are good and practical reasons for distinguishing such times for potential binding; our argument with the conventional approach is that the things which can be bound at these various times are too rigidly fixed. Thus, in most languages one can only bind or fix the data type of a quantity at compile time; while this is a perfectly reasonable restriction for many programs it is completely inadequate for a general purpose output routine. That is, one should be able to write an output routine competent to deal with any type of quantity, charging it with the responsibility for determining the type of any quantity submitted for output and switching to the appropriate conversion and formatting sub-routines. Requiring a user to supply some data type code as a parameter is not an adequate solution. ALGOL-68 has an interesting kind of compromise solution to this problem. In ALGOL-68 one can specify that the mode of some quantity is to be one of a fixed set of modes, and that the particular one of these modes that is has at any time may vary dynamically as assignments are made to the quantity. It would appear that we could go somewhat further than ALGOL-68, however, particularly in permitting variability (i.e. deferred binding) in arguments to procedures.

In a similar vein, it is generally the case that a procedure is bound to all of its actual parameters dynamically at the point of its call; there is no provision, for example, to specify that it be bound to certain parameters at compile time or block entry time, a provision which, implemented properly, could result in much more efficient programs. Also, most languages basically have two ways of inserting procedures in a program - in line and as subroutines. However it is seldom the case that the user has any control over which operators or procedures are handled in which manner. This aspect of binding should clearly be brought more under the users' control; the "macro" and " subroutines" specification permitted in GPL are certainly steps in the right direction.

A related kind of restriction is that of insisting that the way in which arguments are made accessible to a procedure adhere strictly to the mode declared for the corresponding formal parameter. For example, if a procedure is to take a matrix as an actual parameter and we want to insure that the procedure treats the matrix as a value, then we either have to pass a copy of the matrix or pass an address (or dope vector) and hope that the procedure does not directly or indirectly affect the value of the matrix. Surely considerably more can be done here.

One kind of provision which would be useful is to allow specifying two modes for certain parameters; the mode which will be passed and the mode which the procedure is to presume. Thus, we might indicate in the matrix case that a matrix was to be passed by address but to be treated as a value (i.e. not converted to the desired mode but treated as though it were of the desired mode and providing for the fact that it is not). We might note here that given proper system support, facilities for dealing with the whole notion of custody and responsibility as described in [L1] should be available at the programming language level.

REFERENCES

C1 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.

D1 Dijkstra, Edsger W. Letter to the Editor. Communications of the ACM, Vol. 11 (March 1968) pp. 147-148.

F1 Fischer, Alice E. and Jorrand, Philippe. BASEL: The Base Language for an Extensible Language Facility. To be presented at the 1968 FJCC.

F2 Floyd, Robert W. Syntactic Analysis and Operator Precedence. Journal of the ACM, 10 (July 1963) pp. 316-333.

G1 Garwick, J.V., Bell, J.R., and Krider, L.D. The GPL Language. Control Data Corporation, Palo Alto, California, Programming Technical Report TER-05, 1967.

G2 Garwick, J.V. A General Purpose Language (GPL). Forsvarets Forsknings-institut, Norwegian Defence Research Establishment, Intern Report S-32.

G3 Galler, B.A., and Perlis A.J. A Proposal for Definitions in ALGOL. Communications of the ACM, Vol. 10 (April 1967) pp. 204-219.

L1 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.

L2 Landin, P.J. The Next 700 Programming Languages. Communications of the ACM, Vol. 9 (March 1966) pp. 157-166.

P1 Perlis, A.J. The Synthesis of Algorithmic Systems. First ACM Turing Lecture. Journal of the ACM, Vol. 14 (January 1967) pp. 1-9.

P2 Perlis, A.J. Private Communication.

S1 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.

S2 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.

V1 Van Wijngaarden, A., Mailloux, B.J., and Peck, J.E.L. A Draft Proposal for the Algorithmic Language ALGOL 68. IFIP Working Group 2.1, MR 92, January 1968.

W1 Wirth, N. and Weber, H. EULER: A Generalization of ALGOL and its Formal Definition: PART I. Communications of the ACM, Vol. 9, (January 1966) pp. 3-9.
PART II. Communications of the ACM, Vol. 9, (February 1966) pp. 89-99.