1-1-1977

Programming Language Requirements for Human Communication Structures or Computer Conferencing

Computerized Conferencing & Communications Center

Peter Gordon Anderson
New Jersey Institute of Technology

COMPUTERIZED CONFERENCING & COMMUNICATIONS CENTER

at

NEW JERSEY INSTITUTE OF TECHNOLOGY

PROGRAMMING LANGUAGE REQUIREMENTS
FOR
HUMAN COMMUNICATION STRUCTURES
OR
COMPUTER CONFERENCING

RESEARCH REPORT NUMBER 5
by
PETER GORDON ANDERSON

c/o Computer & Information Science Department
New Jersey Institute of Technology
323 High Street, Newark, N. J. 07102

PROGRAMMING LANGUAGE REQUIREMENTS FOR HUMAN COMMUNICATION STRUCTURES OR COMPUTER CONFERENCING

by

Peter Gordon Anderson

January, 1977

Department of Computer and Information Science

RESEARCH REPORT NUMBER FIVE
COMPUTERIZED CONFERENCING
AND COMMUNICATION CENTER

NEW JERSEY INSTITUTE OF TECHNOLOGY
323 High Street
Newark, New Jersey 07102

A detailed publication form. Copies may be obtained for $3.00 by writing the RESEARCH FOUNDATION at NJIT. (Checks payable to the Foundation at NJIT)

TABLE OF CONTENTS

I. INTRODUCTION

Our overall goal is to be able, simply and quickly, to construct computer conferencing systems for new requirements, applications, and even experimental ideas or fantasies. One way to view this requirement is as the creation of a highly parameterized conferencing system itself. Our view, however, is that of a programming language; i.e., an integrated notational system for the specification of communication structures and the associated actions or computation to be taken by the computer system hosting the structure.

A communication structure consists of a group of people (and storage devices) each endowed with some characteristics, and some means of person-to-person communication. We view the expression of such structure as a set of rules, R (a,b,c), that expresses the actions to be performed in case a participant of characteristics -a sends a communication of type -b to a participant (or set) of characteristics -c. These rules and characteristics may change over time -- a dynamic structure.

The new language to be developed must be able to express the formation of these rules (the details can be supported, of course, by existing coding systems). It must support the organization of such a scheme of dynamic rules.

This Report details the consideration (and examples) for such a language that we have unearthed in our studies.

II. GENERAL LANGUAGE REQUIREMENTS

We understand the main issues for the lower level language, i.e., the one for system programming. They are the features that systems programmers give up when they abandon assembly languages.

Data types include integers and bit and character strings, and record type and one-dimensional array data aggregates. Structured programming control sequencing and subroutines are needed.

There is, however, a large unknown component when we attempt to fit a language to the higher-level requirements, for here is the research border. Our answer to this issue, which will permit us to proceed rather than stagnate and "research" is to investigate and construct a highly flexible general system into which language features can be put (temporarily for evaluation as well as permanently for proved value).

A. THE IDEAL PROGRAMMING LANGUAGES

A language--natural or artificial--is a tool for human thinking and for communication with other beings. Some languages are more adequate to the task than others, richer, more expressive, more helpful, closer to the problem at hand, more natural, easier to manage, more readable, and more writable. But no language is perfect, and none is likely to be so. When a better language is made available, better people come forth with better ideas, and it's a sure bet that they better the language. No language inventor can foresee all the uses the language will be faced with, and eventually (more sooner than later) the users find themselves simulating the tools that they wish were built-in facilities. Thus, the FORTRAN user

clumsily works with character strings, push down stacks, linked lists, recursive processes, and laborious translates Dijkstra-dicta into conditional GOTO's.

What to do? Make the richest language imaginable at the time, and provide for an orderly growth process? That is known as the PL/1 answer. PL/1 is the answer to the FORTRAN programmer's prayer, or is it the gods driving mortals mad by granting their every wish? There is plenty of evidence for the latter assertion. PL/1 has such an embarrassment of riches that few users know them all, and this ignorance is paid for by a high coefficient of surprise. It's far from bliss for the user of "the engineering application subset" to find that 25+1/3 gives a fixed point overflow. (What's fixed point overflow, you ask? It's what you get from not using 25+1/03.) The goodies in this cornucopia are not pairwise independent; there is much tripping over each other to be avoided and feared.

There is another side. The true professional must be a master of his tools or have an expert on call. There are such people. But they present the most damning evidence against granting one's every wish--each of these experts has a personal wish list of 50 desirable features and fixes for PL/1 (no, these lists don't overlap much). (cf. The Magic Fish.)

What can we do to get the tools powerful enough for our tasks? We understand that coding in some programming language is no substitute for designing. Yet we must design (think, notate) in some language whose form is usually formed by our conception of the problem, its requirements, and

constraint. Designing proceeds by an iterative process of elaboration of the details of the problem solution until it's available for computer processing. This is the process known as "programming by successive refinement," or "top-down programming." A high-level programming language is a device to permit this iterative elaboration to terminate as early as possible.

An alternative to the everything-language, or PL/∞, is a means whereby a programmer can take a simple programming language and endow it with the features needed for the problem at hand. This is not as far-out as it may appear. At a primitive level, every programming language that allows programmer-defined functions and subroutines has such a feature--and that includes most programming languages. The most glaring exception is Assembly language, but even that is redeemed if it has a macro preprocessor. The first macro to invent is subroutine-call, and from then on we're competing on an equal basis.

B. PROPERTIES OF A V.H.O.L.

A Very High Order Language (V.H.O.L.) functions at the level of the user's problem area. A user can use his own notation and terminology. It imposes no requirements to simulate macro steps at the source level. For instance, a message may have a SENDER, a CONTENT, and a set of RECIPIENTS, all of which are invoked at this level. The user need never resort to an explicit handling of substructures or support mechanisms (e.g., pointers, indices, addresses, loop counters, etc.).

The system automatically fills in correct assumptions (BASIC,

FORTRAN, and PL/1 hint at this with their providing syntax for the source names of the various data types and contextual declarations for procedures and files). Even when assumptions can't be filled in automatically, the system must provide the user an out. It must avoid aborting the run, for instance by dynamically passing unknowns to the user for interpretation.

The principal requirement is for a flexibility of design, not necessarily a senscient computer. For instance, an unspecified function can be provided a "stub" by the system and a V.H.O.L. translator must allow a wide variation of expression. Names should be spellable in different ways; e.g., abbreviations. Calling sequences should be of many alternative styles, from assembly form to PL/1 form, to cookbook form ("add gravy made like the previous recipe, but omitting the salt"). All of the following should be legal and mean the same thing:

```
CALL INIT (SCORE)

INITIALIZE SCORE

NEXT, INITIALIZE THE SCORE
```

Similarly, when a procedure requires arguments of different data types, these should be expressable in arbitrary order, as:

```
GIVE PLAYER THE GOODBYE_SPEECH.

GIVE THE GOODBYE_SPEECH TO PLAYER.
```

If an argument can be determined by context, it shouldn't need to be given explicitly:

```
WHEN ANY PLAYER PASSES GO, PAY $200.
```

III. EXTENSIBLE LANGUAGES

In order to provide the highest level language necessary for our ends without having to make irrevocable decisions early in the project, we propose an extensible language facility.

It is crucial that flexibility be maintained and that progress be made. Hence, we must identify what decisions we can make or need to make at this time that will not be changed. These decisions are: we are to construct programming tools, or "languages," for people with a wide variety of skills who will construct, modify, and use computer conferencing systems.

These decisions then entail the construction of a systems programming high order language, the identification of conferencing primitives ("run time system"), and the development of a methodology for language construction.

This methodology constitutes the outstanding aspect of our study; namely, our language system must contain within itself the means for defining new constructs and redefining existing ones.

An Extensible Language

A classical approach to problems such as ours is to give up a programming language which serves as a problem-oriented-language for a highly restricted problem area. (In our case, we may argue, it's not so restricted. Our application area encompasses a multi-programming executive scheduler, user interfaces, data base management, statistical data collection and reduction, etc., etc.). What results is an ad hoc curiosity

of limited value outside of that individual research project and that particular team of dedicated researchers--dedicated to their research project and to the virtues of "their own" programming language.

So much effort goes down the drain, even if the higher-level-language constructed has obvious merit to users outside its own community. Results and capabilities that transcend a parochial application are needed for a higher-level-language to live. As is the case in so many other areas of software requirements, the needed attribute is flexibility.

Therefore, in this document we propose a family of computer-programming languages; we show some members of the family; and we describe how other members of the family arise. The family conceivably contains a member suitable for any given application area, but not necessarily a member suitable for every application area.

In our particular application area, there will be areas of programming which would best be handled with different programming languages. But our approach will be to identify members of the same larger family as the suitable tools for the different applications. (Examples of such disparate areas within our activity are: multi-user scheduling and text-editing.) This approach stands in sharp contrast with the PL/1 approach, in which every application area was to be serviced by a suitable subset of the whole language. The unfortunate result was a high surprise factor for the user who had not mastered the encompassing language and tried to concentrate only on his own subset. Furthermore, the compiler and run-time-system

are unwieldly large. Users pay for the parts of the language they don't employ. (There seems to be a trend towards improvement in this area. However, some investigators believe the cost of unused components is guaranteed to burden the user for a long time to come.)

Our research will be directed towards the discovery and creation of an extensible language. This is a system consisting of a loose language along with a mechanism for introducing new language constructs.

The base language will be an extension of Fortran, since we are currently working exclusively on a computer with an excellent "Fortran V" which permits escape to assembly language instructions mixed in with the Fortran source statements. Although this mechanism is an extremely tricky one to use reliably, it does give us a running start as with a poor man's extensible programming language. Their mechanism is provided by a classical cascade compiler: The Fortran translator produces assembly language input and simply copies over the non-Fortran code to be handled by the latter processor. This feature permits a language extension mechanism to be included in several ways, two of which are particularly easy in this context:

1. A pre-processor (cf., PL/1's compile-time facility) can be provided to translate new constructs into assembly language code at the Fortran source level.
2. Macros, as well as novel assembly language psuedo-operations, can be provided at the FORTRAN output level.

A. SOME EXISTING LANGUAGE EXTENSION MECHANISMS

If we look at this issue from the correct perspective, we can see language extension mechanisms in all existing programming languages. In fact, a High Order Language itself can be viewed as a mechanism for extending the functionality of the underlying hardware and software.

1. A fundamental concept of extension mechanism is that of the subroutine. This notation is the most important advancement ever put forward for "software engineering." We say that not only for its obvious advantages for work sharing, using canned programs, and core savings, but also for its introduction of the mental tool of abstraction and generalization into the software business. We are given the privilege to use abstraction the same way we could use physical objects.

To a large extent, subroutine invocations are used to implement language extensions that are not on their faces subroutine calls. We mentioned above the variations on invocation syntaxes; the possibilities are limitless here.

2 New operators form an obvious class of extensions. The profusion of built-in operators in APL suggest many useful forms to us (except that we prefer to introduce a means rather than pre-suppose everyone's desired end).

The Extended Algol translator supplied by Data General for their Nova and Eclipse mini computers allows the programmer to declare new operators, provide them with a priority, and give a subroutine to be invoked for evaluation.

The Algol approach is static. SNOBOL 4 allows the running program to modify the meaning of any operator, where an "operator" is understood to be one of a class of special symbols with system supplied priorities. Several of these operators (e.g., +,−, *, /) have an initial but changable meaning, and several others have no initial meaning (e.g., #, ¢).

3. Assembly languages are often provided with Macro expanders which allow new opcodes to be defined in terms of existing opcodes (including other macros). The general form of this feature is the definition of a parameterized procedure for generating code, which depends upon the form and attributes of the invoking parameters and the state of the macro expander system at (expansion) invocation time.

4. Parameterization refers to the facility for writing general programs in which constant configurations are replaced by symbolic names to be replaced during compilation with the desired values. This feature is a part of the proposed 1976 Fortran revision, where it takes the form

"PARAMETER X = 10, Y = 20"

and the parameters may be used, for instance, as

```
DIMENSION A(X,X), B(X,Y), C(Y,Y)
EQUIVALENCE (Q, C(X,X))
DO 1000 I = X,Y
```

and so on.

Parameters are given to assembly programmers by means of the EQU-statement. A less well understood parameterization is found in the

assembler (e.g., for IBM/360) DSECT facility where storage forms are defined but not allocated.

5. Libraries may also be considered extension tools. A library can be the home of the macro definition, the subroutines, the EQU-statements, and so on. Many language systems are equipped with an INCLUDE or COPY pseudo-op that is used to bring together portions of a program from outside libraries.

6. A special extension mechanism that needs to be mentioned, although many of its features are covered above from modified viewpoints is the PL/1 precompiler. This tool permits its user to describe, in a PL/1 type notation, the form of the PL/1 source program to be generated and compiled. It supports integer and character string function procedures and operations. In addition to explicit calls, it supports "active" variables which are evaluated into preprocessor source; that is, they are converted to a character string value and that character string is re-scanned, looking for more active variables to be replaced.

7. The computer science world abounds with other examples. Meta-assemblers are a fascinating example, but rather ill understood. Examples are "Fergason's Language," RCA's PLASM, and Xerox's Sigma series assemblers.

B. A MODEST APPROACH TO EXTENSIBILITY

As a modest initial step, we identify some of the particularly valuable extension mechanisms which can be gotten inexpensively.

1. INCLUDE source from a library (cf. "COPY" in COBOL and Assembly). All that would be needed for this feature is for the compiler or a pre-processor to open a file in a library and merge source lines with the user supplied source. There should be provision made for the INCLUDED information to have in it INCLUDEs of other files This makes the effort only slightly non-trivial; a few files may have to be open simultaneously, and the merging mechanism would have to incorporate a push down stack or be encoded recursively, But this entails little sophistication.

The utility of such a facility is not evident for small ("toy") projects, but for ones of the size we are planning it would allow a flexibility and management control otherwise almost unobtainable through FORTRAN. In particular, the major data structures embodied in COMMON, DIMENSION, EQUIVALENCE, and type statements need only be maintained by project leaders ("Chief Programmers") and not accessible to unauthorized coders. If decisions are reversed, the INCLUDE files have the only source code that needs to be changed. All that remains is a universal re-compilation without human intervention).

2. PARAMETER statements, which allow symbolic use of compile time constants (such as those requisite constants in DIMENSION, COMMON, and EQUIVALENCE statements) is an easy-to-implement facility for a compiler or a pre-processor.

3. An enriched facility, taking up where PARAMETER left off, is the

SUBSTITUTE identifies = string, . . .

which means that everywhere the "identifier" is detected in the source text, it is to be replaced by the "string" it is associated with, and that string is to be re scanned for substitutable identifiers.

Examples of the use of SUBSTITUTE:

a. SUBSTITUTE IJK = '(I, J, K)'

A IJK = B IJK + C IJK

b. SUBSTITUTE DEGREES = '*PI/180.'

Z = COS (THETA DEGREES)

c SUBSTITUTE DIM = 'DIMENSION', EQ = 'EQUIVALENCE'

DIM X(10, 20), Y(20)

EQ (Y(1), X(5, 1))

C. A LESS MODEST APPROACH TO EXTENSIBILITY

The SUBSTITUTE facility given above is but a hint at a more general text replacement and generation tool along the lines of Macro assemblers and PL/1's compile time feature. Specifically, the SUBSTITUTED string needs to depend upon several things:

- user-supplied parameter forms
- attributes and values associated with parameters
- states of other relevant variables in the system

The dependence is not to be restricted to simple formulas, but will be based upon algorithmic computations performed during substitution.

Macro Processing

This activity is one of a pre-processor nature, and, although it pre-

pares input to a language processor, it does not deeply modify the language (specifically its semantics). The requirements are that a macro pre-processor have:

1. Parameter Processing
2. Memory (Global variables)
3. Computational ability (arithmetic and testing)
4. Symbolic typing
5. Storing processing

With these capabilities in a macro processor, much of the facility of a full-blown extensible language will be available. Direct control of machine-level object code and optimizations will be unaccessible, but the source level facilities will permit the users to experiment and determine the usefulness of their ideas before a large implementation investment is made. (The macro processor could, however, output CAL assembly code, to bypass FORTRAN completely.)

An interesting application is the NEW and OLD value generation idea. Everytime NEW is referenced in the source, it stands for a different (unique) value; and everytime OLD is referenced it refers to the most recent reference to NEW. Unique values for "nominal values" can be generated by:

PARAMETER JOE = NEW, MOE = NEW, BOBO = NEW

and locations in a list can be easily given by

EQUIVALENCE (X, A(NEW)), (Y, A(NEW)), (Z, A(NEW)).

Thus the system can make decisions for the programmer:

READ A(NEW), B(OLD), A(NEW), B(OLD), A(NEW), B(OLD)

D. AN AMBITIOUS APPROACH TO EXTENSIBLE LANGUAGE

A programming language is known by its data types (and accompanying operation), its control mechanisms, and its surface form (syntax). The control and form are (largely) determined once we have chosen a base language. (Pre-processors can change the looks of something, but they provide a limited amount of leverage.)

To introduce new data types, consider the simplest example: that of nominal data. The terminology comes from statistics where it denotes a finite (small) classification scheme that cannot be treated as numerical (except via coding, like identification numbers). Examples include: day of the week, marital status, profession, religion, city, state, blood type, etc.

To provide nominal types to a language, one needs simply to list all its values; e.g.,

```
NOMINAL TYPE COLOR = (RED, BLUE, GREEN, YELLOW)
TYPE COLOR X, Y, Z
```

The operations are only those of assignment and equality comparisons:

```
IF (X.EQ.RED) THEN
    Y = GREEN
ELSE
    Y = Z
    Z = YELLOW
ENDIF
```

A more interesting data type invention must involve directions (algorithms) for specifying the storage structures to be built and allocated for data of that type. This could entail some very large (core consuming) structures, especially for our application involving networks with attributed modes and arcs as data structures.

The operations can be defined in terms of several of their defining qualities:

1. The character string defining the operator
2. The syntax of the operator
    a. prefix, postfix, or infix
    b. priority
3. The data type of its operand(s)
4. The data type it returns
5. The rules for its evaluation

In the current document, we shall not presuppose mechanical linguistic form for such specification, but rather proceed with a list of examples to show the utility of such a method and, hopefully, suggest some forms for its realization.

IV. Examples of the Utility of an Extension Mechanism

In this paragraph, we will set forth a collection of examples of desirable features for a high order language that we would like to achieve through our extensibility mechanisms. These features are not what belongs in any HOL, but what might be needed at some time, possibly in the forms shown We stress that this is a goals list, and that we are not proposing at this point any specific mechanisms to support them.

a. Store within an expression

```
X = A(I := I + 1)
```

b. Matrix multiply

```
C = A *** B
```

c. I/O

```
X := Unit_5          (read)
```

d. Stacks

```
STACK := NEW_TOP     (Push)
STACK =: OLD_TOP     (Pop)
```

e. Cross Sections of arrays

```
A *-* I
```

f. Equality considered as a numeric

```
I := X == Y
```

g. Swap

```
A(I) :=: A(I + 1)
```

h. Virtual variables. These variables have no address permanently associated

with them. By overlaying a virtual variable somewhere in memory we achieve a template effect. This is similar to FORTRAN's EQUIVALENCE, but it is dynamic.

i. Ragged arrays. Usually arrays have the same number of elements in each of their rows; if not, they are called ragged. These are useful for small data bases.

j. Nominal variables. These are variables that take on a small, finite set of values. They can be compared for equality or assigned values. Examples are sex, religion, region, city, breed, marital status, color, etc.

k. Queues.

l. Lists.

m. Trees.

n. Hash Tables.

o. Sets.

p. I/O buffers. Data can be added to an I/O buffer; when it's full, it will flush automatically. Input buffers work the same way, in reverse.

q. Graphs.

r. Diagrams. Extensible languages may be the proper setting for computer graphics.

s. Formulae. For symbolic calculations.

t. Records. Data aggregates with (possibly) non-homogeneous elements.

V. An Implementation Technique

The burden of this tool could be taken over by a pre-processor for a cascade compiler (which we have) with an assembly escape clause (which we have) with a powerful macro expansion facility (which we plan to implement). There are shortcomings here of an efficiency nature, but the semantics are available for the initial research. Thus, new (defined) constructs can be turned over to macro calls by a pre-processor without having to modify an ill structured compiler.

A. CHOICE OF A BASE LANGUAGE (EXTENDED FORTRAN)

The base language is the tool to be used for systems programming. It must be powerful enough to write compilers and operating systems. It must also be amenable to the attachment of self-extension mechanisms.

We are given, on our computer, a FORTRAN system which is implemented by a cascade compiler (i.e., a translator that produces assembly language source) and consequently easily allows assembly code to be mixed with FORTRAN source. While this setup is not necessarily ideal (the defects of FORTRAN are well known), this does provide enormous flexibility of linguistic structure and content. Processors can operate both before (pre-processors) and after (macro systems) the FORTRAN system itself. So the FORTRAN compiler need never be touched.

There are some minor, but very important modifications that need to be made to the FORTRAN language of control figures and new data types.

Required Control Structures

FORTRAN and its dialests force programmers to code up the most dreadful object code to simulate the control structures we require to do sensible programming. Largely we intend to obviate statement labels except for Formats. Specifically we require:

1. The WHILE loop.
2. The conditional group ("IF").
3. Generalized conditionals ("CASE").
4. Generalized loops.

Denote by C-exp a Fortran IV conditional expression, e.g., the inside of a logical IF or the right hand side of a logical assignment statement, such as X(I) .LT. X(J) and A .NE. BLANK .AND. A .NE. COLON.

Next, denote by s-list a sequence or list of Fortran statements or complete blocks. We expect--except in extreme circumstances--that the flow chart of this sequence of statements will be a single entry, single exit process box which has no jumps out or in except by straight flow (maybe lots of internal jumps).

A WHILE loop is of the form

```
WHILE (C-exp) LOOP
    s-list
ENDLOOP
```

This is semantically equivalent to

```
1001 IF (.NOT (C-exp)) GOTO 1000
          s-list
     GOTO 1001
1000 CONTINUE
```

A conditional group is of the form

```
     IF (C-exp) THEN
          s-list
     ENDIF
```

The conditional group is semantically equivalent to the Fortran IV

```
     IF (.NOT. (C-exp)) GOTO 1000
          s-list
1000 CONTINUE
```

Style Notes

The list of statements in the s-list above can include complete WHILE loops and conditional groups. Nesting to any level is OK.

If the current tab stop is to be at card character position N (N = 7 at the start of a program), then the words WHILE and ENDLOOP are coded starting in position N, but the intervening s-list has N+3 (or N+4) as its current tab stop. Conditional groups have the same indentation convention.

This makes the nesting of the flow structure evident to the reader. An automatic formatting routine will be supplied to take care of this indenting

for programmers.

The conditional group is distinguished from the Fortran IV logical IF by the word THEN.

Generalized conditionals: alternatives (IF-THEN-ELSE)

This takes the form

```
IF (C-exp) THEN
     s-list-1
ELSE
     s-list-2
ENDIF
```

The If-THEN-ELSE construction is semantically equivalent to the following sequence of FORTRAN code.

```
     IF (.NOT. (C-exp)) GOTO 1000
          s-list-1
     GOTO 2000
1000 CONTINUE
          s-list-2
2000 CONTINUE
```

The ELSEIF construction is added as a convenience to preclude having to nest several levels of IF statements and thereby confuse the structure.

```
IF (C-exp-1) THEN
     s-list-1
ELSEIF (C-exp-2) THEN
```

```
        s-list-2
ELSEIF (C-exp-3) THEN
        s-list-3
            -
            -
            -
ELSEIF (C-exp-n) THEN
        s-list-n
ELSE
        s-list-n+1
ENDIF
```

(The ELSE-clause is optional and zero or more ELSEIF-clauses are permitted.) The semantics are that the first true C-exp-k is found and the corresponding s-list-k is executed. Only one s-list is executed.

The CASE construction is similar to the IF-ELSEIF condition in that one out of several s-lists is chosen to be executed. In this case, the selection is on the basis of an integer valued expression.

```
SELECT integer-expression OF
CASE (i_{1,1}, i_{1,2}, ... i_{1,j_1})
        s-list-1
CASE (i_{2,1}, i_{2,2}, ... i_{2,j_2})
        s-list-2
            -
            -
```

ELSE

s-list-n+1

ENDSELECT

Note the symmetry (one arrow in and one arrow out) which is missing from Fortran IV's computed GOTO.

Generalized Loop Constructions

The UNTIL Construct

This is a convenience feature which permits one to postpone condition checking until after the body of a loop has been executed once.

Its form is:

LOOP

s-list

UNTIL (C-exp)

It is semantically equal to the Fortran

1000 CONTINUE

s-list

IF (.NOT. (C-exp)) GOTO 1000

The EXITIF Construction

Often we want to exit from a loop neither at the start nor at the end (cf. RETURN from other than the end of a SUBROUTINE).

This can be accomplished via:

LOOP

s-list-1

```
      EXITIF (C-exp)
            s-list-2
      ENDLOOP
```

For example:

```
      LOOP
            fetch input
      EXIT (end of file)
            process input
      ENDOFLOOP
```

This is semantically equivalent to the following

```
1000  CONTINUE
            s-list-1
      IF (C-exp) GOTO 2000
            s-list-2
      GOTO 1000
2000 CONTINUE
```

There has been a great deal of effort in the past few years towards making FORTRAN into a language suitable for expressing "structured programming" constructs. Such work has been done on the FORTRAN (ANSI) standard committee as well as unilaterally by compiler writers and users. If any definitive standards emerge (or appear to) we will follow them immediately. We will also track these studies and adopt the good ideas that show up.

B. New Data Types

We need to allow our systems programmers to get near the host machine without resorting to assembly code escape. Programming efficiency requires a binary or bit string data type and our eventual goal of conferencing as well as compiler development requires character strings.

a. Character strings are part of the 1976 Fortran revised standard and are closely modelled after WATFIV. They are declared as:

CHARACTER * n identifier,...

where n is the length of the string. The string "identifier" may be dimensional or provided with another !*m" following the name to override the given length.

Character string literals (constants) are delimited by apostrophes or question marks. The left and right delimiters of a literal string must be the same. This permits literals like:

X = 'HE SAID, "HELLO!"'

Y = "DON'T SAY AIN'T!"

Character string variables can be used in assignment statements, comparisons (with .EQ. and .NE. obviously, and with .LT., etc., using the machine's collating sequence), subroutine arguments, and in I/O lists with A-Formats. A character string may also be named in place of a Format line number in a READ or WRITE statement or in place of a data set reference number in a READ or WRITE statement (this gives the ENCODE/DECODE facility).

Special operators to be used for character strings are concatenation which is denoted by //; substring denotation which is given by X(I:J) where I and J are integer-valued expressions, I giving the initial character position of the substring of X and J the length; and the INDEX function, where

INDEX (X, Y)

is the smallest value, I, such that X(I:LENGTH(Y)) = Y. If no such I exists, then INDEX (X, Y) is zero.

The binary data type is the other new necessary addition. These may be used as non negative n-bit numbers as well as n component binary vectors. The declaration form is

BINARY * n identifier, . . .

as was used for CHARACTER. Binary data may be manipulated the same way character strings are, with concatenation, substring, and index function specified the same way. In addition, there are bit pattern functions for AND, OR, EXCLUSIVE-OR, NEGATE, ROTATE, and SHIFT, to allow functions that are normally performed at the assembly language level to be expressed in a H.O.L.

Binary literals are sequences of 0's and 1's, enclosed in apostrophes or quotation marks, and followed by the letter B: '010110111'B. For abbreviation, one may use hexadecimal notation with a sequence of hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) in apostrophes or quotation marks, followed by the letter X or H.

C. Syntactic Issues ("Sugar")

Comments take a modified PL/1 form: information on one card between /* and */ are syntactically equivalent to a single blank. If the */ is missing from the card, then the remainder of the card following /* is a comment, and the next card is not part of that comment.

Listing control will be by way of special pseudo cards detected by the presence of a minus sign in column 1. Options are

EJECT

SINGLE SPACE

DOUBLE SPACE

OFF PRINT

ON PRINT

TITLE

Identifiers can be long names with the embedded break characted. E.g., THIS_IS_A_LONG_NAME.

Symbolic operators: since the characters > , < , &, etcetera are available on our equipment (key punches and terminals) we will utilize them instead of the less convenient, less mnemonic, .GT., .LT., .AND., etcetera.

Superficial Syntax (lexical level syntax)

Programming language compilation resembles human information processing in many aspects. At the most primitive levels of input, humans

receive sensations and computers receive individual characters; at a higher level, where things can be thought about, humans receive percepts (sensations integrated into connected, namable clumps) and computers receive input data tokens (connected meaningful strings of input characters). The superficial syntax of a programming language consists of the definition of what constitutes the tokens of the language. This is in contrast with the usual notion of syntax given in, say, Backus-Naur-Form formation, which defines the higher or conceptual levels of linguistic entities or phrases, such as expressions, statements, blocks and programs. Our goal is to produce an extensible language such that the user can specify the higher syntax using the token rules according to superficial syntax, along with some very simple guidelines.

Our superficial syntax is similar to that of PL/1. We approach this by specifying several types of tokens and showing how they are detected. Words are strings consisting of letters, digits, and break (_) characters. Operators are strings consisting of the characters in the following string +-*/$¢#?@/= ❮❯!&%. Separators are the single characters , . ; : ' " ( ). Blanks are needed to separate a pair of adjacent words or a pair of adjacent operators. Between other pairs of adjacent tokens, blanks are optional and serve no purpose other than human readability; a string of two or more blanks is (almost) equivalent to a single blank.

There are two exceptions to the above rules: literals and comments. Inside either of these constructs, all bets are off. A comment string is

syntactically identical to a single blank; it begins with the operator /* and ends with the operator */. A literal is a self-defining value which is replaced by the token detector by an internal name and is moved as a constant into the object program unchanged. Literals begin with the operator ___ and end with the operator ___. The character strings inside comments and literals are not processed by the token detector.

Definitions

One of the reasons for the above superficial syntax as opposed to that of, say, FORTRAN, in which blanks are optional and permitted (almost) everywhere, involves an extremely simple but most valuable language extension facility: that of replacable words.

A word (v. s.) can be DEFINED to be the same as a character string, and the presence of such a word in the source is equivalent to the presence of the string the word has been defined to stand for. This facility, by itself, allows shorthand and parameterization. It is also an approach to security, whereby programs can be written, symbolically referencing critical numbers which can be guarded by themselves.

Such replacement can be done directly by the computer's token-detector or by a second layer inserted above the token-detector and below the parser.

A further improvement, and a requirement for a useful implementation language, which fits into this context, is macro processing.

E. THE PROBLEM OF OPTIMIZING EXTENDED CONSTRUCTS

Suppose that we define the "whole array" sum and product by ++, ***. We must avoid generating a temporary then a move for

A = A++B

One solution (cf. Cobol's "ADD B TO A") might look like this.

A := ++B

String concatenation poses the same problem. If S is a varying string, what code comes from

S = S//T ?

Or if S, T, and U are fixed-length strings disjoint from eachother, what about

S = T//U ?

This is the optimizing problem for high level languages.

In order to avoid these extraneous expensive temporary states of computation there are two approaches we might take. One is described above where operations are written as though the machine being programmed were a 2-address machine (with operations X = f(X, Y)). The other approach requires a top down code generator scheme, where we have a parse tree given root-first rather than leaves-first. I. e., this

Assignment

Target A

Expression ++

Left array A

Right array B

rather than

Left array A

Right array B

Operate ++

Target array A

Assignment

The root-first approach allows one to establish the context of an operation before the operation is performed (generated). In this way, more of the special cases will come to light and good, efficient code can be generated. This still puts a burden on the definer who wants a specific extension. However, the existence of a worked out method for efficiency will permit possible effective use.

Notice also that the top-down parsing will allow us the power to define an AND operator that permits evaluation only up to as far as necessary, as in

P(1) AND P(2) AND P(3) AND P(4)

when the P's represent long calculations, and

IF ((A .NE. O) AND (X/A .NE. B))···

where the falsity of the first condition implies the impossibility of evaluating the second.

VI. The Communication Network

A. The conferencing system works by sending and receiving messages between the computer and individual members in a pseudo-simultaneous mode. Any communication network that is desired with the members as nodes is achieved by computer-simulating a messenger. If any non trivial structure is required--and there almost always will be--the messenger needs to be supplied with a network specification: who can do what, when, and to whom.

Formally, a communication network specification is a rule which associates an action (or sequence of actions, i.e., subroutines) with a given pattern of:

1. transaction originator or sender,
2. type of transaction requested,
3. requested recipient of the transaction,
4. state of the system.

Although this may sound like a request to be able to process every imaginable crazy fluidic networking rule, we have in mind only a small astronomical number of crazy fluidic network rules.

A Principal Simplification. Suppose that there are a small number on conference members; a small set of transaction types, and a state independent decision rule for legality and interpretation. In this case, the network specification can be stored in a three-dimensional array of action names (or identification numbers). The code

ACTION = NETSPEC (SENDER, RECEIVER, TRANSTYPE)

is all that is needed.

For larger member sets, we can use, say, a privilege level or a subclass name instead of the fully identifying SENDER or RECEIVER code. There are many possible variations on this theme. The various transaction types, or categories, can have their own network matrices whose coordinates are aspects of member types as above, but different aspects for different transactions or types of transactions.

If the dependence on system state is so interpretable, it can be implemented into a matrix-driven network specification by allowing the matrix itself to vary as a function of system state change (e.g., when time is almost up, all the members may do is vote on the main issues and send private messages.)

In other cases, a simple rule can be put forth to determine the response to a request, examples of this are the following:

1. In the ring of five subjects, a member may send a message to another member if the two members are in the class of experimental subjects and if their identification numbers differed by 1 or 4. Notice that this can be extended to larger sets of experimental subjects and larger neighborhood sizes without a quadratic increase in core requirement expected with a stored matrix.
2 Certain transaction--e.g., private messages--might be legal if and only if the two parties belong to the same sub class. In this case, the matrix would be constant and needn't be stored; closely

related are transactions whose legality depends only upon an order relation between some attributes of the parties; e.g., one may be able to send a non-refusable message to any member of lower privileged class. This can be summarized by appeal to the following FORTRAN examples.

Alternative I.

Generate a legality network matrix as follows:

```
      LOGICAL NETWORK (5,5)
      DO 1000 I=1,5
          DO 2000 J=1,5
              NETWORK (I,J) = FALSE
2000      CONTINUE
1000      CONTINUE
      DO 3000 C=1,5
              K=I+1
              IF (K .GT. 5) K=K-5
              NETWORK (I,K) = TRUE
3000      CONTINUE
```

Then the legality of a transaction from A to B can be established by

```
      LEGAL = NETWORK (A,B)
```

Alternative II.

A matrix need not be generated in case of a simple rule, as follows:

```
      DIFF = ABS (A,B)
      LEGAL = (DIFF.EQ.1).OR.(DIFF.EQ.4)
```

B. Conference Participants as a Data Type

The conferencing mechanism involves handling communication between the computer storage and any one of a large set of members of a particular conference. Any subset of the member set may be on-line to the computer at any given time, and the management of this collection is the principal business of the conferencing system.

Conference members originate transactions and the system, simulating a conference member, may also originate a transaction. Only certain transactions are legal, and the legality as well as the interpretation for handling them depends upon the type of the transactions, the attributes of the individuals, and the state of the system. Note that all of these aspects are dynamic and central to our problem area. Consequently, they are all appropriate candidates for data types in a language intended for the description and support of conferencing systems.

Software development for conferencing support will be intimately involved with the member data, especially at the conceptual high levels (at the lower levels of detail elaboration meaning, are obscured and the differences between applications is not evident).

Member data types contain a host of information (or allow one to retrieve and use a host of information) about a member, such as:

The activity level,

The protocol involved in the current communication,

The privileged class.

This outline in in no way intended to prejudice the implementation strategy

or forms of coding representation of this data. In fact, it is essential that such details be left unspecified and hidden or transparent to implementors, for reasons of flexibility. Methods of access for retrieval and modification of the data will be provided in terms of higher-level or functional constructs. Furthermore, this data type will allow access to all information about a member. So an extreme, but workable implementation would be simply that of a key to the file of member data. More likely some compromise will be made allowing much information to be maintained in core.

C. Textual Data

One of the most obvious needs of conferencing systems is the ability to handle character string information in rather sophistocated ways.

There will be many different uses in which character strings will appear, and many different forms they may take. The obvious ones are those of variable length (cf. PL/1), or of distributed contents (where substrings of a string are found in different sections of storage), or of virtual existence (overlayed substrings of other strings). Here we propose a novel string type that should appeal both to a systems programmer and a computer conference spec writer.

Define a "Hyper-String" to include the notion of ordinary pedestrian character strings that are manipulated as though they had no meaning; but also allow another "font" to be mixed in with the usual font. Characters in the new font need to be processed when their host character string appears in certain contexts (especially as the object of our output verb).

Denote by ((the change to, and by)) the change from the "need-

to-be processed" font (assume double parens are never used in ordinary text). We could get the following, whose meaning is crystal clear

```
MSG-TO-X:
        HELLO THERE, ((NAME_OF (X) )): I HOPE YOU ARE
        HAVING A HAPPY ((DAY_OF_WEEK)).
        I HAVE ((SIZE_OF_QUEUE (X)) THINGS TO TELL YOU.
END-MSG.
```

VII. BRIDGE

Another example of the use of our envisioned V.H.O.L. is provided here. This example is not shown coded out in full detail; only enough code is given to illustrate the case with which such a specification can be made. We estimate that using such a V.H.O.L. (as contrasted with assembly code or a language like BASIC) reduces the time to create a bridge-monitoring session from man-months to man-days.

The bridge monitoring system services four people at their separate interactive terminals, allowing them to play bridge. (Imagine people who for some reason love bridge but can only get to a computer terminal; they could play the game at their leisure by simply calling up a central bridge playing service. This could cover handicapped people or busy executives with 20 minutes to spare between meetings.)

A not uncommon college hazing practice has four upperclassmen each sitting alone in their dormitory rooms while a poor freshman runs between them dealing cards, taking their bids, gathering the tricks as played, and keeping score. This computerized system simulates the hazee.

```
PLAY_A_RUBBER:
        INITIALIZE_SCORE;
        CHOOSE_DEALER;
        UNTIL NS = 2 OR EW = 2: PLAY_A_HAND.
END OF PLAY_A_RUBBER.
```

```
PLAY_A_HAND:
     SHUFFLE;
     DEAL;
     BID;
     UNLESS NO_BID, PLAY;
     UPDATE SCORE;
     ROTATE DEAL;
END OF PLAY_A_HAND.

DEAL:
     GIVE N & SCOREBOARD_N'S_HAND     CARDS(1..13), SORTED
     GIVE E & SCOREBOARD_E'S_HAND     CARDS(14..26),SORTED
     GIVE S & SCOREBOARD_S'S_HAND     CARDS(27. 39),SORTED
     GIVE W & SCOREBOARD_W'S_HAND     CARDS(40..52),SORTED
END OF DEAL.

SHUFFLE:
     RANDOMIZE CARDS(1..52)
END OF SHUFFLE.

BID:
     WHEN ANY PLAYER ASKS "REVIEW", GIVE SCOREBOARD BIDS;
     BIDDER IS DEALER;
     LAST BID IS NOTHING;
     PASS COUNT IS ZERO;
```

```
    REPEAT UNTIL PASS COUNT IS 3
            ASK BIDDER FOR NEWBID UNTIL VALID
                (NEWBID, OLDBID)
            GIVE ALL PLAYERS: "((BIDDER)) BID ((NEWBID))"
            ADD NEWBID TO SCOREBOARD_BIDS
            IF NEWBID IS "PASS" ADD 1 TO PASS COUNT
            IF NEWBID NOT "PASS" LASTBID IS NEWBID
            BIDDER IS PLAYER AFTER BIDDER
    GIVE ALL PLAYERS: "THE CONTRACT IS ((LASTBID))."
    DETERMINE WINNER; LEADER IS PLAYER AFTER WINNER;
       DUMMY IS PLAYER AFTER LEADER;
    GIVE ALL PLAYERS: "THE HAND WILL BE PLAYED BY
       ((WINNER))."
END OF BID.

PLAY: NEW AND WHEN ANY PLAYER SAYS "BOARD",
    GIVE ALL PLAYERS "DUMMY"S HAND IS: ((SCOREBOARD
       DUMMY'S HAND))",
    DO 13 TIMES:
            A(1) IS LEADER; A(I+1) IS PLAYER AFTER A(I),
               I = 1..3;
            ELICIT I-TH CARD FROM I = 1..4
            COMPUTE TRICK WINNER (NEW LEADER)
END OF PLAY.
```

SCOREBOARD DATA STRUCTURE:

CARD(1..52)

ORDERED VALUES:

C-2, C-9, C-10, C-J, C-Q, C-K, C-A

D-2, D-9, D-10, D-J, D-Q, D-K, D-A

H-2, H-9, H-10, H-J, H-Q, H-K, H-A

S-2, S-9, S-10, S-J, S-Q, S-K, S-A

NUMERICAL VALUES:

WE(ABOVE, BELOW, GAMES, TRICKS)

THEY (ABOVE, BELOW, GAMES, TRICKS)

BIDSEQUENCE:

CONTRACT:

SUIT, NUMBER, WINNER, DUMMY

HANDS: N(1..13), E(1..13), S(1..13), W(1..13).

END OF SCOREBOARD.

VIII EXAMPLES OF COMMUNICATION STRUCTURES AND CONCLUSION

To keep in mind the goals of our project--that of being able easily to describe the construction of human communication structures--we herein provide a catalog (rather unstructured list) of possible such structures. Of course, our paradigm is the existing conferencing systems (e.g., EMISARI, EIE, bell Canada, etc.), but these need to be generalized in the same way that any scientific computer program may be a paradigm for the generalized FORTRAN or Algol implementation structure.

A. Communication Structures:

1. games, like: Bridge, Backgammon, Scrabble, Monopoly, Battleship
2. Robert's-rules-of-order meetings (one conferee is a parliamentarian)
3. courts of law, with lawyers, judge, jury, witnesses, etc.
4. multi-lingual conferences (some conferees are translators)
5. magazine editorial staff
6. college faculty committees
7. delphi studies
8. panel of expert consultants and their customers
9. auctions
10. stamp-collectors' club trading session
11. team report writing
12. psychological counselling
13. tutoring
14. students' study session

15. writing, testing, and refining CAI systems
16. market research survey taking
17. TV script-writing
18. labor negotiations
19. modeling and simulation
20. routing slips with predetermined order (or partial ordering) and cancellation privilege
21. realtor's network ("MLS")
22. problem-solving network
23. research teams (including directors, technicians, clericals)
24. document de-classification procedure (cf. Privacy Act of 1974)
25. collective intelligence structures
26. brainstorming
27. mail
28. demanding questions
29. office business

B. SUMMARY

Our goal includes gaining the ability to construct such example systems with the same effort and time delay that it currently takes an engineer to construct a 100 - to - 1000 - line FORTRAN programmer. That is, the design and skeleton is together in a week; in one or two man-months, the system is in working order, able to produce usable results.

IX. REFERENCES

Anderson, P.G. A Structured Approach to Computerized Conferencing Proc. 1975 Symp. on Computer Networks: Trends and Applications. pp 61-68

Anderson, P.G. and John Ryon III "A language for describing human communication Structures", Proc. 1976 ICCC. pp 222-279

Bales, R.F. and Strodtbeck, F.L. Phases in Group Problem Solving, J. Abnorm. & Soc. Psych. 46:485-495

Bales, R.F. A Set of Categories for the Analysis of Small Group Interaction. Am. Soc. Rev., 15:257-263

Bales, R.F. 1950b Interaction Analysis: A Method for the Study of Small Groups. Reading, Mass.: Addison-Wesley, 1950.

Bavelas, A. 1950 Communication Patterns in Task-oriented Groups. J. Accoustical Soc. Am. 22:725-730.

Carter, George "Planning Computer-based Conferencing Systems", University of Illinois.

Chapanis, Alphonse "Interactive Human Communication"

Day, L. Computer Conferencing: An Overview, in Computer Communication: Views from ICCC '74

Engelbart, D.C. "Augmenting Human Intellect: Experiments, Concepts, and Possibilities", Stanford Research Institute, March 1975.

Hall, T.W. Implementation of an Interactive Conference System, Proceedings of the 1971 Spring Joint Computer Conference.

Hiltz, Roxanne "The potential social impact of some near future developments in computer conferencing", World Futures Society Panel on "Electronic Links for Invisible Colleges", June 4, 1975.

Hiltz, Roxanne "Communication and group decision-making", NJIT-Sept. 1975

Hough, R.W. "Teleconferencing System: A State of the Art survey and preliminary analysis", Stanford Research Institute, May, 1976

Johansen, Robert Pitfalls in the Social evaluation of teleconferencing media, Second Annual Int'l Communications Conference

Language and Systems Development, Inc. "XBASIC for Univac 1100 series computer", 1971

Leavitt, H.J. 1951 — Some Effects of Certain Communication Patterns On Group Performance. J. Abnorm. & Soc. Psych. 46:38-50

Martin, Shirley M., et al — "Practical experience in computer based message systems", U.S.Army Material Development and Readiness Control, Alexandria, Va.

Nilles, Jack J. — "Telecommuting--an alternative to urban transportation congestion", IEEE Trans on System Man and Cybernetics, Feb. 1966

Panko, Raymond R. — "The outlook for computer message services: a preliminary assessment", STanford Research Institute, March 1976

Scher, J.M. — "The Constrained Computerized Conference--A Methodological Tool for the Implementation of Simulation Games", Proc ICCC, Aug 1976, pp230-235.

Shaw, J.C. — "JOSS: experience with an experimental computing service for users at remote typewritten consoles"

Shaw, J.C. 1954 — Some Effects of Unequal Distribution of Information upon Group Performance in Various Communication Nets.

Shaw, J.C. 1964 — "Communication Networks: in Berkowitz Leonard, (eds.) Advances in Experimental Social Psychology. Vol 1 NY Academic Press, pp111-147.

Shore, J.W. — "C.M.I. Conferencing Phase 2: An Instruction Manual", Bell Northern Research, April 1974, TR3C30-1-74.

Stanford Research Institute — "NLS-8 Glossary", Augmentation Research Center July 1975.

Knowledge Workshop Development, Jan 1976

"NLS-8 Command Summary" May 1975

Turoff, Murray — "The cost and revenue of computerized conferencing", Proc ICCC, Aug 1976 pp214-221

Computerized Conferencing and Real Time Delphis. Second Int'l Conference on Computers and Communications, Aug 1974.

"Party-Line" and discussion, Computerized Conference Systems. Int'l Conference on Computer Communications, IEEE, Oct 1972

"Computerized Conferencing and the Homebound Handicapped", NJIT May 1976.