A LANGUAGE FOR THE IMPLEMENTION OF ONLINE INFORMATION SYSTEMS

by

Howard John Ferch

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

Department of Computer Science

The University of Manitoba

Winnipeg, Manitoba

May, 1973

ABSTRACT

Information systems for processing stored data have performed data manipulation in either of two manners: using a set of subroutine calls or extensions to current host languages such as COBOL or PL/1, or with new self-contained special purpose languages. Systems of both of these types share many common features.

This thesis proposes an implementation language for use with online interactive information systems. It is designed to provide facilities such as list and character manipulation for the implementation of a system, and facilities needed by applications in the information system such as the ability to manipulate complex data structures.

This language contains several important features including a simple and concise generalized syntax, dynamic data declarations, and extended data structure handling capabilities. It is an extensible language in that a user

may define his own data types and operators.

It is intended that this language be used in a manner analogous to the use of the APL system for mathematical applications. The information system would be written as a set of procedures loaded from an external library into a workspace. Applications on the data would be handled with new procedures added by the applications programmer and interfaced directly with the information system (a host language facility); or using a self-contained language implemented with a set of procedures loaded with the information system.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 2

CHAPTER 2 DATA ITEMS 5

CHAPTER 3 OPERATORS 13

CHAPTER 4 PROGRAM STRUCTURE 20

CHAPTER 5 AGGREGATES 30

CHAPTER 6 PROCEDURES AND ARGUMENTS 38

CHAPTER 7 DECLARATIONS AND OPERATOR SELECTION . . 49

CHAPTER 8 CONCLUSIONS 73

REFERENCES 77

BIBLIOGRAPHY 79

CHAPTER 1 INTRODUCTION

With the advances being made in hardware and software technology in modern computer systems, teleprocessing is becoming more widely used. This growth is especially evident in large-scale information systems (1) with more and more emphasis on on-line information entry and retrieval using terminals remote from the main computer installation.

Since they are mostly written in assembly language, such systems require a large expenditure in terms of man-hours and machine time to design and implement. However, a careful study (1) of these systems reveals that there are many common elements. These include the support of similar data structures and operations, similar host languages (1) and similar terminal operation.

This thesis proposes a special high-level language designed to be the base language in the implementation of information systems. It is also designed to supply the host language needs for writing application programs for

the information system in a manner similar to the APL system (2) used for mathematical systems. That is, the information system is written as libraries of procedures written by the systems programmers and debugged directly on a terminal. To use the system, the user causes the loading of the specific part of the information system that is relevant to his needs and communicates with these procedures to process his requests. New procedures may be added to the basic information system directly on the terminal to support particular applications if a host language facility is needed.

The proposed language is designed to be implemented in an interpretive manner, as is done in the APL system. Thus statements in the language are not translated into internal machine language but are interpreted every time they are to be executed. This allows for quick debugging and correction of statements since an entire procedure does not require translation into machine language every time a statement is modified. This also allows for procedures whose variables are not fixed in terms of data type and structure until execution. Thus data structures

may be modified without extensive modifications or re-translation of referencing procedures.

Some of the features of this language are similar to those found effective in existing languages: the use of libraries of loadable procedures from APL; the basic syntax from ALGOL 68 (3); the data types and operations from COBOL (4) and PL/1 (5);and data structures from CODASYL committee reports (1,6).

CHAPTER 2 DATA ITEMS

The first consideration in the design of the language will be the data items it will support and manipulate. The basic data types fall into several categories: arithmetic items, string items, list-processing items, procedure items, and aggregate items. References to data items may be made using constants, or variables (identifiers). A variable is a symbolic name having a value that may change during execution: a constant specifies a value which cannot change.

Arithmetic data items may be of several types: binary integers, henceforth called integers; decimal fixed-point items, called decimal items; and floating-point items, called reals. Each of these items has an associated precision or size. For integers and reals the precision is the number of units of memory the item occupies; for decimal items the precision is a pair of numbers, the total number of digits and the position of the decimal point. An integer constant consists of a field of decimal digits as in 26 or 01302. A decimal constant consists of

a field of decimal digits with a decimal point as in 26.35 or 39.0. A real constant consists of an integer or decimal constant followed by an "E", and followed by a possibly signed integer exponent, as in 15E2 (1500), 1.5E+3 (1500), 15000E-1 (1500), or 0E0 (0).

There are three types of string items: character, picture, and boolean. A character string is a contiguous sequence of characters recognized by the machine configuration. A picture string is the same except that information is provided specifying the accepted characters allowed in each position in the string. Both character and picture strings have an associated length, the number of characters in the string. A character or picture constant consists of the string of characters enclosed within single quotation marks. A single quotation mark within the string is represented by two adjacent quotation marks. The null string (string of length zero) is represented by two adjacent quotation marks. Examples of strings are: 'JOHN DOE', '' and 'JOHN''S BOOK'. Boolean strings are sequences of binary (logical) values. The length of a boolean string is the number of binary values

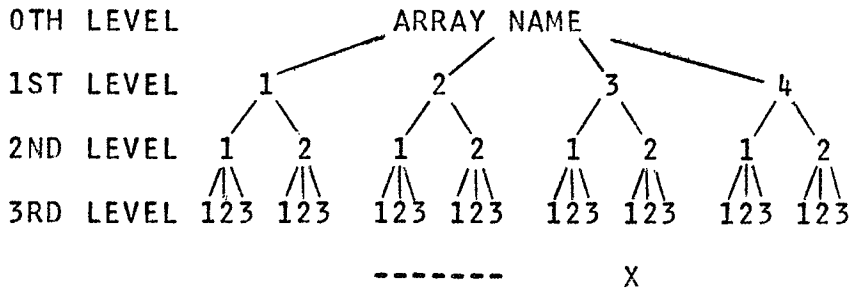
in the string. A boolean constant consists of an integer "n", the character "R" (standing for radix), and the constant expressed in base "n", as in 2R1011, and 16R3F70A2. Boolean strings of length one may also be represented by "TRUE" (1) or "FALSE" (0).

The basic list processing data item is the pointer. A pointer value is an address of a memory location. The precision of a pointer item is the number of units of memory which the address occupies. The only pointer constant is "NULL", representing a value which does not point to any memory location.

A procedure item is a set of statements of the language. It may be called or invoked and may have associated parameters which are given values when the procedure is to be executed. Procedures and procedure constants will be discussed in detail in chapter 5.

In many cases, it is desired to work with collections of data items or aggregates. The most common and useful aggregate is the array. An array is an n-dimensional

collection of elements of the same type and precision or length. The array is given a name, with individual items in the array being referenced by their relative positions in the array. The elements of an array are stored physically in contiguous words of memory. Conceptually, an array can be represented by a tree such as follows:

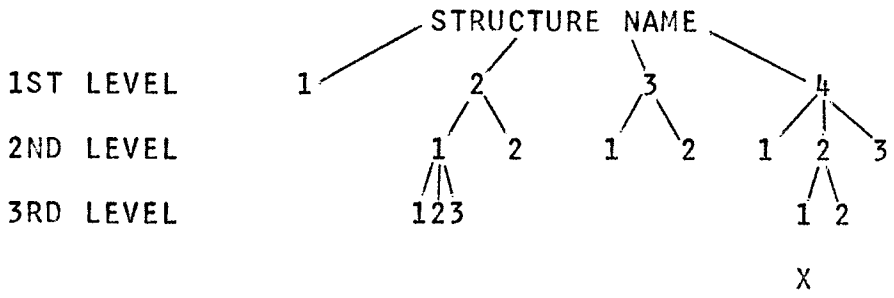


This tree represents a three-dimensional array of size 4 by 2 by 3. The number of branches from a node at the level of dimension "n" yields the extent of dimension "n+1". The nodes at each level (dimension) may be numbered starting at some integer called the lower bound and are numbered sequentially to the upper bound. Thus the extent of a given dimension is the difference between the upper and lower bounds plus one. The product of the extents yields the total number of elements in the array. A particular element is referenced by giving the number of

the node at the first dimension level, followed by the number of the node at the second dimension level, and so on. These numbers are referred to as subscripts. For example, to select the marked element in the above tree, the subscripts needed are 3,2, and 1 in that order. Sets of elements may be selected by giving only some of the subscripts. For example, the underlined elements above may be selected by giving only the subscript 2. These elements would in turn be presented in the form of a 2 by 3 array (that part of the tree from node 2 of the first dimension down becomes the tree for the resultant array of elements).

While arrays are very useful for representing lists, tables, and matrices a more general method is needed to handle collections of business data. Instead of requiring all elements of an aggregate to be of the same type and length, we will introduce a new aggregate in which the elements differ in type and length. In addition the aggregate need not be rectangular (each node at the same level may have a different number of descendants). Such an aggregate will be called a structure and corresponds to

the PL/1, COBOL or ALGOL 68 structures. A typical tree representation of a structure thus might be:



As for arrays, the elements are stored in contiguous memory locations; however, each node and each element is given its own subname. A particular element may be referenced using subscripts as for arrays (the subscripts for the marked element above are 4,2, and 1), or the subname of the element may be combined with the structure name using the qualification operator, "." (see chapter 3) to name the element directly. The subnames of the nodes and elements are internal to the structure and hence the same subnames may be used in many structures. A set of elements from a structure may be referenced as for arrays by giving a partial subscript list or by using the subname of the originating node for the set.

Structures are most useful for transferring related items to and from external storage media. An entire block of data can be read into a structure as a unit and then parts of it may be referenced either with subscripts or subnames. It can be seen that an array is a special case of the structure used for handling rectangular collections of items of the same data type.

Quite often within a program we wish to perform the same operation upon a set of unrelated data items. For example, we may wish to increment several counters at once, or we may wish to group several different items for an output line for a report. To facilitate this, we introduce another aggregate, the group. The group has the same tree description as the structure; however, the elements are any data items, both variables and constants, that we may wish to manipulate. In fact, they may be parts of other aggregates, or they may themselves be complete aggregates. We will provide named group variables for cases in which we wish to use a group several times and unnamed group constants, or group temporaries, when a group is required only once. Groups

are discussed in more detail in chapter 5.

In order to facilitate the use of data types other than the basic ones supplied in the language, we will allow the user to define his own data types in terms of the basic ones supplied. For example, a new data type for representing dates may be defined as a picture item consisting of two numeric digits, followed by a slash, followed by two more digits, another slash, and two more digits. A polynomial data item may be defined as a vector (one-dimensional array) giving the coefficients of the polynomial. These new data types would be referred to by their own user-defined names. We will later allow the definition of operations which may be performed upon these data types. For example, the regular subtraction operator "-" may be extended to work on date data types so that one may subtract two dates in the same manner as subtracting regular numeric quantities.

CHAPTER 3 OPERATORS

In order to manipulate data items a method of performing operations upon them is needed. For this purpose, we will define an entity known as an operator. An operator is a set of procedures together with an integer called a priority. Provided with the language will be a standard set of operators, such as those for addition of numbers, and concatenation of strings, and facilities will be provided for the addition of user-defined operators.

An operator may appear with a single data item only, such as the operator "ABS" which yields the absolute value of its operand, or it may operate on two items, such as the addition operator "+" which adds the values of its two operands. An operator used with one operand is known as a monadic operator, while a dyadic operator has two. A data type specification is associated with each member of the set of procedures for a monadic operator. When a monadic operator is to be applied to an operand, the data type specifications are searched for one matching the data type

of the operand. If a match is found, the procedure associated with that data type specification is invoked with the operand as an argument. If no match is found, then an error message will be generated. The rules for matching a type specification with the operand type will be given in a later section.

For dyadic operators, a similar technique is used: each procedure in the set has two associated data type specifications. In order for a procedure to be selected from the set, both specifications must match the operand types. This selection of procedures based on data type matching is very powerful. A single name may be used to specify a logical operation - regardless of the types of data supplied as its operands. For example, as mentioned in a previous chapter, the subtraction operator "-" may be extended to be used with a user-defined date data type so that a subtraction of two dates may be expressed in as natural a manner as the subtraction of ordinary numeric items.

One operator may represent both dyadic and monadic

operations, depending upon the context in which it is used. For example, "-" used with one operand produces the negative of its operand; with two it calculates their difference. Any one of the procedures associated with an operator may be changed to affect its operation with the associated data types. New procedures could be added to the set to support the operator's use with new data types.

A set of basic operators will be built into the language to provide the operations which are most often used and which therefore should be implemented by procedures written in machine code. The monadic operators provided include:

¬ provides the boolean negation of a boolean string.

- takes the negation of numeric data items.

ABS produces the absolute value of numeric items.

@ produces a pointer item giving the address of its

operand.

_ takes a character string operand and causes it to be scanned and executed just as if the character item had appeared in the source program. This allows for source statements of the language to be created dynamically as character items and then to be executed. For example, statements of a simple user language may be translated into statements of this language for execution. Descriptions of structures used with files may be read in with the file and may then be executed. This is a very powerful operation which appears in other interpretive languages such as SNOBOL (7).

The standard dyadic operators include the following:

+ adds two numeric operands.

- subtracts two numeric operands.

* multiplies two numeric operands.

/ divides two numeric operands.

** raises the first operand to the power specified by the second.

|| concatenates two string items together.

& produces the boolean "and" of two boolean strings.

| produces the boolean "or" of two boolean strings.

= yields "TRUE" if the first operand has the same value as the second; or "FALSE" otherwise.

¬= yields "TRUE" if the two operands have different values.

> yields "TRUE" if the value of the first operand is greater than that of the second.

>= yields "TRUE" if the value of the first operand is greater than or equal to the second.

< yields "TRUE" if the value of the first operand is less than that of the second operand.

<= yields "TRUE" if the value of the first operand is less than or equal to that of the second operand.

:= converts the value of the second operand to the data type of the first and assigns this value to the first operand.

+:= the first and the second operands are added and the result is assigned to the first.

-:= the second operand is subtracted from the first and the result is assigned to the first operand.

*:= the first and second operands are multiplied together and the result is assigned to the first operand.

/:= the first operand is divided by the second and the result is assigned to the first operand.

. both operands must be variables; the first must specify a structure or group; the second the name of an element within the structure or group. The result is the value of the element with the name given by the second operand which is found in the aggregate specified by the first operand. This is used for selecting structure elements which have the same subnames as elements in other structures.

FORMAT The result is a character string resulting from formatting the first operand according to the value of the second operand. This is very useful for producing reports with inserted dollar signs etc. The particular values of the second operand and the supplied formatting routines will not be discussed in this thesis.

CHAPTER 4 PROGRAM STRUCTURE

Now that the data types and operations desired have been specified, we will define the structure of the language. The syntax of a language is dependent upon its intended use: a language such as APL uses a non-hierarchial syntax in that we perform a series of operations in sequence with little decision-making. A hierarchial structure is more appropriate for a language involving many (hierarchial) decisions. A hierarchial or nested block structure is also useful for expressing aggregates, since aggregates are hierarchial data structures.

The first consideration is the naming of variables, operators and data types. We will use as delimiters the reserved symbols ( ) , ; : ' ? " and the blank. The other symbols will be divided into two classes: alphanumeric, consisting of the letters, digits, and the dollar sign; and special, consisting of all other characters supported by the machine architecture. A name will consist of a

sequence of alphanumeric characters, with the first character other than a digit, as in A , A23 , $LABEL , and A23P6 ; or it may consist of a sequence of special symbols as in +:= , @@ or @.+&_* . We will generally use alphanumeric names for variables and data types and special symbols to name operators. Certain sequences of symbols will be reserved and may not be used by the programmer as names.

The basic syntactic unit of the program is the expression, which consists of a sequence of operators and operands. A monadic operator must appear preceding its operand and a dyadic operator between its operands as in A&¬B&C. The order of evaluation of an expression is dependent upon the priorities of the operators as in PL/1, ALGOL, or FORTRAN. Monadic operators are applied to their operands first, from right to left when two monadic operators are to be applied in turn to one operand, as in:

ABS _ A .

If two or more dyadic operators of the same priority appear the leftmost one is evaluated before the rightmost one. We will choose left to right since this is the

common direction used in most hierarchial languages such as PL/1 and ALGOL: it also allows for simple interpretation. Operators of higher priority are evaluated before those of lower priority. For example, if * has a higher priority than + then A+B*C results in B*C being evaluated with the result then being added to A. A+B*C*D results in the product of B and C being multiplied by D with this result then being added to A. Parentheses may be used to modify the order of evaluation as in (A+B)*C which causes the addition to be performed before the multiplication. An installation will supply default priorities for the built in operators.

In this language every executable construct is an expression and computes a value. In a similar manner to BLISS (8) there are no statements: expressions may be joined with semicolons to form compound expressions, or paragraphs. The value of a paragraph is its last component expression. Thus the result of executing the paragraph:

```
A:=3;B:=5;A+B
```

is eight. A paragraph may be enclosed in parentheses to form a simple expression as in:

```
2+(A:=3;B:=5;A+B)
```

which yields a resultant value of ten. Note that when the result of evaluating an expression is a variable, the result is the name of the variable and not its current value. Thus the following expression assigns the value 10 to the variable A:

```
(B:=5;A):=10
```

In order to control the order in which expressions are evaluated we will define a special expression, known as a conditional. A conditional consists of either two or three paragraphs separated by "?" symbols and enclosed in parentheses as in:

```
(I+J ? A;B ? C)  or  (X=0?A)
```

The evaluation of a conditional is as follows: The first paragraph is evaluated and the resultant value is converted to an integer value. If this result is less than one or greater than the number of expressions in paragraph two then the second paragraph is skipped and the third paragraph is evaluated to yield the result. If the third paragraph, the OUT clause, is omitted then the result of the conditional is zero. For example, in:

```
(I?1;2?3)
```

if I is not positive, or is greater than 2 then the result is the third paragraph, or 3. In (I?A), if I is not a one, then the result is zero. Note that the evaluation of a logical expression such as X<6 yields a boolean value "TRUE" or "FALSE". A true value is converted to the integer 1; a false to a zero, to give the integer test value. If the integer test value, "i", lies in the range 1 to "n" where "n" is the number of expressions in paragraph two, the IN clause, then the "i"th expression of the IN clause is evaluated to yield the result of the conditional. For example, in:

```
(I?2;1;10?-1)
```

the resultant value is 2 if I is a 1, 1 if I is a 2, 10 if I is a 3, or -1 otherwise. Note that each of the expressions in the IN clause may be compound expressions (or other conditionals) as in:

```
(I?(J?2;(A:=3;A+6));8?-1)
```

the result of the evaluation would be:

2 if I and J are both 1;

9 if I is 1 and J is 2. (9=A+6)

0 if I is 1 and J is less than 1 or greater than 2.

8 if I is a 2.

-1 if I is not 1 or 2.

An implementation may choose to substitute the words IF,THEN,ELSE,FI or CASE,IN,OUT,ESAC for (,?,?,) as in:

```
IF I=0 THEN A ELSE B FI    for (I=0?A?B)
```

or `CASE I IN 2;3;1 OUT 0 ESAC` for (I?2;3;1?0).

Since a conditional is an expression, it may appear anywhere as in:

```
(X=0?A?B):=6
```

which assigns the value 6 to either A or B depending upon the value of X.

Quite often it is desirable to repeatedly evaluate a set of expressions. This facility will be provided by a special operator, the "DO" operator. To repeatedly execute an expression, it is preceded by "DO" as in:

```
DO X+:=1
```

or `DO (X:=3;A+:=X)`

In the first example X+:=1 will be repeatedly executed, adding 1 to X each time;in the second example 3 will be added to A each time. In each case, we have no way of stopping the execution: we will introduce a special construct, the EXITDO, for this. The word "EXITDO" is

optionally followed by an expression as in:

```
DO ((I+:=1)=10 ? EXITDO X ? X+:=I)
```

When an expression consisting of an EXITDO is encountered, the loop is terminated and the result of the DO operation is the expression supplied with the EXITDO. In the above example, if X and I were previously set to zero, the loop would be terminated when I is a 10, and the result of the DO would be X, in this case with value 45, the sum of the integers from 1 to 9. (note that this example is not necessarily the best way to calculate this sum). If the expression is omitted from the EXITDO then the result of the DO operation is the value of the last expression previously evaluated within the loop. Another construct, the SKIP construct, is used to terminate the current iteration of the loop and to continue with the next iteration as in:

```
DO ((I=0?SKIP); ... )
```

which skips the evaluation of the loop if I has the value zero.

In order to make the DO loop more useful, a means of controlling the execution of the loop expression is

provided: the DO operator may have a preceding operand specifying a list of values with which the loop expression is to be evaluated. For example:

```
(10,2,3) DO X+:=1
```

The expression X+:=1 will be evaluated three times, once with a special control variable set to 10, the second time to 2, and the third time to 3. (The use of commas to describe a list of items will be described in greater detail in chapter 5: a means of accessing the special control variable within the loop expression will also be described later). The result of the DO operation is the value of the last expression evaluated within the loop. For example, in:

```
X:=0; A:=2+((1,2,3) DO X+:=1)
```

the value assigned to A would be 5.

Since the use of commas to specify a list can be tedious for large lists, a special operation is provided for generating vectors. In its simplest form, this consists of : followed by an expression as in :N. This operation would generate all the integers from 1 up to N. Thus:

```
:10 DO X+:=1
```

would cause the loop to be executed 10 times. A different starting value than 1 for the vector may be specified by preceding the : with an expression as in I:N. This yields a vector of all the integers from I up to N. An increment may be specified as in I:K:N. In this case, the integers from I up to N incrementing by K would be generated. For example, 1:2:7 is equivalent to (1,3,5,7). The increment need not be positive as in 7:-2:1 which generates the list (7,5,3,1). In fact, when an increment is specified, the vector need not consist of integers, as in 0.0:0.1:0.5 which generates (0.0,0.1,0.2,0.3,0.4,0.5). Thus, the : operator may be combined with the DO to give the same iterative facilities as the corresponding PL/1 and ALGOL 68 DO statements. Note that, since in this language DO is an operator, and not a statement, we may use it anywhere an expression is allowed as in: A:=2+(:3 DO X+:=1). A deeper discussion of the iterative capabilities of the DO operation will be discussed after procedures have been introduced.

One other means of modifying the sequential flow of

control is commonly provided in other languages: the branch or "GOTO", This form of control will not be included for several reasons: the high degree of program structuring using parentheses, especially in conditional expressions, obviates the need for explicit branches. Secondly, in an interpretive environment, branches are time-consuming to support, due to the possibility of jumping out of blocks and due to the disruption of the natural structuring. Also, in an expression language such as this, the use of branches leads to awkward situations such as jumping into an expression which is used as an operand in another expression. These situations are quite difficult to detect, with the result that large sections of the interpreter which would be needed to handle these cases would very rarely be used. It is the author's opinion that explicit branches are not needed in this language and their use would lead to programs which are difficult to understand and debug.

CHAPTER 5 AGGREGATES

In chapter 2 we mentioned aggregates in fairly general terms. Now that the basics of the language have been described we will turn our attention to the implementation of aggregates. This chapter will concern itself with the defining of group and array temporaries and the selection of elements from aggregates: the method of defining aggregates will follow in chapter 7, as well as their use with operators.

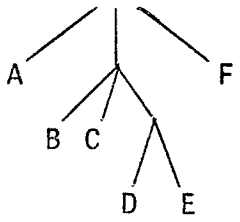
To begin, we will discuss the creation of group temporaries since they are a basic and important construct. A group temporary is created by listing its elements, separated by commas and enclosed in parentheses as in:

```
(A,B,C)
```

which forms a one-level group (vector) with elements A,B, and C. To create groups containing several structural levels (hierarchies), parentheses are used as in:

```
(A,(B,C,(D,E)),F)
```

which has the structuring:



An empty group may be created using (). An empty element may be created by omitting it from the list as in:

(A,,C)

in which the second element is empty. Empty groups are useful for procedure argument lists; empty elements are useful in subscripting. Note that the elements of a group temporary need not be simple variables; they may be described by expressions as in:

(A+B*C,3.8)

Each expression is evaluated before the group is constructed; the elements being composed of the values resulting from the evaluation of their expressions.

During our discussion of the DO operator we mentioned the use of the : to generate vectors. This construct is not just applicable to DO loops; it may be used in more general contexts to generate one-dimensional array temporaries. When used as a standard vector it acts just like the corresponding group temporary. Thus:

1:2:5 and (1,3,5)

could be used interchangeably although an implementation would probably optimize the memory requirements of the : construct.

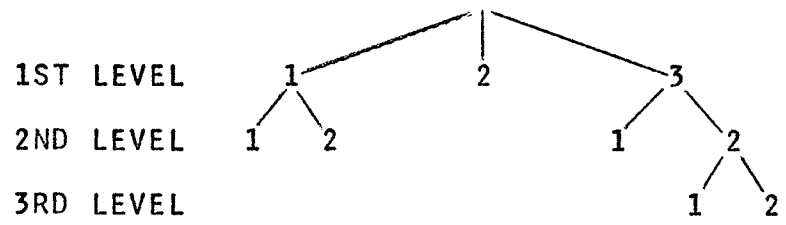
The DO operator is not restricted to the simple iterative control mentioned in the previous chapter. The control operand may be any one-dimensional aggregate. The control variable will take on the elements of this aggregate in turn for each evaluation of the DO expression. Thus:

(10,A+B,5:-1:3,0) DO ...

would cause the control variable to take on the values 10,A+B,5,4,3, and 0 in turn. This allows for much greater flexibility than the simple iteration of, say, a FORTRAN DO-loop.

Having described the methods for generating aggregate temporaries, we are now ready to discuss the selection of particular elements from aggregates. A general selection technique, applicable to all aggregates, is the subscript list mentioned in chapter 2. A subscript list consists of

a one-dimensional aggregate whose elements give the numbers of the branches to select from the tree structure of the aggregate. For example:

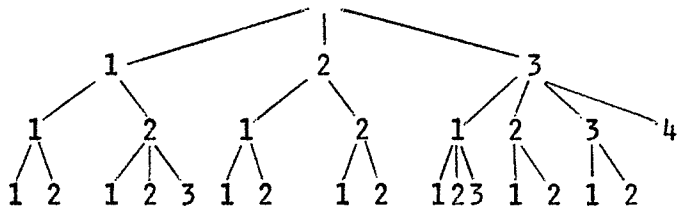


The subscript lists for the elements, from left to right, are: (1,1),(1,2),(2),(3,1),(3,2,1), and (3,2,2). To use a subscript list, it is enclosed in parentheses and is placed immediately after the aggregate to which it refers. For example, if the above aggregate were named A, then the elements would be obtained by coding A(1,1), A(1,2), and so on. The subscript list need not be a group temporary; for example, if B named an aggregate with elements 3,2,and 1 then A(B) and A(3,2,1) would be equivalent. The use of aggregate variables as subscripts is a very powerful construct: subscript lists may be manipulated much more readily than in the more common construct which requires the specification of each subscript independently.

A subscript list need not totally specify all

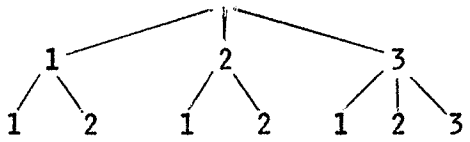
branches in the tree structure: a partial list may be used to select a set of elements. This is best illustrated with an example: referring to the above tree structure, A(3,2) would select both A(3,2,1) and A(3,2,2); A(3) would select A(3,1), A(3,2,1), and A(3,2,2). The resultant value would in turn be an aggregate, with tree structure that of the selected subtree. Such an aggregate could in turn be subscripted as in A(3)(2,1) which would be equivalent to A(3,2,1).

Other sets of elements may be selected by omitting leading and middle subscripts from the subscript list as in A(,1) which selects A(1,1) and A(3,1) in the form of a two element vector. (these omitted subscripts were the reason we allowed group temporaries to have empty elements). As a more complicated example, consider the following structure:



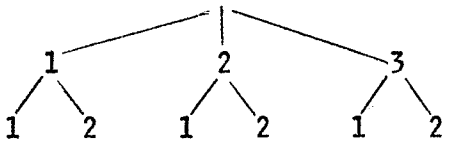
If this structure were named B then B(,,1) would select

B(1,1,1),B(1,2,1), B(2,1,1),B(2,2,1),B(3,1,1),B(3,2,1), and B(3,3,1). The structure of the selected set of elements would be:



In other words, the resulting structure is formed by filling in all possible permutations of the omitted subscripts. The above tree structure would be used if we wished to subscript the resultant selection in turn: for example, B(,,1)(3,2) would reference the same element as B(3,2,1).

Another means of selecting sets of elements is provided: each subscript position may be specified as a list of values as in B(3,,(1,2)) which would select the elements B(3,1,1),B(3,1,2),B(3,2,1),B(3,2,2),B(3,3,1), and B(3,3,2) in the following resultant tree structure:



Thus we are using a two-dimensional aggregate as our

subscript list to give lists of particular subscripts to pick out. If we had specified B(3,,(2,1)) in the above example, the resultant structure would be the same but the elements would be, from left to right:

B(3,1,2),B(3,1,1),B(3,2,2),B(3,2,1),

B(3,3,2), and B(3,3,1).

Thus, to summarize, an omitted subscript acts just as if a list of all the possible subscript values were provided:this action is sometimes known as a slice and is most useful for selecting rows and columns of matrices. For each particular subscript we may provide a list of values to order the selection and to choose smaller subsets. Thus B(3,,(1,2)) is equivalent to B(3,(1:3),(1,2)); B(3,(3:-1:1),(2,1)) would exactly reverse the order of selection of the elements while maintaining the same logical structuring of the result. In order to determine the tree structure of the result we examine the selected elements as follows: the first subscript position that varies determines the number of branches at the first level; the second varying subscript the next level, and so on.

Another more specialized selection technique exists for groups and structures. In these aggregrates the elements may be named: in structures all elements as well as the nodes (sub-structures) are named; in groups they may or may not be named. A particular element is referenced by simply using its name. For example, in the group (A,B,C), A refers to the first element, and so on in obvious fashion. In structures, the situation is somewhat more complex in that the same name may be used for elements in two different structures. We identify the element we wish to reference by qualifying its name with the structure name. For example, if X is the name of an element in structure A then A.X would be used to reference it. If A were in turn a substructure of a structure B, then B.A.X would be used to refer to X. Note that the "." is an operator and hence its operands may be expressions as in:

```
(Q=0?A?B).X
```

which selects the element X out of either structure A or B depending upon the value of Q.

CHAPTER 6 PROCEDURES AND ARGUMENTS

One of the most useful and powerful facilities in a computer language is the ability to define and name subroutines or procedures. A procedure in this language will consist of a set of expressions which together perform some action such as calculating the maximum of a set of numbers. Procedures also provide a mechanism for controlling the scope of names (see chapter 7) and allow the programmer to write his total program in segments which are easy to debug and document. One of the initial goals mentioned in chapter 1 was the ability to store routines on external libraries in order that many different users may have access to them; procedures provide this facility.

A procedure constant will consist of a paragraph, preceded by a list of parameters enclosed in double quotation marks, and all enclosed in parentheses as in:

```
("X,Y,Z" A:=2;Z:=(X+Y)*A)
```

A named procedure is constructed by assigning a procedure

constant to a variable of type procedure: the variable name may then be used to represent this procedure until another procedure constant is assigned to it.

The appearance of a procedure constant or variable does not cause the procedure to be evaluated: it is performed when invoked. To invoke a procedure, a list of arguments is provided in a similar manner to subscripting as in:

```
("X,Y" X+Y)(2,3)
```

and

```
P(A,B,C)
```

The argument list may be any aggregate having the same number of direct descendants as the number of parameters in the procedure: it is most commonly specified as a group temporary. To evaluate the procedure, the first parameter variable (each parameter must consist only of a variable name) assumes the data type and value of the first argument (the first element in the aggregate), the second parameter the type and value of the second argument, and so on. The paragraph of the procedure is then evaluated; the resultant value returned by the procedure being the value of the last expression evaluated in the paragraph.

For example, a procedure to compute the minimum of its arguments might be:

```
("X,Y" (X<Y?X?Y))
```

If MIN is a procedure variable then this procedure could be assigned to it by:

```
MIN:=("X,Y" (X<Y?X?Y))
```

Note that if a procedure consists only of a conditional a particular implementation may allow the omission of the second pair of parentheses.

To compute the minimum of variable A and the value 1 we would use MIN(A,1): this value could then be used as an operand in an expression as in:

```
B:=2+MIN(A,1)
```

An argument may itself be an entire aggregate, as in:

```
P((A,B,(C,D)),2)
```

The first parameter would become the group temporary (A,B,(C,D)); the second the value 2. A procedure may have no parameters as in:

```
INIT:=("" A:=0)
```

which would be invoked by:

```
INIT()
```

Zero-parameter procedures are useful for initializing

variables or for making system enquiries such as requesting the time of day.

The parameters of a procedure are special local or dummy variables. That is, if X is used as a parameter, then any references to X inside the procedure refer to the value of the associated argument passed to X. References to a variable X outside the procedure refer to a different variable. However, if the value of a parameter variable is changed, the value of its associated argument is also changed, as in the following degenerate example:

```
A:=0;("X" X:=1)(A)
```

which results in A being assigned the value 1. Of course, X could not be passed a constant as its argument. This technique of having the parameters assume the type and value of their associated arguments is very powerful. A general procedure which does not depend upon the data types of its arguments may be used to compute the same function, regardless of the type of its arguments. For example, the procedure determining the minimum of its arguments given above would work equally well for any values accepted by the less than operator. When

contrasted to the techniques used in many languages such as PL/1, ALGOL 68, or FORTRAN which require their parameters to have explicitly stated data types it can be seen that much generality has been gained.

A procedure variable may represent an array of procedures. For example, if Q were a 2 by 2 procedure variable then:

```
Q(1,1):=("A,B" A+B)
Q(1,2):=("A,B" A-B)
```

would assign different procedure constants to the elements Q(1,1) and Q(1,2) of Q. The same result could be obtained with:

```
Q(1):=(("A,B" A+B),("A,B" A-B)).
```

To invoke an element from a procedure array, both a subscript list and an argument list are provided as in Q(1,2)(X,Y) which computes the difference of X and Y. Q(1)(X,Y) would evaluate the first row of Q with arguments X and Y. When a section of a procedure array is evaluated, all of the procedures are evaluated with the resultant values combined in an aggregate having the same logical structure as the section of the procedure array. The

above example would thus yield the sum and difference of X and Y as a two element vector. Q(,)(X,Y) would evaluate the entire array yielding a two-dimensional result.

A procedure may invoke itself. If such a recursive procedure has parameters then at each invocation of the procedure, the parameters refer to the new arguments passed at this invocation. For example, if the following procedure is assigned to the variable FAC:

```
FAC:=("I" (I<=1?1?FAC(I-1)*I))
```

then FAC could be used in a recursive manner to compute factorials. FAC(3) would recurse twice: once to compute FAC(2) and once again to compute FAC(1).

Normally, the evaluation of a procedure terminates after the last expression in its paragraph has been executed. However, in order to allow the termination at any point within the paragraph, a special exit mechanism is defined. This consists of the keyword "EXIT" optionally followed by an expression. Its action is identical to the EXITDO construction described in chapter 4. For example, if we wish to exit from a procedure

immediately with a zero result if the value of the argument is zero we might use:

```
("A" (A=0?EXIT 0);...)
```

Procedures have a special use with the DO operation. Earlier we have mentioned the use of a DO with a special control variable without mentioning how such a variable is implemented. When we wish to use the control variable within the DO expression, we replace the expression with a one-parameter procedure (without any argument list) as in:

```
1:2:9 DO ("I" SUM +:=A(I))
```

Each time the DO expression is to be evaluated, the current value of the control variable is passed as the argument to the procedure which is then evaluated. In the above example, the action would be to add A(1),A(3),A(5), A(7), and A(9) to the variable SUM.

Just as we defined a standard set of operators best implemented in machine code, so we will define a standard set of procedure variables, representing a set of internal procedures. Such a set of procedures would probably include the following: (the list of arguments after each

name is provided to indicate the number of arguments needed and to indicate the argument values expected). We will not include implementation-dependent functions used for source editing, error-correction, and maintenance of procedure libraries.

INDEX(S,C) used to locate a substring C, of a given string S. The result is an integer giving the position of the first occurrence of substring C within S (or zero if the substring does not exist within S).

VERIFY(S,C) verifies that all the characters in S come from the set of characters given by C. The result is an integer giving the position within S of the first character not found in C (or a zero if all characters in S appear in the string C).

REPEAT(S,I) for string items, the string S is concatenated with itself I times: for other items, a vector of extent I is created, with each element a copy of S.

SUBSTR(S,I,J) the result is the substring of S starting at position I and extending for J positions (or the character string of length zero if the substring does not exist).

LENGTH(S) computes the number of memory locations occupied by S.

ALL(X) returns "TRUE" if all elements in X consist entirely of "TRUE" values, and returns "FALSE" otherwise.

ANY(X) returns "TRUE" if any elements of X are "TRUE", and returns "FALSE" otherwise.

ELT(X) returns the total number of elements in aggregate X.

SELT(X,I) treats the aggregate X as a vector and selects the Ith element from it.

DESC(X) returns a count of the number of direct

descendants of X.

SDESC(X,I) selects the Ith direct descendant of X.

DIM(X) returns a count of the number of tree levels of aggregate X.

HBOUND(X,I) returns the upper bound of the Ith dimension of array X, or zero if the Ith dimension does not exist.

LBOUND(X,I) returns the lower bound of the Ith dimension of array X, or zero if the Ith dimension does not exist.

CNFRM(A,B) returns "TRUE" if A and B are of the same data type, and have the same logical structuring.

SELECT(X) returns the position of the first boolean value within X which has value "TRUE".

The following procedures allow input and output from the user's terminal.

PRINT(X) the elements of aggregate X are converted to character representation and are listed on the terminal.

PRINTC(X) X is dumped as is on the terminal. (used for listing character items, such as messages to the user)

READ(X) data values are read from the terminal until as many elements as are in X have been read. The values must be valid constants separated by blanks or commas. These values are assigned to the elements of X in turn.

READC(X) a complete input line from the terminal is read as is and is assigned to the character argument X.

CHAPTER 7 DECLARATIONS AND OPERATOR SELECTION

Now that we have described the basic constructs of the language we are ready to describe the methods used for defining or declaring variables. We will differ from most languages in that declarations will be executable. That is, the data types and structuring of variables will be determined dynamically and may be changed during execution of the program.

Declarations serve two purposes: they define a data type for a variable, and they allocate memory locations to store the value associated with the variable name. We will discuss the assigning of data types first. Each of the basic data types is given a special declaration name:

INT for integers,

REAL for floating-point,

DEC for decimal,

CHAR for character,

BOOL for boolean,

PICT for picture,

PNTR for pointer,

and PROC for procedure.

To declare a variable, the desired declaration name is followed by the variable name as in:

```
INT A
```

A set of variables may be declared together by parenthesizing them in a list as in:

```
REAL (X,Y,Z)
```

If a variable had previously appeared in a declaration then its previous data type, value, and memory locations are lost. A declaration may be used as an operand, usually to assign it a value, as in:

```
INT I:=2
```

Before continuing, it is appropriate to describe the scope of a variable. Once a variable has been declared within a procedure any references to that variable refer to the current value of that declared variable. Any references to a variable of the same name outside the procedure refer to a different variable. A reference to a variable which has not been declared within the same procedure refers to the variable with that name in the calling procedure: if no such variable has been declared,

then the next procedure back is used and so on. If a variable of that name has not been declared in any calling procedure then a reference to it is in error. As mentioned previously, parameters of a procedure are automatically local to that procedure. For example:

```
("...INT X ... B(1,2))
  B:=("Y,Z" ... X ... C(5))
    C:=("I" REAL Z:=Y+X)
```

X is common to all three procedures: Y is common to the last two procedures: the Z's are different variables in the last two procedures.

Associated with a variable in addition to its type, is its precision or length (except for type procedure). In the examples given above, no precision was stated: thus an implementation-defined default precision is applied. In order to override the default a precision specification may be provided using a parenthesized paragraph preceded by the symbol "#" and following the declaration name as in:

```
    INT#(4) A
or REAL#(INT ALEN:=X/2) B
```

In the case of decimal items, where a precision consisting of two values is allowed, the length specification may evaluate to a two-element aggregate to specify both values; or a scalar may be used in which case the number of decimal digits is set to zero. For example:

```
DEC(5,2) MONEY
```

could be used to represent amounts of money up to 999.99 . For picture items, the length specification must evaluate to a character item: the length of this item gives the length of the picture item; the value describes the characters allowed in each position of the picture in a similar manner to PL/1 or COBOL pictures. The possible picture specifications will not be described in this thesis.

In order to declare an array, a dimensioning specification is inserted preceding the declaration name. The dimensioning specification consists of a parenthesized paragraph. When the declaration is encountered the paragraph is evaluated. The result must be a scalar or a one or two dimensional aggregate. The number of elements at the first level yields the number of dimensions of the

array: the values specify the upper and lower bounds of each dimension. If a one-dimensional aggregate is specified each value gives the upper bound of its associated dimension: the lower bounds are set to 1. If a two-dimensional aggregate is specified, there may only be two descendants at the second level from each node at the first level. These two elements give the lower and upper bounds for the corresponding dimension. For example:

```
(2,(0,3),4) INT A
```

specifies a three-dimensional array with upper bounds 2,3, and 4 and lower bounds 1,0, and 1. Thus the subscript for the first element would be (1,0,1); for the last (2,3,4). The dimension specification need not be a group temporary: it may be any aggregate with the proper logical structuring. Note that while an array may have lower bounds other than 1, if we select a subaggregate from it using subscripting the result would be numbered starting from 1. Thus, in the range of:

```
(2,(0,3),4) INT A
```

then A(1,,2)(1) would refer to A(1,0,2).

Up to this point we have not mentioned the allocation

of memory to a variable. When a declaration is encountered, any previous units of memory allocated to the variable by a previous declaration within the current block are released, and the necessary memory locations are allocated. When control returns from a procedure, all memory belonging to variables declared within that procedure is released.

In many cases we wish two variables to refer to the same memory locations, to refer to an item in two different ways, or to pick out a section of an item. To implement this, we override the normal allocation of memory by inserting a basing specification. A basing specification consists of a parenthesized paragraph preceded by the symbol "." and inserted after the declaration name either before or after the length specification. When the declaration is evaluated the basing specification is evaluated. If the result is not of type pointer, then the declared variable will occupy the same memory locations as the basing result. If this result is of type pointer, then it specifies the address of the area of memory that the declared item will occupy.

For example:

```
INT#(2).(INT#(4) B) A
```

specifies that A is 2 memory locations long and occupies the first two memory locations of B.

```
CHAR.(@BUFFER+2) SCANCHAR
```

specifies that SCANCHAR is a character item occupying the third memory location in BUFFER. Note that to base a variable upon a pointer variable then a pointer to the pointer variable would have to be specified.

Sometimes it is useful to be able to dynamically alter the address of a based variable. For example, we may wish to scan across a character item: a single character item may be based on the current character being examined. We supply a procedure as the basing algorithm; it is evaluated for every reference to the variable to yield the current location of the variable. This feature is also useful for data item security checks. Since the procedure is evaluated every time the variable is referenced it can be used to check if access to the variable is allowed. The procedure may have up to four parameters: the first will be passed the user

identification established when he signed on the system, the second the variable name, the third the boolean value "TRUE" if an attempt is being made to change the value of the variable, or "FALSE" if it is just being read, and the fourth a MODE variable (discussed later in this chapter) specifying the data type of the variable. If the procedure returns a value NULL then the access to the variable will be denied. For example:

```
INT.("" A(M)) AM
```

declares AM to occupy the same memory locations as element A(M). Thus, if M is changed AM will effectively move with it. Note that the parameters may be omitted if they are not used or a partial list may be specified as in:

```
("ID,VNAME" ... )
```

If a basing specification is provided in either of its two forms then the length and dimension specifications may be provided as two-parameter procedures. When the variable is referenced these procedures are evaluated to yield the current length or dimensioning values. This allows for the support of variable length string items and variable sized arrays. The parameters, if provided, will

be passed the variable name and a MODE variable giving the data type of the variable as the arguments. For example,

```
CHAR.(BUFFER)#("" LEN) SECTION
```

declares SECTION to be based on BUFFER and to have a variable length whose current value is given by variable "LEN".

These techniques for declaration are generalized for the declaration of structures and groups. To declare a structure the declaration name "STRUCT" is used. It must be followed by a parenthesized list of declarations giving the element items, as in:

```
STRUCT(INT I,REAL X) S
```

which defines a one-dimensional structure S having as elements I and X. The declarations of the elements are all processed together and the elements are assigned consecutive memory locations (thus they must not be based). To declare a multi-dimensional structure, STRUCT declarations are nested as in:

```
STRUCT(INT I,STRUCT(INT(A,B)) SS) S
```

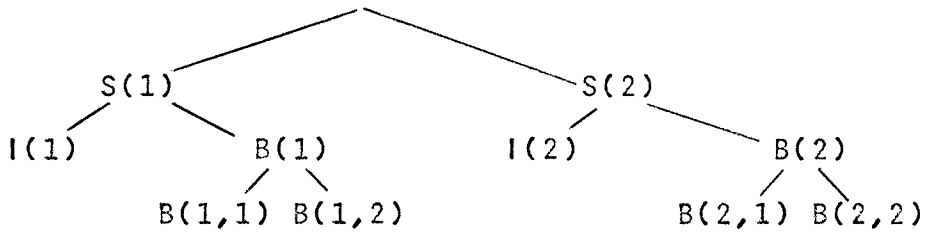
As mentioned previously the names of the elements are internal to the structure: the qualification operator "."

is required to access them.

A structure may itself be an array. In this case, each element in the structure has the dimensions of the structure added preceding its own dimensions. For example:

```
(2) STRUCT(INT I,(2) INT B) S
```

has the effect of making I a two-element variable, and B a 2 by 2 matrix. The tree structure for S would be:



A structure may also be based using either of the two forms of the basing specification. In this case, the elements are assigned contiguous memory locations starting at the location specified by the base. If a base is provided, then the element list may be a procedure, allowing for dynamic alteration of the logical structure of the structure variable. This procedure may have one parameter, with the name of the structure as its argument.

We may also declare named groups, using the

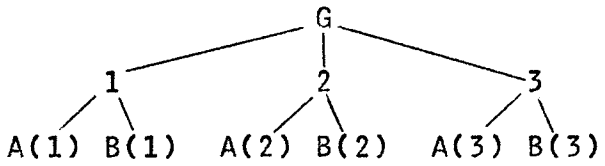
declaration name GROUP. The keyword GROUP must be followed by a procedure or parenthesized paragraph to describe its elements. However, the evaluation of this paragraph need not result in a list of declarations: group elements may be any values (except that a group may not contain itself). A group may not be based, but any of its elements may be based, since a group only implies logical structuring and not memory allocation. A one-parameter procedure may be used to dynamically describe a group in a similar manner to structures.

Groups may be arrays, but in a different manner to other data types. Dimensioning is used to select subsets of elements and to reorder them. Each element of a group must have at least as many dimensions as a containing group and the extents of corresponding dimensions of the group must lie within the extents of all the elements. The group then contains the sections of its elements selected by its dimensioning specification. For example:

```
(10) INT (A,B);
(3) GROUP(A,B) G
```

selects the first three elements of A and B into the group

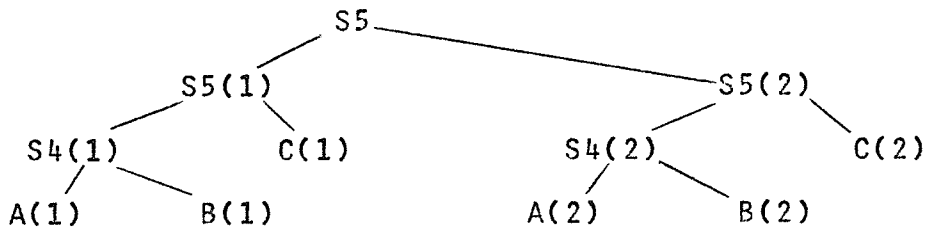
G. The elements are ordered by the group dimensioning. For example, the above group G has the logical structuring:



Thus G(1) selects both A(1) and B(1). When a group with "n" specified dimensions is subscripted, the first "n" subscripts select the first "n" dimensions of the elements; additional subscripts select the particular element variable. In the above example, G(1,1) selects A(1); G(1,2) selects B(1). For a more complicated example, consider:

(2) GROUP((3) GROUP((4,3) INT (A,B)) S4,(2) INT C) S5

then S5(2,1,1,3) refers to A(2,3). The "2" specifies the first dimension of the elements of S5 (i.e. S4(2) and C(2)), the first "1" selects S4 (a two would select C), the second "1" selects "A" out of S4, and the "3" specifies the second dimension of "A". The logical structuring of S5 would be given by:



Note that S5 only contains part of S4 (the first two sections). A procedure may be provided for the dimensioning specification of a group to specify the dimensions dynamically.

These techniques for combining elements into groups with a dimensioning capability is a new and very useful capability. It allows for the combining and possible reordering of items into logical structures. Groups also provide an effective way of specifying subscript and argument lists allowing for dynamic alteration of complete lists. In most computer languages elements of these lists must be inserted independently, thus making changes to an entire list more difficult to specify.

In order to simplify the use of user-defined data types (such as dates) we will provide a capability for the

definition of new data types. Complicated declarers such as the structure S5 in the above example may be represented by their own names. A special mode variable is declared using the mode assignment which is a special statement allowed in a paragraph. A mode assignment has the form:

```
MODE list := declarer
```

"list" is a single name or a list of names in parentheses, specifying the names to be given to the data type. "declarer" is a standard declaration, omitting the variable list, as in:

```
MODE VECTOR:=(5) REAL;
MODE REC:=STRUCT(VECTOR X,INT COUNT)
```

VECTOR may then be used in place of (5) REAL to declare a vector of reals; REC may be used in place of

```
STRUCT(VECTOR X,INT COUNT).
```

These user-defined modes may then be used as mode declarers as in VECTOR X, with the exception that if a user-defined mode contains a length or base specification then another such specification cannot be used with the mode. For example, it is in error to use:

```
MODE LINT:=INT#(4);
```

```
LINT#(2) X
```

However, an additional dimension specification is allowed as in:

```
(10) VECTOR X
```

which declares X to be a vector of vectors. Thus the two dimension specifications are combined into one. Due to implementation difficulties this feature would probably only be allowed for non-procedure (non-dynamic) dimension specifications.

A mode variable is a dynamic entity since it may appear in more than one mode assignment. Thus a mode variable is made to represent a new data type by its redefinition in a new mode assignment. Mode variables adhere to the same scope rules as regular variables. That is, a mode variable declared within a procedure is local to this procedure allowing different procedures to use the same name for different purposes.

Definition of user data types leads to the definition of operators. As specified earlier in chapter 3, an operator consists of an integer, its priority, along with

a set of procedures with their associated data type specifications. A procedure is associated with an operator using the operator assignment statement:

OP opname(type1) := expression for monadic use, or

OP opname(type1,type2) := expression for dyadic use.

"opname" specifies the operator with which the procedure is to be associated; "type1" and "type2" are the type matching specifications associated with the procedure, and "expression" specifies the procedure itself. "expression" must evaluate to a single-parameter procedure for monadic use, or to a two-parameter procedure for dyadic use. "type1" and "type2" are declarers with possible length or dimension specifications but not base specifications. If a dimension specification is supplied, the actual bounds may be omitted to specify only the number of dimensions as in (,) INT. The length and dimension specifications may be procedures to allow for dynamic alteration of their specifications. If a length procedure yields the empty group upon evaluation, then the action is the same as if the specification were omitted. The action for dimension specifications will be described later.

As many procedures as desired may be specified for the operator by including as many OP statements as desired. If two OP statements with the same data types are specified, the second one overrides the first. One operator may be used with both monadic and dyadic specifications.

When an operator is used monadically the associated list of one-parameter procedures is searched for one with a "type1" specification matching the operand type. The data type, dimensioning, and if specified in "type1", the precision must match. If a match is found in the list, then the procedure is evaluated with the operand as its argument. If a dimension specification is omitted in the data type specification, then the operand must be a scalar to have a match. If a procedural dimension specification yielding an empty group as its value is supplied then the dimensioning information is ignored in type matching. (an operand which matches in type and precision is accepted regardless of its dimensioning). Otherwise the number of dimensions must match; and, if supplied in the dimension specification, the bounds of the dimensions must match.

If no match is found, then an error is generated for a scalar operand, while an aggregate operand is broken down into its constituent components at the first level. Each of these components is then tested against the operator definitions: this process being repeated as necessary as in:

```
OP TRUNC(INT):=("A" A);
OP TRUNC(REAL):=("A" INT X:=A);
INT X; REAL Y;  (3,2) INT Z;
```

TRUNC X selects the first procedure.

TRUNC Y selects the second procedure.

TRUNC Z since no match is found Z is broken down into

(TRUNC Z(1),TRUNC Z(2), TRUNC Z(3)) .

since this also fails we try:

```
((TRUNC Z(1,1),TRUNC Z(1,2)),
 (TRUNC Z(2,1),TRUNC Z(2,2)),
 (TRUNC Z(3,1),TRUNC Z(3,2))).
```

Each of these succeeds, using the first TRUNC procedure.

Thus the result has the same logical structuring as the operand (the operator is applied element by element). This selection technique allows aggregates to be used in a

natural way, element by element with operators defined only for scalars. Thus, only a few procedures need be specified to cover a large number of possibilities.

In the case of a dyadic operator, the selection is similar: a search is made for a pair of data types matching the operands. If found, the associated procedure is evaluated with the operands as arguments.

If no match is found, and both operands are scalars, then an error is generated. If one operand is an aggregate and the other a scalar, then the aggregate is distributed as described for monadic operators. This allows for scalar-aggregate operations to be performed in a natural manner by operating on the scalar in an element by element manner with the aggregate. Thus, for example, a constant may be added to each element of an array. The logical structuring of the result matches that of the aggregate operand.

If both operands are aggregates, then the number of descendant components at the first level must agree. Both

operands are distributed in the same manner and corresponding components are tested against the operator definitions. For example:

```
OP SUM(INT,INT):=("A,B" A+B);
(2,2) INT(A,B);
A SUM B
```

Since neither A nor B match,(the dimensions do not match), a test is made of:

```
(A(1) SUM B(1),A(2) SUM B(2))
```

Since there is still no match, we try:

```
((A(1,1) SUM B(1,1),A(1,2) SUM B(1,2)),
(A(2,1) SUM B(2,1),A(2,2) SUM B(2,2)))
```

This succeeds yielding a 2 by 2 result matrix. Note that:

```
(2,3) INT A  SUM  (3,2) INT B
```

would be illegal.

An alternate method for selecting operator procedures is provided to handle more complex selections. Instead of the type specifications in an operator definition, a single procedure constant with two (for monadic use) or three (for dyadic use) parameters may be provided as in:

```
OP MOVE("I,A,B"(I>0 ? @A¬=@B ? FALSE)):=("A,B"A:=B)
```

In operator selection, the descriptor procedure is invoked with the operands of the operator as its second and third arguments. (the use of the first parameter will be described below). The descriptor procedure must return a boolean string as its result. If all the values in the boolean result are "TRUE", then the associated procedure will be used as the operator.

If both standard and procedure descriptors for the same operator exist then they are tested as follows: the procedure descriptors are called first with the first parameter being passed the integer value zero. If the result "TRUE" is returned, then the associated procedure is used as the operator. If the value "FALSE" is returned, then the standard descriptors are tested. If no match is found, then before distribution of aggregates is attempted, the procedure descriptors are attempted, this time with a first argument value of one. This procedure is repeated, with the first parameter of the procedure descriptors being passed the number of times distribution of the operands has been attempted. Thus, the above MOVE example would assign its second operand to the first, and

would be selected if the standard descriptors did not match before distribution is attempted (I > 0), and if both operands have different addresses.

As another example, consider:

```
OP +("I,A,B" (I=0?A=0?FALSE)):=("A,B" B+0.5);
```

This procedure will be tested before the standard built in descriptors are tested, and will be selected whenever the first operand has the value zero. Thus whenever the first operand of the + operator is zero, then the result is the value of the second operand plus 0.5 . Thus we have the ability to specify fairly complex selection criteria for operators which gives great flexibility both in creating new operators and modifying particular procedures for built in operators.

We have just touched on the use of priorities for determining the order of evaluation of dyadic operators in an expression. As mentioned previously each operator has an associated priority which is an integer. In an expression the operators with the highest priorities will be evaluated first. The built in operators all will have implementation-defined priorities. (for example,

multiplication usually has a higher priority than addition). We may assign or alter priorities of operators using the priority assignment which has the form:

PRTY opname:=expression

"opname" is the name of the operator or a list of operator names in parentheses; "expression" is evaluated to yield a scalar integer giving the new priority of the operator. Thus the same expression when evaluated twice may be evaluated in different orders by changing the priorities of some of its operators. If an operator is defined but no priority is assigned to it, then a default priority of the value zero is assigned to it. Certain special "operators", the "," for group temporaries and the DO for repetitive expressions also have priorities: they are given lower priorities than the standard operators so that an expression such as:

A+B,C*D

will evaluate as ((A+B),(C*D)) and not (A+(B,C)*D). DO will have a lower priority than "," to cause

DO A,B;

to be interpreted as DO (A,B); and not as (DO A),B. The priorities of DO and "," may not be changed although the

Priorities of all other operators may be changed relative to them.

CHAPTER 8 CONCLUSIONS

In assessing a new computer language two aspects must be considered: does it provide the desired facilities and is it practical to implement?

In answer to the first it is the author's belief that there are many new features and combinations of features which are not present in any other known language. The ability to dynamically assign and change data types allows us to perform such functions as determining the structure of data at execution time. Thus changes to these data structures may be made independently of the procedures using them, and general procedures may be written to handle many record formats. The generalized groups and structures allow us to simply express operations upon collections of data, and to pick out subsets of these aggregates. The use of aggregates as subscript lists and argument lists is a simple but very powerful facility allowing for easy specification and alteration of lists. Dimensions, lengths, and bases may be specified as dynamic

quantities allowing attributes of variables to be dynamically specified and altered. The ability to create new data types and operators is a rare facility which is most useful for simplifying the writing of programs. Certain of the standard supplied operators are particularly useful: the execute operator, "_" and the FORMAT operator both perform necessary and complicated functions in a simple manner. No other language known to the author provides all of these features in a simple and concise syntax as does this language.

These elegant features would lose their usefulness if they are impractical to implement. To test the practicability of some of the features a partial implementation has been performed. This implementation has indicated that most of the features, in particular the use and subscripting of aggregates, will run with acceptable performance. Certainly the heavy use of such features as operators with complex selection criteria will be expensive ,but the power of the language will allow most applications to be performed with a minimum of such constructs. The intended use of this language is for an

online environment in which the inefficiencies of an interpreter are small in comparison to the slow speed of data display and entry by the user's terminal. The APL system has demonstrated that in an online environment, an interpreter can meet with great success.

In an implementation of this language certain alterations may be desired. For terminals without some of the special characters, reserved words may be substituted such as LT for < or EXEC for _. As mentioned, for readability we might use IF,THEN,ELSE FI or CASE,IN,OUT,ESAC for (,?,?,). We might wish to extend the EXIT and EXITDO constructs to return out of more than one block or we may wish to add a special clause after a DO loop which will be executed if an EXITDO is encountered but not if the loop is terminated normally. We would need to add functions for editing and correction of source statements, error handling, and the addition and retrieval of procedures from external libraries. These features were not discussed in this thesis since they are to some extent implementation-dependent.

Another facility which has been completely ignored in this thesis is the ability to perform input and output to external files. Due to the high variability in input/output requirements for each installation and application, these facilities have not been included in the language. It is intended that the input/output requirements would be met by assembly language procedures called by procedures of the language. This provides for the greatest flexibility possible among installations, and in addition, reduces the memory requirements for those users not requiring a high degree of complexity in input and output.

It is the author's belief that this language presents new ideas which are very useful, and which are possible to implement in an efficient manner.

REFERENCES

The following are the major references for this thesis. A selected bibliography of other relevant works follows.

(1) CODASYL Systems Committee. Feature Analysis of Generalized Data Base Management Systems. May 1971 (available from the Association for Computing Machinery)

(2) Pakins. APL/360. Science Research Associates, inc.

(3) Van Wijngaarden(editor), Mailloux, Peck, Koster. Report on the Algorithmic Language ALGOL 68, Numerische Mathematik 14,2 (1969) 79-218.

(4) IBM. OS Full American National Standard Cobol. Form C28-6396.

(5) IBM. System 360 PL/I Reference Manual. Form C28-8201.

(6) CODASYL Data Base Task Group Report. April 1971. (available from the Association for Computing Machinery).

(7) Griswold, Poage, Polonsky. The SNOBOL4 Programming Language. Prentice-Hall, Englewood Cliffs, N.J. (1968).

(8) Wulf, Russell, Habermann. BLISS: A Language for Systems Programming. Communications of the ACM 14,12 (DEC 1971) 780-790.

BIBLIOGRAPHY

Alsberg. Extensible Data Features in the Operating System Language OSL/2. SIGOPS Operating Systems Review. 6,1 (June 1972) 31-34

Balzer. An Overview of the ISPL Computer System Design. CACM 16,2 (Feb 1973) 117-122

Branquart, Lewis, Sintzoff, Wodon. The Composition of Semantics in ALGOL 68. CACM 14,11 (Nov 1971) 697-708

CODASYL Systems Committee. Introduction to "Feature Analysis of Generalized Data Base Management Systems". CACM 14,5 (May 1971) 308-318

Conway, Maxwell, Morgan. On the Implementation of Security Measures in Information Systems. CACM 15,4 (Apr 1972) 211-220

Dijkstra. GOTO Statement Considered Harmful. CACM

11,3 (May 1968) 341-346

Dijkstra. Notes on Structured Programming. (Aug 1969)

Earley, Caizergues. A Method for Incrementally Compiling Languages with Nested Statement Structure. CACM 15,12 (Dec 1972) 1040-1044

Earley. Toward an Understanding of Data Structures. CACM 14,10 (Oct 1971) 617-627

Gimpel. Blocks - A New Datatype for SNOBOL4. CACM 15,6 (June 1972) 438-447

IBM. Information Management System/360. Application Description Manual. Form H20-0524.

Iverson. A Programming Language. Wiley, N.Y. 1962.

Knuth. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass.

Leavenworth(Editor). SIGPLAN Notices 7,11 (Nov 1972). Special Issue on Control Structures in Programming Languages.

Morgan. An Interrupt Based Organization for Management Information Systems. CACM 13,12 (Dec 1970) 734-738

Morris. Protection in Programming Languages. CACM 16,1 (Jan 1973) 15-21

Rosen (Editor). Programming Systems and Languages. McGraw-Hill, N.Y. 1967.

Rosen. Programming Systems and Languages 1965-1975. CACM 15,7 (July 1972) 591-600

Sammet. Programming Languages: History and Future. CACM 15,7 (July 1972) 601-610

Shoshani, Bernstein. Synchronization in a Parallel Accessed Data Base. CACM 12,11 (Nov 1969) 604-607