# VERS Manual

Version 4

Jay Earley

Paul Caizergues

Computer Science Department

University of California, Berkeley

October 1971

## Table of Contents

## Introduction

This document contains all the information necessary for the understanding and use of the programming language VERS (for versatile) as it exists on the CAL 6400 Time Sharing System. VERS is an interactive language for manipulating complex data structures. It is intended to be useful because of the generality and semantic clarity of its data structures.

The system is completed and ready for use. For further information on the system or consulting on its use, contact Jay Earley, 507 Evans, (64)2-1879.

The current system is rather inefficient in both time and space. This is because (1) VERS is implemented by an interpreter, and (2) it makes no effort to pick a good internal representation for the user data structures. We intend to remedy these problems by adding to the VERS system a compiler which uses an "implementation facility." See [1] for the ideas behind this.

Reference [2] describes part of the current implementation of VERS.

[1] Earley, Jay. "Toward an Understanding of Data Structures," Comm. ACM, October 1971.

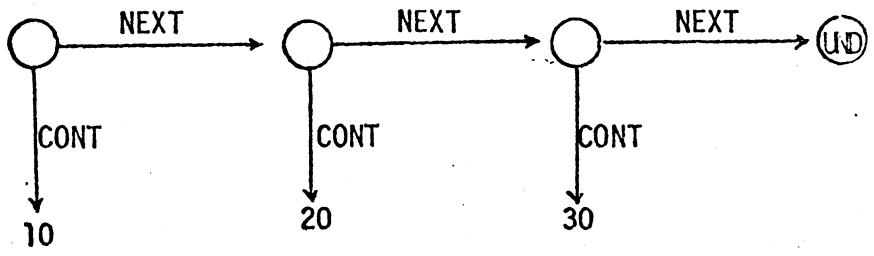
[2] Earley, Jay and Caizergues, Paul. "A Method for Incrementally Compiling Languages with Nested Statement Structure," Computer Science Department, University of California, Berkeley, 1971.

# PART I

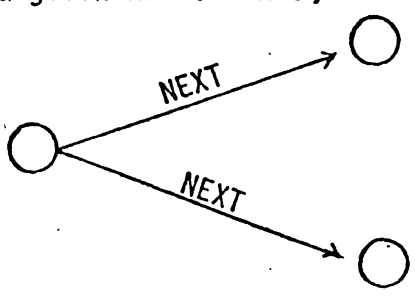
## Introduction to V-graphs

The most important feature of VERS is its data structures. These are designed with the idea of distinguishing very carefully between semantics and implementation. The semantics of a data structure is the essential meaning of how its data is stored, how it is accessed, and how the structure may be changed. It is often a model of a situation or structure in the real world. The implementation is how this semantics is realized in a machine. Another way of looking at it is that semantics is a description in which all the extraneous detail has been omitted. What is extraneous depends, of course, on the purpose of the description, but this is as it should be.

VERS deals with the semantics of data structures. It represents them as directed graphs with names on the edges (called V-graphs). Each node of the graph represents a part of the data structure. The arrows between nodes represent access paths in the structure, and arrows from nodes to "atoms" represent the fact that the atom is stored at the node. Thus, a linked list of the atoms 10, 20, and 30 might be represented as follows:




The graphs have the further property that there is at most one link from any node with a given name. Thus,



is illegal, so that when one is at a certain node, any given link name selects a unique access path from that node; consequently we call these "selectors". In the above example NEXT and CONT are not reserved words in any sense; we could have just as easily used any two distinct strings in their place. UND, however, is a special reserved object which can be used for such things as the end of a list.

The following is a more precise definition of V-graphs: There are three kinds of objects -- nodes, links, and atoms. There are an infinite number of atoms and an arbitrarily large number of nodes and links. Atoms are objects which have no parts that can be examined or changed as such. Only tests and operations on atoms as wholes are allowed (such as "=" or "+"). Nodes have no intrinsic meaning. All their meaning is derived from the structural and access relationships represented by the links. Each node may have an arbitrary number of links from it and no two links from the same node may have the same name.

We can represent an array of numbers (7,3,5,6,18) which are accessed by subscript only as follows:

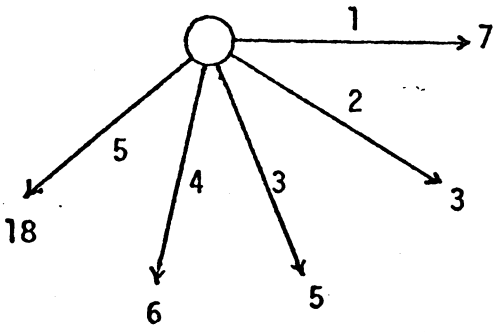



Figure 1

The crucial feature about V-graphs is that they represent the set of data and the way it will be accessed, not necessarily the way it is implemented.

Thus, if our list elements 10, 20, 30 can be accessed by ordinal number as well as by the "next" operation, then we might represent them as follows:

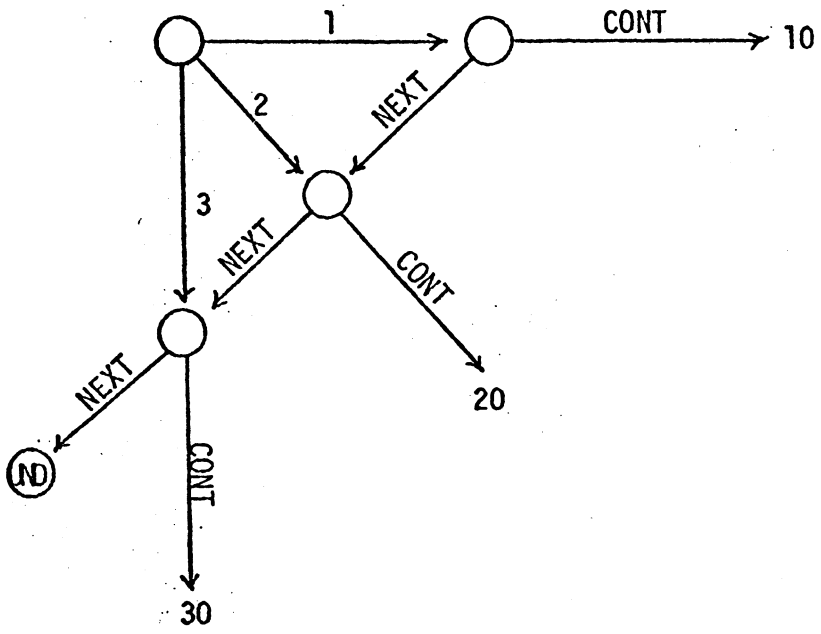
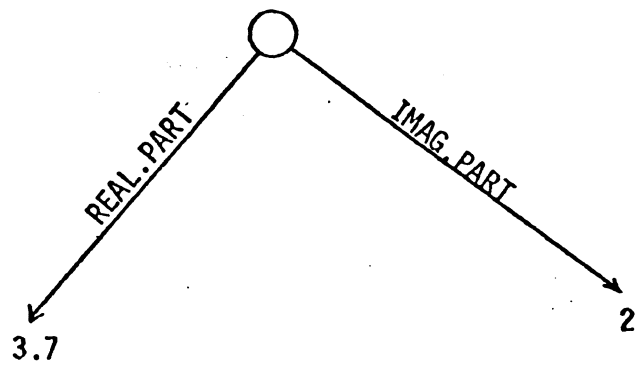



Figure 2

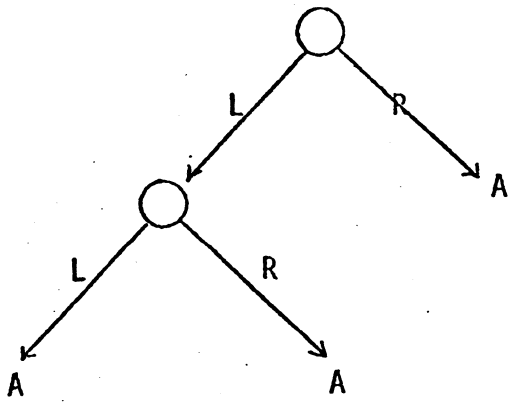
In addition, if there is a "prior" operation available, we would draw in the inverses of the NEXT links and name them PRIOR. It should be clear at this point that we look at a data structure as more than just a set of atoms in some relationship to each other. The way in which the atoms are accessed is an intrinsic part of the data structure. Thus, for instance, an ordered set can be many different data structures depending on the way its members are accessed.

The following graph represents a complex number 3.7 + 2i as a pair of reals:



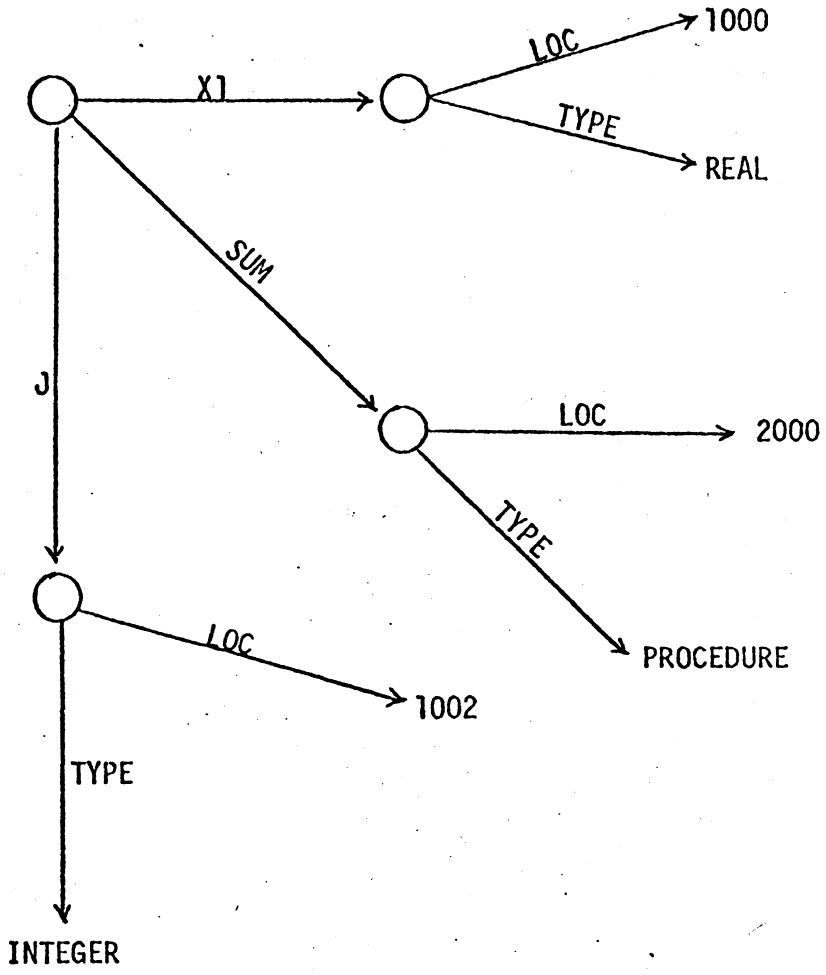
There are exactly five objects in this graph, and each one represents an essential part of the data structure. The node represents the pair itself, the two links represent the "left-hand values" or references to the members of the pair, and the two atoms represent the values of the members of the pair. This correspondence between objects in the V-graph and intuitive notions about the data structure is what gives V-graphs the capability of describing semantics.

As one might expect, we can represent a binary tree (as in LISP) as follows:




Here we have chosen L and R instead of CAR and CDR; the A's are atoms. In this example as in the linked list, links in the graph will most likely correspond to pointers in an implementation, but this is definitely not the case in general. The number links in our array example (Figure 1) certainly do not represent pointers, nor do the NEXT links in the list-array example (Figure 2), if we store it contiguously in core. In fact, the fundamental idea behind V-graphs is that the links may represent various things at the machine level depending on how one chooses to implement the graph. A link may represent a pointer, an indexing operation, contiguity in memory, or even an entire search algorithm. This last case is illustrated by the following example.

This graph represents a symbol table in which the identifiers X1, SUM, and J have been entered. Each symbol has a location and a type associated with it.

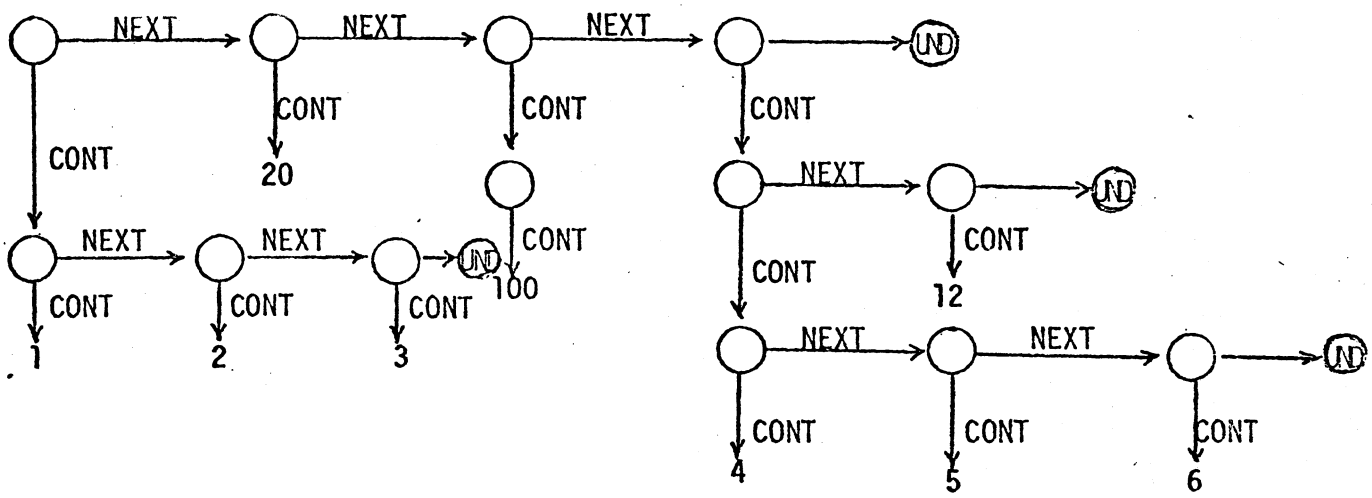



Notice that the X1, SUM, and J links may be implemented by some kind of search algorithm while the LOC and TYPE links will probably just be fields within a block of storage. Nevertheless, all these links really do represent access paths to the data, and that is why they are present in the graph.

We often work with data structures which are naturally hierarchical and so we would like to reflect this in V-graphs. This can be done without

adding any mechanism by simply using a certain selector (such as CONT) to represent the idea of sub structure. Thus a list of lists might be represented as follows:

((1,2,3),20,(100),((4,5,6),12))




For reasons which will become clearer later, each node and each atom has a type chosen from a finite set of types. There are some standard atom types: integer, Boolean, string, type, etc.. In general, a link in a V-graph can link to any object -- node, atom, or link. The reason for including links in this category is that one would like to be able to manipulate references (links) just like any other object, and this necessitates the ability to link to them.

The following primitives exist for manipulating V-graphs. OBJECT stands for ATOM, NODE, or LINK.

FOLLOW (SELECTOR A, NODE N) = LINK

This produces as a result the link from node N named A. It is an error if the link does not exist.

STORE(LINK L, OBJECT Q) = Q

This causes link L to link to Q.

CONTENTS(LINK L) = OBJECT

This produces as a result the object pointed to by link L.

ATTACH(SELECTOR A, NODE N) = LINK

This creates a link from N named A and produces it as a result. It is an error if a similarly named link already exists.

DETACH(SELECTOR A, NODE N)

This deletes the link named A from N if it exists. It is an error if it doesn't.

TEST(SELECTOR A, NODE N) = BOOLEAN

This returns TRUE if a link named A exits from N and FALSE otherwise.

CREATE(TYPE T) = NODE

This creates a node of type T and produces it as a result.

SEL(NODE N) = sequence of SELECTOR's

This produces as a result the sequence of selectors of the currently existing links from node N.

TYPE(OBJECT Q) = TYPE

This produces as a result the type of Q.

KIND(OBJECT Q) = TYPE

This produces as a result ATOM, NODE, or LINK, depending on which kind of object Q is.

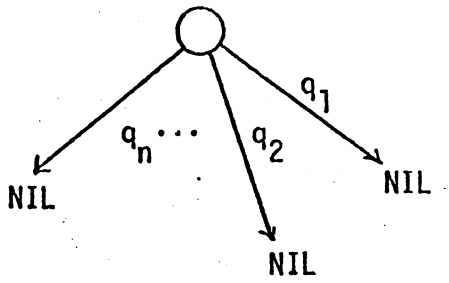
EQ(OBJECT P, OBJECT Q) = BOOLEAN

Unless P and Q are nodes, this returns TRUE if P and Q are the same object and FALSE otherwise. If they are nodes, it may be more

complicated, as is explained below.

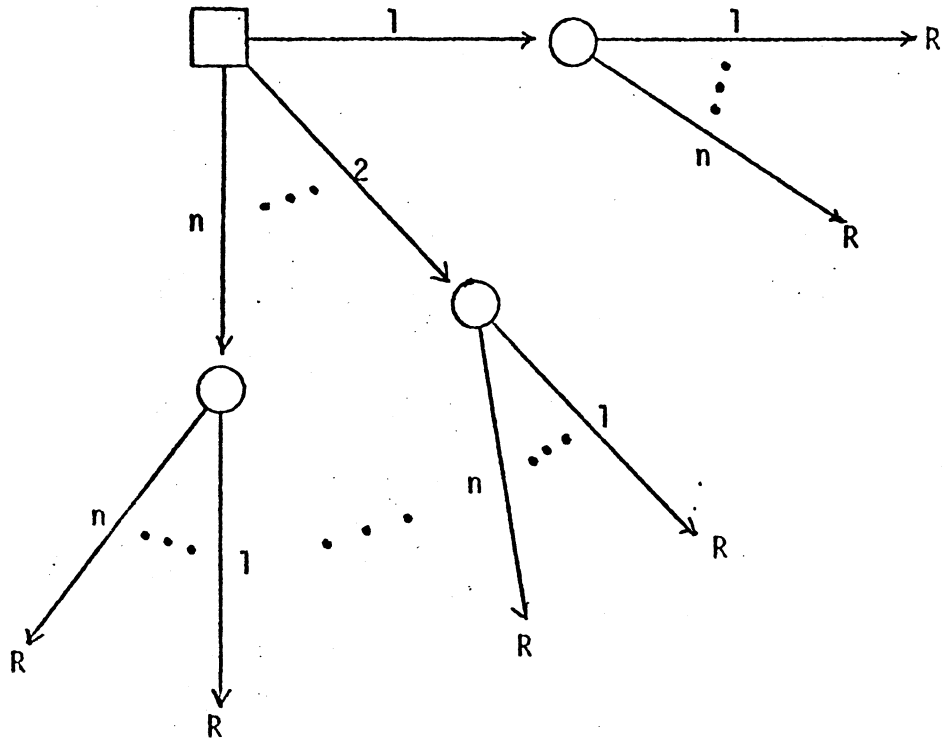
We have not yet specified what may be used as a SELECTOR, though we have used only atoms in our examples. We now present examples where nodes are used as selectors:

Consider how to represent a finite set as a V-graph. A finite set of atoms $q_1,\ldots,q_n$ can be represented as follows:

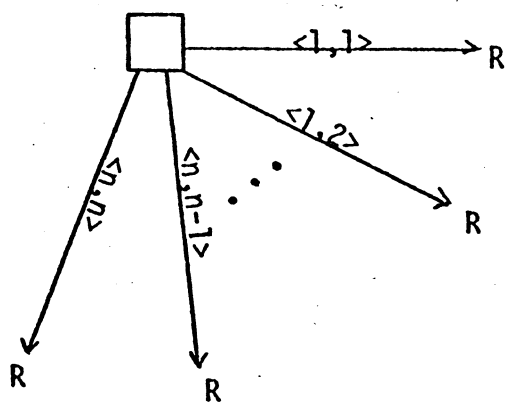



Adding an element to the set is done with TEST and ATTACH, deleting an element with DETACH, testing for set membership with TEST, and generating the elements of the set with SEL. However, if we want to represent a set of data structures, such as a set of pairs, we must allow nodes to be link names and then use the header node of each pair in place of the $q_i$ above.

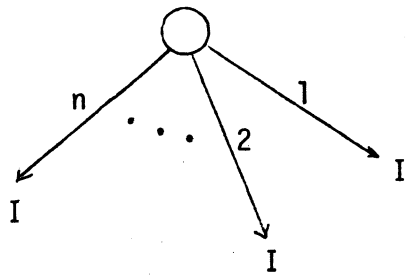
Consider how to represent a two-dimensional array as a V-graph. One might think of the following for an $n \times n$ array:




But this has some extra structure which is not really there, i.e. are we selecting rows or columns first? Do those circular nodes really represent meaningful entities? Of course, they might, if we want our semantics to allow for the manipulation of rows or columns as units. But suppose we don't. A better representation would be obtained by using ordered pairs as selectors:



So in general, in order to be able to represent n-dimensional arrays, we need to be able to use ordered n-tuples as selectors. But these are data structures themselves:




so we can do this by using the header node of the n-tuple as the link name. However, this means that the EQUALS primitive must act different on such nodes. It must have the property that two different pair nodes which both represent the pair <1,2>, for instance, must be equal. This means that if one does an ATTACH with one pair node and a FOLLOW with the other he will get the right link. For this purpose some node types are designated "separate" so that two such nodes are equal only if their components are equal.

B. Introduction to VERS

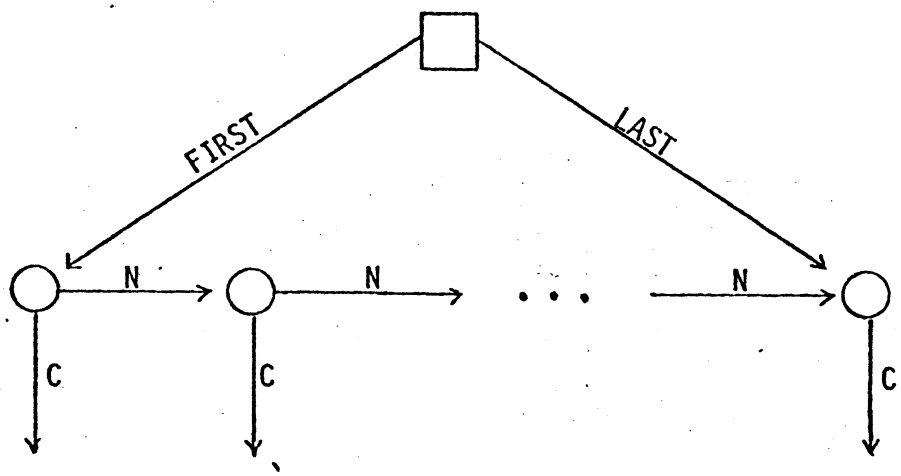
In VERS there are basically two categories of objects -- a fixed number of "static" objects whose names can be written in a program, and an arbitrary number of "dynamic" objects which are created by the program at run-time and may be referenced only through the static objects. Thus there are static and dynamic varieties of nodes, links, and atoms. Static nodes are headers for data structures whose identity cannot be changed by the program. (The symbol table header is an example of this.) Dynamic nodes are created by the CREATE primitive and are parts of data structures which can be altered (such as stack element nodes). They may be referenced only through links. Static links are links which do not come from nodes, but have an existence of their own. These are analogous to variables in other languages. The

links emanating from nodes are dynamic and represent access paths in the data structure. Similarly, a static atom is a constant, and a dynamic atom is the value of a variable or a value stored in a data structure.

Now we illustrate how node types are declared in VERS. The "list element" node type is declared as follows:

```
TYPE LISTEL IS
     N TO LISTEL
     C TO ANY
```

This specifies that there are two allowable selectors from a LISTEL node, N and C. N names links which may point to other LISTEL nodes, and C names links which may point to anything (ANY). Here we have underlined the keywords which are built into VERS. We also need to declare a "sequence" node type. Sequences are used frequently in VERS for ordered collections of things. A sequence has the following general form:




where the round nodes are LISTEL's. The sequence node type is declared as follows:

```
TYPE SEQ IS
     FIRST TO LISTEL
     LAST TO LISTEL
```

We will now write a VERS routine to take as input a sequence of integers and return the sum as its result.

```
ROUTINE SUM(S TO SEQ) TO INTEGER IS
LINK P TO LISTEL
RESULT ← 0
P ← FIRST OF S
IF P = UND THEN RETURN
LOOP: RESULT ← RESULT + C OF P
IF P = LAST OF S THEN RETURN
P ← N OF P
GO TO LOOP
END
```

S is the formal parameter. It is a static link which may point to SEQ's. In addition, there is one declared static link, P, which can point to LISTEL's, and one implicit static link RESULT, which can point to INTEGER's since that is the type of the routine. All three of these are local to SUM. "←" stands for STORE, and "OF" stands for FOLLOW. We have set it up so that the CONTENTS primitive does not have to be written explicitly in most cases. When a link is generated by FOLLOW or ATTACH or by using a static link, its contents is automatically taken unless it is suppressed. It may be suppressed explicitly by using the special operator @ or implicitly in certain contexts (see section II.F).

There is a FOR statement in VERS which works on sequences. It stores into the controlled variable the successive values of the C links of the sequence. Using this, we can rewrite SUM as follows:

```
ROUTINE SUM(S TO SEQ) TO INTEGER IS
LINK I TO INTEGER
RESULT ← 0
FOR I ← IN S DO RESULT ← RESULT + I
END
```

In these two examples, we have used FOR and IF statements which are contained entirely on a single line. We also allow versions of these which continue over many lines and can be nested (see II. G).

Our next example shows how the symbol table node is declared and used.

```
TYPE TABLE IS
    [STRING] TO INFO
TYPE INFO IS
    LOC TO INTEGER
    TYP TO STRING
```

In all the previous examples the set of possible selectors from a node was finite. [STRING] specifies that any number of links can be attached to TABLE, but they must all be selected by objects of type STRING. Now if we want to enter a symbol S in a table T, it is done as follows:

```
ATTACH(S,T) ← CREATE(INFO)
```

If we want to set the LOC of S to an address H, we code

```
LOC OF S OF T ← H
```

To look up the TYP of S:

```
TYP OF S OF T
```

Finally to print all the symbols in the table:

```
FOR S ← IN SEL(T) DO PRINT(S)
```

We continue now with examples illustrating other features of VERS: There is a second kind of FOR statement which can be used to get the effect of numerical FOR statements as in Algol, and for other purposes. The following program prints out the largest value in a vector of 100 positive integers. First we declare a node type to represent the vector:

```
TYPE VECTOR100 IS
     1 THRU 100 TO INTEGER
```

Then the routine is as follows:

```
ROUTINE LARGEST(V TO VECTOR100) TO INTEGER IS
LINK I TO INTEGER
RESULT ← 0
FOR I ← 1, I+1 WHILE I ≤ 100 DO
IF V[I]>RESULT THEN RESULT ← I
ENDFOR
END
```

Here "V[I]" is just another way of writing "I OF V." Notice that we have used a FOR statement format which extends over several lines. This is signified by the fact that there is nothing following the DO in the FOR header. The body of the FOR statement is terminated by ENDFOR, which must be on a separate line. Either format for FOR statements (single line or multi-line) may be used with either kind of FOR statement. There is a similar multi-line format for IF statements.

We can also use this second kind of FOR statement to move through a sequence, generating the elements instead of the values. For instance, if we know that the sequence ends with UND, we write

```
LINK L TO LISTEL
LINK S TO SEQ
FOR L ← FIRST OF S, N OF L WHILE L# UND DO body
```

Here "#" stands for "not equal". This assigns to L the successive LISTEL's of the sequence, and terminates when L becomes UND.

Primitive forms of input and output are provided in the form of routines PRINT and READ, PRINT takes a string as input and prints it at the teletype. READ returns control to the teletype, collects a line, and

returns it as a string. The following example program reads lines from a teletype continuously until it finds one consisting of the word "LOOK". It then prints "FOUND" and terminates.

```
ROUTINE FIND( ) IS
LOOP: IF READ( ) = "LOOK" THEN PRINT("FOUND"); RETURN
GOTO LOOP
END
```

Notice that this IF statement has two statements in its scope. In general, more than one statement may be placed on a line if they are separated by ";"'s. However, only one IF or FOR statement may be placed on a line, and its scope is the entire remainder of the line. See II.G for the exact syntax.

The primitive NAME is provided for use with PRINT. It converts any object into a string so that it may be printed. Examples:

| code | printed |
|---|---|
| PRINT(NAME(3)) | 3 |
| A ← TRUE; PRINT(NAME(A)) | TRUE |
| A ← 1; PRINT(NAME(@A)) | A |
| A ← CREATE(SEQ); PRINT(NAME(A)) | NODE |
| PRINT("A" & "BC") | ABC |

A is a link. If it is used without the @, its contents are input to NAME, otherwise the link itself is. This explains the difference between lines 2 and 3 above. (This is a property of "@", not of NAME.) In line 4, the node which is the contents of A is dynamic and therefore has no name, so VERS prints "NODE". In line 5, "&" is the string concatenation operator.

These primitives will be useful in writing detailed I/O routines for completed programs. We also provide two library routines, P1 and P2 which should be especially useful for debugging. Each of these prints out any VERS object (including the entire graph structure that can be accessed from a node or link) in a given format. See section II.K for details.

Scopes of names are controlled by "data blocks" as well as routines in VERS. A data block is a collection of global declarations. Associated with each data block is a list of the routines which may use declarations from this block. Routines are not nested within each other, so that the only names global to a routine are those found in data blocks which contain it. In the example below we have one data block and a routine which it contains. In a larger program, the data block would contain other routines as well. In addition a routine may be contained in several data blocks.

We now present a more complete and practical example of a VERS program. The program deals with a BNF grammar, and a knowledge of BNF is assumed here. We will represent each symbol in the grammar (terminal and non-terminal) as a STRING. Each non-terminal will map onto a set of its alternatives using a GRAM node, as follows:

```
TYPE GRAM IS
     [STRING] TO NON.TERM ← CREATE(NON.TERM)
TYPE NON.TERM IS
     [SEQ] TO NULL
```

Here we have a node of type GRAM representing the grammar. It will have one link for each non-terminal N, named by the string which is the name of N and linking to a node of type NON.TERM. NON.TERM nodes

represent sets, in this case sets of SEQ's, because each alternative is a sequence of strings. In general, the best way to represent a set in VERS is as a node with links which point to UND (undefined), where the selectors are the elements of the set. Thus a general set would be

```
TYPE SET IS
     [ANY] TO NULL
```

We use NULL here because UND is of type NULL.

The "← CREATE(NON.TERM)" indicates that whenever a link of this kind is attached to a GRAM node that it is to be initialized to a newly created NON.TERM node. A static link may also be given an initial value in the same way. Any link which is not initialized in this way will be set to UND when it is created.

Notice that the representation we have chosen for the grammar make explicit exactly those access paths which we need and leaves out details of implementation which are not essential:

(1) There is a way to get from a non-terminal to its set of alternatives.

(2) It is clear that the alternatives are a set (unordered) and the symbols within an alternative are a sequence (ordered).

The routine we are going to write will take as input a symbol C and output the set of all terminals which can begin some derivation from C. The output will simply be C, if it is terminal; and if it is non-terminal, it will call itself recursively on the first symbol of each alternative of C.

The complete program is as follows:

```
DATA IN BEG
TYPE NON.TERM IS
     [SEQ] TO NULL
```

```
TYPE GRAM IS
     [STRING] TO NON.TERM ← CREATE(NON.TERM)
GRAM G
TYPE TERM.SET IS
     [STRING] TO NULL
END
ROUTINE BEG(CH TO STRING) TO TERM.SET IS
LINK NT TO NON.TERM
LINK ALT TO SEQ
LINK T TO STRING
RESULT ← CREATE(TERM.SET)
NT ← (CH OF G)\TERM
FOR ALT ← IN SEL(NT) DO
FOR T ← IN SEL(BEG(C OF FIRST OF ALT)) DO
ATTACH(T,RESULT)
ENDFOR
ENDFOR
RETURN
TERM: ATTACH(CH,RESULT)
END
```

In line 6, "GRAM G" declares G to be a static node of type GRAM. In line 15, we introduce the use of "error returns" in VERS. The idea is as follows: Any call on a primitive routine may be followed by "\LABEL". This means that if the routine gets an error, instead of halting execution and printing an error message, we execute a GOTO LABEL. In this particular example, the primitive is FOLLOW, and it will get an error if the selector doesn't exist. That is, if CH is a terminal, the selector will not exist and we will branch to label TERM. We could have equivalently written that line

```
IF CH FROM G THEN NT ← CH OF G ELSE GOTO TERM
```

Here "FROM" calls the primitive "TEST".

This error return feature is extended to programmer defined routines as follows. There is a special label "ERROR" which a routine may branch to. This causes a return from the routine and a branch to its error label, if its call has written using "\LABEL", and an error message if it was not.

Interactive control of VERS programs is partly achieved by using BREAK, CONT, and SET. BREAK is a special label which causes control to return to the teletype. Breaks may also be caused implicitly by setting traps on statements or links. CONT is used to continue from a break or a trap. For example, suppose we have entered the following VERS program:

```
ROUTINE R( ) IS
LINK B
PRINT("1")
GOTO BREAK
L: PRINT("2")
B ← 3
PRINT(NAME(B))
END
```

A session with this program might go as follows:

```
+ R( )
1
BREAK AT R+3
+SET L
+CONT
BREAK AT L
+SET B
+CONT
2
BREAK AT L+1
+CONT
3
+
```

The "+" is a prompt character printed out by VERS when it is waiting for input from the user, so every line beginning with "+" was typed by the user and every other line was printed by VERS. SET L puts a trap on the line labeled L, so that every time that line is executed a break will occur. SET B puts a trap on link B, so that every time it is stored into a trap occurs.

During a break, the user may execute any statement in VERS as an immediate statement by simply leaving the first column blank and typing the statement. It is then executed as if it were in the current routine. This can be used to look at data or change values of data, or to call other routines, or to resume execution by executing a GOTO to a place in the current routine.

Detection of errors during execution also produces a break (along with a message). The only difference here is that CONT may not be used to continue after an error (because its not always clear what that would mean). So the user can do a GOTO to continue at some point, or he may do an INIT, which has the effect of cancelling all execution and putting the system back to its initial state.

For example, suppose we have the following highly error ridden VERS routine:

```
ROUTINE R( ) IS
LINK A TO INTEGER
A ← TRUE
L: A ← 3 OF 4
END
```

A typical session might be as follows:

```
+ R( )
ERROR AT R+2
ATTEMPT TO STORE VALUE INTO LINK OF WRONG TYPE
```

```
+ P2(A)
UND
+ A ← 3
+ P2(A)
3
+ GOTO L
ERROR AT L
WRONG TYPE FOR SECOND PARAMETER OF
FOLLOW
+INIT
+
```

Here P2 is a library routine which prints out its argument. Notice that every immediate statement must be preceded by a blank, and that every command (INIT, CONT, SET) must start in the first column.

During a break the user may also edit any part of his program by using the command EDIT. This calls the normal TSS editor to work on his source program. (A knowledge of it is assumed here.) He may then continue execution if he hasn't modified the line he was executing when the break occurred or certain other lines which could have disastrous effects (see II.J). VERS syntax checks every line entered by the editor in the program and refuses the invalid ones.

Here is an example illustrating some of these points.

```
ROUTINE R(S TO STRING) IS
TYPE THING IS
     C TO BOOLEAN
THING A
L: S OF A ← TRUE
P2(S OF A)
END

+ R("D")
ERROR AT L
ATTEMPT TO FOLLOW LINK WHICH DOESN'T EXIST
```

```
+ P2(S)
D
+EDIT
:M3;E
     D DO BOOLEAN
SYNTAX ERROR
     D DO BOOLEAN
REFUSED
:E
     D TO BOOLEAN
:F
+ GOTO L
TRUE
+
```

## PART II - VERS Reference Guide

The description of VERS consists of two parts. There is an abstract VERS machine which consists of basic objects and primitive operations on them and ways of combining these into programs. Second we can describe the syntax of VERS and show how any VERS program corresponds to a program for the VERS machine. This is also the way VERS is implemented; there is an interpreter for the VERS machine and an incremental compiler to translate programs for it. In the following description, we will intermix these two parts, giving both the syntax and the primitive operations and objects and the correspondence between the two for each different concept in the language.

### A. Syntax Conventions

While running under the VERS system, the current text file must always contain a legal VERS program, except when one is in "edit" mode. In addition, commands may be typed in at the teletype in order to edit the program, to execute statements immediately, and to perform other functions. Thus we present two grammars: PROGRAM describes a legal VERS program, and COMMAND describes the form of legal teletype lines.

Our variant of BNF does not use < > and replaces ::= with ~. In addition the following abbreviations are used:

| | |
|---|---|
| Z ~ A{X \| Y}B | means Z ~ AXB \| AYB and { } are used generally for grouping |
| $X^?$ | means X may appear optionally |
| $X^*$ | means X may appear 0 or more times |
| $X^+$ | means X may appear 1 or more times |
| X_LIST | means $X\{,X\}^*$ |
| X_SEQ | means $\{X\ \underline{CR}\}^+$ |
| X_IDENT | means an identifier which is declared of type X. |

CR is a terminal in the grammar and stands for carriage return or end of line. Identifiers (IDENT) are strings of letters, digits, and "." beginning with a letter. Integers (INT) are strings of digits. Strings (STRING) are strings of characters enclosed in double quotes ("), where two quotes stand for one, inside the string. All reserved words in the grammar are underlined. This is for readability only, it is not necessary in actually using VERS.

B. Comments and Continuation Lines

Comments may be placed on any line after a "!"; everything on the line after it is ignored. "?" is used for two purposes: (1) When not in column 1 it is a continuation character. This means that everything after it on the line is ignored and the next line of text is to be considered as an extension of the current. If more than one line in a row is continued, they all are put together logically into one line. (2) All sub-lines of a continued line (including the first) must also have a "?" in column 1 (see section I). An immediate statement may not be continued. Also, a token (reserved word, identifier, integer, or string constant) may not be continued from one line to the next.

The following examples illustrate:

```
? ROUTINE P(A TO INTEGER, B TO STRING,?
? C TO BOOLEAN) IS ! THIS IS A COMMENT
! THIS LINE IS IGNORED ENTIRELY
```

C. Naming Structure

```
PROGRAM ~ {DATA.BLOCK | ROUTINE}_SEQ
DATA.BLOCK ~ DATA{IN ROUTINE_IDENT_LIST}? CR
             DECL_SEQ
             END
```

```
ROUTINE ~ ROUTINE.HEAD{IN ROUTINE_IDENT_LIST}? CR
          DECL_SEQ?
          STATEMENT_SEQ
          END
```

A VERS program consists of routines and data blocks. VERS has a modified

one-level block structure. Any declaration (DECL) which occurs within a routine is local to that routine, and in fact the object will have new incarnations if the routine is called recursively. Any declaration which occurs in a data block is valid in all routines which appear in the ROUTINE_IDENT_LIST associated with the data block. If that list is omitted, then the declaration is global (valid in all routines). None of these data block declarations are local, however, in the sense that they do not participate in recursion, and they are not initialized on routine entry. A data block should appear in the program before any routines which are included in its scope.

A routine name is valid within all routines which appear in the ROUTINE_IDENT_LIST in the routine header and within itself. It is also valid in any routines in which it has a forward declaration (see page 21). If the list is omitted, then the routine name is global.

No name may have two declarations which make it valid in the same routine, even if one declaration is global and the other local. This means that one should be especially careful of names in global data blocks and global routine names.

Example:

```
DATA
LINK A
END

DATA IN R,S
LINK B
END

ROUTINE R( ) IN S IS
A,B,R,T may be used
END
```

```
ROUTINE S( ) IN S IS
A,B,R,S,T may be used
END
ROUTINE T( ) IS
ROUTINE S
A,S,T may be used
END
```

D. Basic Objects and Declarations

Declarations in VERS describe static objects and, in some cases, give them initial values. A static object is one which is named by an identifier in the program and is created either at the beginning of program execution (if it is global) or upon routine entry (if it is local). Dynamic objects are created during execution by primitive routines.

The following are descriptions of each of the basic objects which can exist during the execution of a VERS program and, where appropriate, the syntax of their declarations. The full description must also include the primitive routines which are applicable to each kind of object. These are in the next section.

(1) NODE. Nodes are the building blocks for VERS data structures. They have no intrinsic meaning and are given meaning only by the way in which they are linked to other nodes and atoms. Each node has a node type defined by the programmer (see 3.e.2). Static nodes are declared as follows:

```
DECL ~ NODE.TYPE_IDENT IDENT_LIST
```

(2) LINK. Links describe the access paths which exist in the data structure. They go from nodes to other nodes, to atoms, and even to other links. Each link has a link type which specifies which types of objects it may point to.

DECL ~ LINK{IDENT {TO TYPE.SET}?{←EXP}?}_LIST
TYPE.SET ~ TYPE_IDENT{OR TYPE_IDENT}*

This declares a static link and defines what it can link to (its link type) and its initial value. The expression is evaluated and stored in the link, on routine entry if it is local or at the beginning of program execution if it is global. If the link type is omitted it is taken to be ANY, and if the expression is omitted it is taken to be UND.

The use of a TYPE_SET of more than one element implicitly defines a compound type corresponding to the TYPE_SET, which is then used in its place.

Examples:
```
LINK A
LINK B TO STRING
LINK E TO INTEGER ← 0
LINK D TO STRING OR VAL
LINK F,G,H
LINK I TO STRING, J TO VAL
```

(3) ATOM. Atoms are objects which have a meaning in themselves and are indivisible with respect to the data structure operations. There are special operations for some types of atoms, however.

DECL ~ TYPE_IDENT IDENT {IS{-?INT|STRING}}?

This is a general declaration for any static atom. If the atom is defined elsewhere, such as with a routine or label, then it can not be given a value here. In these cases, the declaration is necessary if the object would otherwise be used before it was defined. Other than this, the name must be given a value. Note that this is not just an initial value, it is the object itself, and it cannot be stored into.

Examples:
```
ROUTINE A
INTEGER S IS 3
```

There are seven atom types, which we now describe.

(a) INTEGER. Integers are in the range $2^{15} - 1$ to $-2^{15} + 1$

(b) BOOLEAN. TRUE and FALSE.

(c) STRING. A sequence of characters chosen from those normally acceptable to TSS.

(d) LABEL. A reference to a statement to which control may be transferred.

(e) TYPE. A type may be a) a node type declared by the programmer, b) one of the eight atom types, c) LINK, or d) a compound type.

(e.1) Compound Type

DECL ~ TYPE TYPE_IDENT IS TYPE.SET

This declares the TYPE_IDENT to stand for any of the types in the set. These may also be compound types. A compound type may be used as an "allowed" type (such as a link type), but it may not be the type of an object. That is, a compound type can never be the result of a call TYPE.OF(A). In addition, NODE, ATOM, and ANY are compound types with the obvious meanings.

Example:
```
TYPE VAL IS INTEGER OR LISTEL
LINK A TO VAL
```

Warning: Do not declare a compound type which refers to itself recursively, such as

```
TYPE A IS B OR A
```

or

```
TYPE A IS B OR C
TYPE C IS D OR A OR F
```

This kind of construction may cause an infinite recursive loop in VERS, which will result in a STACK OVERFLOW message.

(e.2) Node Type

```
DECL ~ TYPE IDENT SEP? IS CR
       SEL.SPEC_SEQ
SEL.SPEC ~ SEL.SET TO TYPE.SET{←EXP}?{PERM|IMPERM}?
SEL.SET ~ CONSTANT | [TYPE_IDENT] | {INT|INTEGER_IDENT}
          THRU {INT|INTEGER_IDENT}
CONSTANT ~ INT | IDENT | STRING
```

Each node type has a "separate" flag (SEP) to be described later (see EQ primitive) and a set of selector specifications. Each of these has a selector set which is either a type or a constant. (These should all be disjoint, but the compiler will not check for this. If they are not, then when an ATTACH is done, it will use one of the correct ones depending on implementation.) Associated with each selector set is (a) an allowed type (TYPE.SET) which specifies what are the types which links named by these selectors may point to, (b) an initial expression, which specifies what they are to point to initially when they are created, and (c) a specification of whether or not they are permanent. If a selector set is permanent, it must be a constant (not a [TYPE_IDENT]); then each time a node of this type is created a link is attached automatically for each permanent selector. If the permanent specification is omitted, then the selector set is taken to be permanent if it is a constant and impermanent if it is a [TYPE.SET]. If the initial expression is omitted, it is taken to be UND. If the selector set is of the form "A THRU B TO C", where A and B are integers, A < B, this is equivalent to

```
 A  TO C
A+1 TO C
 :
 :
B-1 TO C
 B  TO C
```

If an identifier A appears as a selector set, then, if it has not been declared as anything else, an implicit declaration is made as follows:

```
STRING A IS "A"
```

```
Examples:  TYPE VEC10 IS
               1 THRU 10 TO INTEGER ← 0 IMPERM
           TYPE SETINT IS
               [INTEGER] TO NULL
           TYPE GLOB IS
               A TO ANY
               [INTEGER] TO GLOB
```

(f) NULL. NIL and UND. UND is of type NULL, but it is also of every allowable type. This is so it may be used as the contents of an undefined link of any type. NIL is a special constant which is the result of a routine which does not have a result. In general it may be used anywhere that a result needs to be returned indicating an error condition. It will cause a type error, unless it is expected, since it is of type NULL.

(g) ROUTINE. i) Primitive routines are built into VERS. They are the basic units of computation of the VERS machine. They are listed in the next section. ii) Defined routines are specified in the program. The execution of these routines and the evaluation of expressions are described in section F.

A declaration must always occur before the use of the object in the lexicographic order of the program except that a label may appear in the context "GOTO L" before it appears labelling a statement. Other uses of a label must follow its declaration. There are certain cases (types, labels, and routines) where the declaration cannot always precede the use. In these cases the user must precede the use by a "forward declaration" such as "ROUTINE R" to let the compiler know the type of R. Then, of course, later he must satisfy the forward declaration by putting in the real declaration. For labels and types, the forward declaration must be satisfied within the same block (data block or routine) in which it appears. Forward

declarations of routines need not be satisfied to allow compilation of the program; only a warning is given. An error will result if the program tries to call such a routine, of course.

If an unsatisfied forward declaration of a type or label is detected, an error message is printed followed by "FORGET?". The user should respond "NO" if he intends to simply correct the error and compile his program, and if he can do this by editing only the block in which the error occurred. This will normally be the case. If he intends to do editing outside that block, he should respond "YES" so that the compiler knows to forget what it has done and recompile the block in question.

## E. Primitive Routines

Each routine is described by a header which is in the normal VERS syntax for declaring defined routines (see Defined Routines).

ROUTINE FOLLOW(A TO ANY, N TO NODE) TO LINK

This produces as a result the link from node N selected by A. It is an error if the selector does not exist from N.

ROUTINE STORE(L TO LINK, A TO ANY) TO ANY

This causes link L to point to A if the type of A belongs to the allowable types for L to point to. Otherwise it is an error. It's result is A.

ROUTINE CONTENTS(L TO LINK) TO ANY

This produces as a result the object pointed to by L.

ROUTINE ATTACH(A TO ANY, N TO NODE) TO LINK

This creates a link L from N named A which is its result. It is an error if the link already exists or if A is not of a legal selector set for the node type of N. The link type of L is set to the allowed type of the selector set. The initial expression of the selector set is evaluated to V and STORE(L,V) is executed.

ROUTINE DETACH(A TO ANY, N TO NODE)

This deletes the link selected by A from node N if it exists. It is an error if it doesn't.

ROUTINE TEST(A TO ANY, N TO NODE) TO BOOLEAN

The result is TRUE if there is a link selected by A from node N and FALSE otherwise.

ROUTINE CREATE(T TO TYPE) TO NODE

If T is not a node type it is an error. A node N of type T is created and returned as the result. For each permanent selector type S of T execute ATTACH(S,N).

ROUTINE SEL(N TO NODE) TO SEQ

This produces a sequence (SEQ) of the currently existing selectors from node N.

ROUTINE TYPE.OF(A TO ANY) TO TYPE

Returns the node type if A is a node, the atom type if it is an atom, and LINK if it is a link.

ROUTINE KIND(A TO ANY) TO TYPE

Returns NODE, LINK, or ATOM depending on which of the three A is.

ROUTINE BELONGS(A TO ANY, T TO TYPE) TO BOOLEAN

Returns TRUE if TYPE.OF(A) = T (if T is not compound) or if TYPE.OF(A) is included in T (if T is compound).

The following routines perform ordinary integer arithmetic:

ROUTINE PLUS(I TO INTEGER, J TO INTEGER) TO INTEGER

ROUTINE MINUS(I TO INTEGER, J TO INTEGER) TO INTEGER

ROUTINE MULT(I TO INTEGER, J TO INTEGER) TO INTEGER

ROUTINE DIV(I TO INTEGER, J TO INTEGER) TO INTEGER

ROUTINE MOD.(I TO INTEGER, J TO INTEGER) TO INTEGER

Returns the remainder upon integer division.

ROUTINE NEG(I TO INTEGER) TO INTEGER

Returns the negative of I.

ROUTINE GOTO.(L TO LABEL)

Transfers control to the statement referred to by L.

ROUTINE IFGO(B TO BOOLEAN, L TO LABEL)

Transfers control to the statement labeled L if B is true, otherwise does nothing.

ROUTINE NO.OP( )

No operation.

The following routines perform respectively logical conjunction, disjunction, negation, and arithmetic greater than and less than tests:

ROUTINE CONJ(B TO BOOLEAN, C TO BOOLEAN) TO BOOLEAN

ROUTINE DISJ(B TO BOOLEAN, C TO BOOLEAN) TO BOOLEAN

ROUTINE LOG.NEG(B TO BOOLEAN) TO BOOLEAN

ROUTINE GT(I TO INTEGER, J TO INTEGER) TO BOOLEAN

ROUTINE LT(I TO INTEGER, J TO INTEGER) TO BOOLEAN

ROUTINE EQ(A TO ANY, B TO ANY) TO BOOLEAN

If A and B are links or atoms, an ordinary identity test is performed. If they are nodes with the separate flag off, an ordinary identity test is also done. If the separate flag is on, however, A and B are e ual if they both have the same set of selectors and if the contents of corresponding links are equal. Notice that if these contents are also "separate" nodes, then the process is applied again.

Warning: Do not create a SEP node which links to itself, or links to a chain of other SEP nodes in which there is eventually a link back to itself. This may cause an infinite recursive loop in VERS, which will result in a STACK OVERFLOW message.

ROUTINE IDENT(A TO ANY, B TO ANY) TO BOOLEAN

An ordinary identity test is performed, even if A and B are nodes with the separate flag on.

ROUTINE UNPACK(S TO STRING) TO SEQ

This produces a sequence (SEQ) of strings which are the characters of S in order.

ROUTINE READ( ) TO STRING

Reads the next line from the teletype up to but not including CR and returns it as a value.

ROUTINE PRINT(S TO STRING)

Prints the string in S on the teletype.

ROUTINE NAME(A TO ANY) TO STRING

The result is a string which stands for A. It is not the name of the variable containing A; it will only be a variable name if A is a static link. The different types produce results as follows:

INTEGER -- a string of decimal digits

BOOLEAN -- 'TRUE' or 'FALSE'

STRING -- itself

LABEL -- the identifier which labels the statement

TYPE -- the identifier which stands for the type in a program

NULL -- 'NIL' or 'UND'

ROUTINE -- the routine identifier

LINK -- the link name in the program if it is static and otherwise 'LINK'

NODE -- the node name in the program if it is static and otherwise 'NODE'

ROUTINE CONCAT(S1 TO STRING, S2 TO STRING) TO STRING

Forms the concatenation of the two strings.

F. Defined Routines

```
ROUTINE ~ ROUTINE.HEAD{IN ROUTINE_IDENT_LIST}? CR
          DECL_SEQ?
          STATEMENT_SEQ
          END |
          ROUTINE.HEAD EXP
ROUTINE.HEAD ~ ROUTINE IDENT(PAR_LIST?)
               {TO TYPE.SET}?
PAR ~ IDENT {TO TYPE.SET}?{\EXP}? | @ IDENT{\EXP}?
```

A defined routine consists of the following:

(1) A sequence of formal parameters (PAR). Each of these is a link plus a default expression. If "@" is present, then the parameter is called by reference (see below), otherwise it is called by value. The TYPE.SET indicates the link type; it is taken as ANY if omitted. If the parameter is called by reference, the link type should not be specified because it will be that of the link which is passed. The default expression is taken as UND if omitted.

(2) A set of locals. These are all the links and nodes declared following the routine header.

(3) A result link, which is included in the locals with an initial expression of UND (or NIL if TYPE.SET has been omitted) and a link type of the TYPE.SET following the PAR_LIST. This link may be referred to as RESULT inside the routine.

(4) A sequence of expressions, called the code.

A defined routine is executed by making a new copy of it in which the links which are parameters called by value and the local links and nodes are replaced by new links or nodes of the same type; the corresponding replacements for these objects are made in the code. Links for parameters called by reference are replaced throughout the code by the actual parameter which is

being passed (which must be a link). Then for each parameter called by value a STORE of the actual parameter into the formal parameter is executed. If the actual parameter is UND then there is a STORE of the default expression into the formal parameter. Then for each local link a store of the initial expression into the link is executed. Each local node is initialized as if a CREATE were done on its node type. Then the expressions are evaluated in order except for changes in flow caused by GOTO., IFGO., or monitoring. The execution ends when a GOTO.(RETURN.) or GOTO.(ERROR) is executed. The value of the routine is the object pointed to by the result link at that time.

Intuitively, an expression is a constant or some nested calls on routines. More precisely, it is either (1) an object or (2) a routine expression plus a sequence of actual parameter expressions plus optionally an error label. The value of an expression is either (1) the object itself or (2) it is obtained by first evaluating the routine expression, then each of the actual parameter expressions in turn. Then if it is a defined routine it is executed and if it is a primitive routine, it performs the action which is specified under primitive routines. If the primitive routine makes an error or if the defined routine executes a GOTO.(ERROR) and the expression has an error label, then control is transferred to that label when the error occurs. If either of these happens and there is no error label, an error message is printed and GOTO.(BREAK) is executed. In addition, if the defined routine makes an error other than GOTO.(ERROR), it will print an error message and GOTO.(BREAK) even if it has an error label.

In addition, the expression has an "automatic contents" flag, which is normally on. In this case, if the expression is a static link or a call on FOLLOW or ATTACH, then the contents of the resulting link is automatically taken except in the following cases:

(a) The link is an actual parameter on a call to a routine which expects this parameter by reference.
(b) It is suppressed explicitly (@ in the syntax).
(c) It is on the left side of an assignment.

Notice that any nodes or links which are created during the execution of a routine and are pointed to by global links, will remain in existence after the routine is finished. This includes static nodes and links. If the routine is then entered a second time new copies of local static nodes and links are created, and the old ones (if they are globally referenced) will still exist.

## G. Statement and Expression Syntax

```
STATEMENT ~ UNLAB.STATEMENT | IDENT : UNLAB.STATEMENT
UNLAB.STATEMENT ~ STMT |
                  IF.CLAUSE CR
                  STATEMENT_SEQ
                  {ELSE CR
                  STATEMENT_SEQ}?
                  ENDIF |
                  FOR.CLAUSE CR
                  STATEMENT.SEQ
                  ENDFOR
IF.CLAUSE ~ IF EXP THEN
FOR.CLAUSE ~ FOR EXP ← EXP, EXP WHILE EXP DO |
             FOR EXP ← IN EXP DO
```

A label on a statement causes an implicit declaration of the identifier as a static atom of type LABEL whose value is the labelled statement. IF statements and FOR statements are defined by the following equivalences:

| Statement | Code |
|---|---|
| IF A THEN | IFGO.(NOT A, L1) |
| B | B |
| ENDIF | L1: NO.OP( ) |

| Statement | | Code |
|---|---|---|
| IF A THEN | | IFGO.(NOT A, L1) |
| B | | B |
| ELSE | | GOTO L2 |
| C | L1: | C |
| ENDIF | L2: | NO.OP( ) |
| | | |
| FOR A ← B, C WHILE D DO | | A ← B |
| E | LOOP: | IF NOT D THEN GOTO OUT |
| ENDFOR | | E |
| | | A ← C |
| | | GOTO LOOP |
| | OUT: | NO.OP( ) |
| | | |
| FOR A ← IN B DO | | LINK T1, T2 |
| E | | T1 ← FIRST OF B |
| ENDFOR | | T2 ← LAST OF B |
| | LOOP: | IF T1 = UND THEN GOTO OUT |
| | | A ← C OF T |
| | | E |
| | | IF T1 # T2 THEN T1 ← N OF T1; GOTO LOOP |
| | | |
| | OUT: | NO.OP( ) |

```
STMT ~ STMT.1 |
       IF.CLAUSE STMT.1{ELSE STMT.1}? |
       FOR.CLAUSE STMT.1
```

These IF and FOR statements mean the same as the previous ones. The reasons for having two forms are for programmer convenience and ease of incremental compilation. The other statements and expressions are defined as follows:

```
STMT.1 ~ STMT.2{;STMT.2}*
STMT.2 ~ ASSIG | GOTO EXP | RETURN{EXP}?
ASSIG ~ EXP | EXP ← ASSIG
EXP ~ EXP.1 | EXP.1 REL.OP EXP.1 | EXP \ LABEL_IDENT
```

```
REL.OP ~ > | >= | < | <= | = | # | FROM | IS
EXP.1 ~ EXP.2 | EXP.1 OP.1 EXP.2
OP.1 ~ + | - | OR
EXP.2 ~ EXP.3 | EXP.2 OP.2 EXP.3
OP.2 ~ * | / | MOD | & | AND
EXP.3 ~ EXP.4 | EXP.4 OF EXP.3
EXP.4 ~ EXP.5 | EXP.4[ASSIG]
EXP.5 ~ UN.OP PRIM | PRIM
UN.OP ~ NOT | $ | - | @
PRIM ~ (ASSIG) | PRIM({EXP}? _LIST) | CONSTANT
```

Parameters which are omitted in a routine call are taken to be UND. If no parameter follows an omitted parameter in the call, then the comma may be omitted as well. The parentheses may never be omitted, even if the routine has no parameters. Notice that omitting parameters may be used to cause default parameter values to be passed (see section F).

| Construct | Code |
|---|---|
| GOTO A | GOTO.(A) |
| RETURN | GOTO.(RETURN.) |
| RETURN A | RESULT ← A; GOTO.(RETURN.) |
| | |
| A ← B | STORE(@A,B) |
| A \ B | routine call A with B as error label |
| A > B | GT(A,B) |
| A >= B | LOG.NEG(LT(A,B)) |
| A < B | LT(A,B) |
| A <= B | LOG.NEG(GT(A,B)) |
| A = B | EQ(A,B) |
| A # B | LOG.NEG(EQ(A,B)) |
| A FROM B | TEST(A,B) |
| A IS B | BELONGS(A,B) |
| A + B | PLUS(A,B) |
| A - B | MINUS(A,B) |
| A OR B | DISJ(A,B) |
| A * B | MULT(A,B) |
| A / B | DIV(A,B) |

| Construct | Code |
|---|---|
| A MOD B | MOD.(A,B) |
| A & B | CONCAT(A,B) |
| A AND B | CONJ(A,B) |
| A OF B | FOLLOW(A,B) |
| A[B] | FOLLOW(B,A) |
| NOT A | LOG.NEG(A) |
| $A | CONTENTS(A) |
| -A | NEG(A) |
| @A | A with automatic contents flag suppressed |
| $A(B_1,\ldots,B_n)$ | routine call |

In addition, if a label L appears as an entire STMT.2, that is equivalent to GOTO L.

## H. Predeclared Objects

All words underlined in the grammar are reserved words or identifiers and cannot be redeclared by the programmer. In addition there are a number of other identifiers which are reserved by the system. These have a type and a meaning as follows:

1) All the primitive routine names are of type ROUTINE with the obvious meaning.

2) All the types mentioned in the basic objects section are of type TYPE with the obvious meaning.

3) UND and NIL are of type NULL.

4) TRUE and FALSE are of type BOOLEAN.

5) INF is of type integer. It stands for the largest integer in VERS $(2^{15} - 1)$.

6) The following identifiers are of type LABEL:

   a) BREAK: transfer control to the teletype

   b) RETURN.: return from the current routine

c) ERROR: return from the current routine by its error exit (if it has one, otherwise it is an error, see section F).

7) LINK CUR.ROUT TO ROUTINE

This link contains the defined routine which is currently being executed, or UND if there is none.

8) LINK RESULT TO result-type

This link may be stored into by the code of a routine in order to indicate what value it should return. The link type will be the type which the programmer declares for the routine. It is of type NULL if there is no type associated with the routine. There is a different RESULT link for each routine and each is local to its routine.

9) In addition, sequences are a very important part of VERS, even though they are not basic objects, so they involve several predeclared identifiers of various types. A SEQ node has a FIRST and LAST link which delimit the ends of a linked list of the elements of the sequence. A null sequence has FIRST and LAST set to UND. The declarations are as follows:

```
TYPE SEQ IS
     FIRST TO LISTEL
     LAST TO LISTEL
TYPE LISTEL IS
     C TO ANY
     N TO LISTEL
```

Note that this also predeclares FIRST, LAST, C, and N as strings.

## I. Commands and Monitoring

This section describes what may be typed from the teletype in VERS. Whenever the VERS command processor is waiting for a response from the user, it types out "+".

COMMAND ~ STMT

This is an immediate statement. It is compiled and executed as if it were in the routine which was being executed when control was transferred to the teletype. If there is no current routine, then an immediate statement may still be executed, but it may only reference things declared in a global data block. After execution, the statement is removed and can never be referred to again. All immediate statements must be preceded by at least one blank when typed in at the teletype. All other commands, whose descriptions follow, must begin in column one to distinguish them from immediate statements. Immediate statements may not be continued.

COMMAND ~ EDIT CR

The standard CAL TSS editor is called to edit the VERS source file. Whenever an editing command implies adding a line to the file, a preliminary syntax check is performed, and the line is not added if the check fails. This check may be postponed by placing a ? in column 1 of the line. (This is necessary for sub-lines of a continued line.) It will then take place at the end of the editing session.

If the editor is in insert mode and the line is accepted by VERS, it prints out a "." to indicate this. If the line is not accepted, it prints an error message and returns the user to insert mode. At the end of an editing session VERS performs some more global errors checks on the program as well as checking any postponed lines. If VERS discovers an error at this point, it returns the user to the editor in order to correct the error.

The editor is exited in the normal way by F and returns to VERS. The user may also type FIN, which exits from both the editor and VERS. Warning: Do not exit from this kind of editor call using Q or F,FNAME. This may cause trouble. However, one may exit from EDIT FNAME in any of

the normal ways. When the editor is called using EDIT, it creates a temporary file XVINP, which contains the VERS program.

COMMAND ~ EDIT FNAME CR

The FNAME (file name) is as defined in the normal TSS commands. This calls the standard CAL TSS editor on the specified file as if it were called in the ordinary way from the TSS command processor. No syntax checking is done on lines entered.

COMMAND ~ {IN ROUTINE.IDENT}?{SET | UNSET} TRAP.LIST CR

TRAP ~ LINK.IDENT | LINE.DESIGNATOR |
LINE.DESIGNATOR THRU LINE.DESIGNATOR

LINE.DESIGNATOR ~ LABEL.IDENT{{+|-}INT}? | INT | - INT

This command provides the facility to SET and UNSET trap flags on LINK's and lines of code. Flags are normally UNSET. If a flag is set on a line, control will be given to the user before the line is executed. If a flag is set on a link, control will be given to the user whenever a STORE is attempted on the link, before the STORE takes place. After a trap occurs the user may continue execution by typing CONT.

If no routine is specified, the link or line to be trapped on is taken relative to the current routine. An integer LINE.DESIGNATOR i refers to the $i^{th}$ line of the specified routine or the current routine (The routine header is line 0.). If there is no current routine, it is taken to be the $i^{th}$ line of the program (again starting at 0). A negative line designator (-i) means to put the trap on the $i^{th}$ line before the specified (or current) routine. Links and labels may not be specified unless the routine is given explicitly or there is a current routine.

Examples: IN R SET L+3 THRU L+6
SET L-2 (must be a current routine)
IN R SET X
UNSET 12

COMMAND ~ INIT CR

This instructs the system to forget the current state of the execution of the program, and to return to inactive state (see section J). It does not change the state of any traps which may be set.

COMMAND ~ RECOMP

This recompiles the entire program. This has the effect of an INIT plus unsetting all traps. In addition it allows the compiler to reclaim some space which has been lost.

COMMAND ~ CONT CR

Continues execution at the last point where it was interrupted by a trap or a break.

COMMAND ~ WHO | FIN

As in all TSS subsystems.

J. Execution Control

Initialization of global data is controlled as follows: The program is in inactive state when the user first enters VERS and just after an INIT or RECOMP. It remains in this state until the user first attempts to execute something. At this point it enters active state, initializes the global data structures, prints out "INITIALIZED", and executes the user's code, in that order. Notice that the INIT or RECOMP may be initiated by the VERS system because of certain editing which has been done (see below).

If a declaration is added to a routine which is currently being executed, then it will not be initialized (if it contains initialization) at that time. Of course if the routine is then called in the ordinary way, the locals for that call will be initialized. If a global declaration is added to a data block during execution, it will not automatically be initialized. The

programmer must do an INIT to get this effect.

Similarly, whenever the compiler is forced to recompile a data block because of editing which has been done, its static links and nodes will then be uninitialized. The compiler types out "SCOPE INVALIDATED" to indicate this.

Definitions: (1) An active routine is one which is currently executing or one which has executed a call on an active routine.

(2) An active statement is one which is currently being executed or one which has executed a call on an active routine.

If the program has returned control to the teletype during execution, and the user has then edited his source file, he may continue execution of this program unless one of the following things happen:

(1) He edits a line, containing a routine header, data block header, or END.

(2) He causes the scope of a line (the data block or routine in which the line resides) to be changed.

If either of these happens, the system prints "TOTAL RECOMPILATION NECESSARY" and does a RECOMP.

(3) He edits or deletes a declaration in an active routine.

(4) He edits or deletes a line containing an active statement.

If either of these happens, the system will print an appropriate message and do an INIT.

If a label is stored in a data structure, and then moved by editing the source, the stored label will refer to the new statement which is labeled. If the meaning of a node type is changed by editing, all future operations on a node of that type will be governed by the change, even if the node was

created before the change. Static nodes and links remain in existence as long as they are referenced, even if their declaration is deleted or changed.

K. Library Routines

Each of these routines resides on a file whose name is the first seven characters of the routine name. There is no special provision for library routines in the VERS processor. You just read them into VERS in edit mode. Some routines in the library use others in the library. In this case you must be sure to read in the routine which is used first.

ROUTINE CLEARNODE(ND TO NODE)

Detaches all links from ND.

ROUTINE P1(A TO ANY) USES CLEARNODE

Prints A in the following format:

If A is an ATOM, it calls PRINT(NAME(A)).

If A is a LINK, it prints "←" followed by the contents of A in P1 format.

If A is a SEQ it prints $[B_1,\ldots,B_n]$ where the $B_i$ are the elements of the SEQ.

If A is a NODE, it prints $(B_1,\ldots,B_n)$ where the $B_i$ are the objects pointed to by the selectors of A. It will not go into a loop even if the V-graph has a loop in it. If it reaches a node which printed before, it prints out the name of the node if it is static and "$NODE_i$" if it is dynamic, where i indicates that it was the $i\underline{th}$ node printed out in this call on P1.

ROUTINE P2(A TO ANY) USES CLEARNODE

Prints A in a format which is the same as P2 except that for a NODE A, it prints out $(B_1$ TO $C_1,\ldots,B_n$ TO $C_n)$ where the $B_i$ are the selectors

of A and the $C_i$ are their contents. In addition, if $C_i$ is UND, it omits the "TO $C_i$".

ROUTINE PACK(S TO SEQ) TO STRING

This is the inverse of UNPACK. It converts a sequence of STRING's into one large STRING. It returns an error if any of element of S is not a STRING.

ROUTINE SCAN( ) TO SEQ USES ADDL

This reads a STRING S from the teletype, and returns a sequence of STRING's which are the tokens in S, where a token is defined as a sequence of characters separated by blanks.

ROUTINE ADDF(A,S TO SEQ)

Adds A as the first element of S.

ROUTINE DELF(S TO SEQ) TO ANY

Deletes the first element of S and returns it as a value. Returns an error if S is empty. Notice that ADDF and DELF can be used as push and pop on a stack S.

ROUTINE ADDL(A,S TO SEQ)

Adds A as the last element of S.

ROUTINE DELL(S TO SEQ) TO ANY

Deletes the last element of S and returns it as a value. Returns an error if S is empty. This routine has to search through the sequence S.

This is only an initial set of library routines. More will be added as time goes on.

## APPENDIX A - Hints and Warnings

Hint: The message "SYNTAX ERROR" may be given because of a misdeclared identifier. Be especially careful of reserved and predeclared identifiers such as "N", "C", and "TEST".

Hint: If anything wierd happens, try a RECOMP. It might go away.

Warning: If you get an error message followed by a "%" prompt character or the bead ghost, it is either a system error or you have run out of some kind of space. See Jay Earley.

Hint: Don't try to run without at least P1 or P2 from the library. VERS was not intended to be used without some sort of print routine.

Known Bug: In the message given for a break or an error, the line will sometimes be wrong: If the error occurs in the header of a FOR statement, the line printed may be the endfor.

Hint: If you get an error during a routine call or during the initialization of its locals, the line indicated in the error message will be the line containing the call, not the routine header.

## APPENDIX B - VERS Syntax

This is simply a collection of all the BNF in Part II of the manual. See II.A for the metalanguage conventions used.

```
PROGRAM ~ {DATA.BLOCK | ROUTINE} SEQ

DATA.BLOCK ~ DATA{IN ROUTINE_IDENT_LIST}? CR
             DECL_SEQ
             END

DECL ~ NOTE.TYPE_IDENT IDENT_LIST

DECL ~ LINK{IDENT {TO TYPE.SET}?{←EXP}?}_LIST

TYPE.SET ~ TYPE_IDENT{OR TYPE_IDENT}*

DECL ~ TYPE_IDENT IDENT {IS{-?INT|STRING}}?

DECL ~ TYPE TYPE_IDENT IS TYPE.SET

DECL ~ TYPE IDENT SEP? IS CR
       SEL.SPEC_SEQ
```

```
SEL.SPEC ~ SEL.SET TO TYPE.SET{←EXP}?{PERM|IMPERM}?

SEL.SET ~ CONSTANT | [TYPE_IDENT] | {INT|INTEGER_IDENT}
          THRU {INT|INTEGER_IDENT}

CONSTANT ~ INT | IDENT | STRING

ROUTINE ~ ROUTINE.HEAD{IN ROUTINE_IDENT_LIST}? CR
          DECL_SEQ?
          STATEMENT_SEQ
          END |
          ROUTINE.HEAD EXP

ROUTINE.HEAD ~ ROUTINE IDENT(PAR_LIST?)
               {TO TYPE.SET}?

PAR ~ IDENT {TO TYPE.SET}?{\EXP}? | @ IDENT{\EXP}?

STATEMENT ~ UNLAB.STATEMENT | IDENT : UNLAB.STATEMENT

UNLAB.STATEMENT ~ STMT |
                  IF.CLAUSE CR
                  STATEMENT_SEQ
                  {ELSE CR
                  STATEMENT_SEQ}?
                  ENDIF |
                  FOR.CLAUSE CR
                  STATEMENT.SEQ
                  ENDFOR

IF.CLAUSE ~ IF EXP THEN

FOR.CLAUSE ~ FOR EXP ← EXP, EXP WHILE EXP DO |
             FOR EXP ← IN EXP DO

STMT ~ STMT.1 |
       IF.CLAUSE STMT.1{ELSE STMT.1}? |
       FOR.CLAUSE STMT.1

STMT.1 ~ STMT.2{;STMT.2}*

STMT.2 ~ ASSIG | GOTO EXP | RETURN{EXP}?

ASSIG ~ EXP | EXP ← ASSIG
```

```
EXP ~ EXP.1 | EXP.1 REL.OP EXP.1 | EXP \ LABEL_IDENT

REL.OP ~ > | >= | < | <= | = | # | FROM | IS

EXP.1 ~ EXP.2 | EXP.1 OP.1 EXP.2

OP.1 ~ + | - | OR

EXP.2 ~ EXP.3 | EXP.2 OP.2 EXP.3

OP.2 ~ * | / | MOD | & | AND

EXP.3 ~ EXP.4 | EXP.4 OF EXP.3

EXP.4 ~ EXP.5 | EXP.4[ASSIG]

EXP.5 ~ UN.OP PRIM | PRIM

UN.OP ~ NOT | $ | - | @

PRIM ~ (ASSIG) | PRIM({EXP}? _LIST) | CONSTANT


COMMAND ~ STMT

COMMAND ~ EDIT CR

COMMAND ~ EDIT FNAME CR

COMMAND ~ {IN ROUTINE.IDENT}?{SET | UNSET} TRAP.LIST CR

TRAP ~ LINK.IDENT | LINE.DESIGNATOR |
       LINE.DESIGNATOR THRU LINE.DESIGNATOR

LINE.DESIGNATOR ~ LABEL.IDENT{{+|-}INT}? | INT | - INT

COMMAND ~ INIT CR

COMMAND ~ RECOMP

COMMAND ~ CONT CR

COMMAND ~ WHO | FIN
```