Snobol4

A Computer Programming Language for the Humanities

Robert Gaskins, Jr.

Laura Gould

University of California

Spring, 1972

Nothing amuses more harmlessly than computation, and nothing is oftener applicable to real business or speculative inquiries. A thousand stories which the ignorant tell, and believe, die away at once when the computist takes them in his grip.

Samuel Johnson,
Letter to Sophia Thrale
(at Bath), July 24, 1783

CONTENTS

[Note: the starred sections are not yet available 4/1/72]

Appendixes

PREFACE

Edmund Fuller has described hearing an interview in which Edward R. Murrow asked Mickey Spillane how he could bring himself to pander to the public taste by writing the kind of books he did; Spillane's luminous reply, according to Fuller, was: "I write the kind of books I want to read and can't find."

We, with much the same motivation, have written this description of Snobol4, a computer programming language for the humanities. Our own training and interest is in the study of language and literature, and so the examples and exercises are directed particularly toward the machine manipulation of linguistic data and literary texts. Even so, the description should be useful to students of many disciplines, since the first part of each chapter presents features of the language in a generalized way, and the particular examples in the second part of each chapter have been chosen to exhibit principles and techniques which can easily be applied to verbal or symbolic data in a wide range of humanistic and social science applications.

This presentation of Snobol4 is particularly designed for members of the University of California community who have no previous knowledge of computers or computer programming. It describes a dialect of the language for Control Data Corporation 6000 series machines, implemented at the Berkeley Computer Center by Paul McJones and Charles Simonyi; Mr. McJones has reviewed our work as it has progressed, and has made many helpful suggestions.

It is intended that this manual will be expanded to provide a complete description of the Snobol4 language and of various related facilities available at the Berkeley Computer Center which are of interest to Snobol users. We would naturally be pleased to receive suggestions for improvements and additions from readers. We hope that few mistakes remain, even in this preliminary version, but each of us blames the other for any that may be found.

1A. COMPUTER PROGRAMMING IN SNOBOL

Snobol is a programming language, one of many such artificial languages which may be used to convey instructions to a computer. Most computers may be instructed in a wide variety of programming languages; these languages differ from one another, as do natural languages, by having different vocabularies and syntactic structures. More importantly, however, they differ in the range of concepts which they are capable of expressing.

Different programming languages have been developed for different kinds of problems or problem areas. Some have been devised primarily for describing general numeric or algebraic problems, others for describing the structure of business records and files, still others for highly specific purposes such as controlling machine tools, simulating economic systems, or making computer-generated movies. Snobol is distinguished by very powerful and general capabilities for manipulating strings of characters, making it particularly convenient for working with data from areas such as linguistics, literature, verbal behavior, and the humanities in general, it is also very useful for expressing sophisticated non-numeric problems in the field of computer science.

<u>Devising a Program.</u> A description of how a computer is to go about solving a problem consists of a list of tasks or actions to be performed. A specification in some programming language which describes such a series of tasks completely is called a "program text." Before a program text can be written, the task which it is to describe must be clearly understood. If, for example, a task has been expressed in English as "find all vowels in a word," the following questions must be resolved before the programming of the task in some programming language can be undertaken:

(1) what is a vowel?
(2) what is a word?
(3) what should be done with the vowels which are found?

The answers might be as follows:

(1) one of the characters A,E,I,O, or U
(2) a string of characters to be provided as data to the program
(3) count them and then print the total

Given these clarifications, one can then translate the unrigorous English sentence "find all vowels in a word" into a rigorous step-by-step description of what must be done; this step-by-step description can then be translated again into a series of statements in an appropriate programming language. The intermediate translation may exist only in the mind of the programmer, as is often the case if the task is a simple one, or may be recorded in some fashion so that it may be considered for correctness.

One of the best ways of recording a step-by-step description is to write down a series of numbered statements specifying exactly what is to be done. These statements are still in English, but a much more detailed and careful English than that of the original problem. The statements differ from the sentences of a natural language paragraph in that they are not intended to be processed only once or in the order in which they are presented; hence, the statements are numbered so that the order in which they are to be processed, often repeatedly, may be specified. A set of numbered statements describing how to count all the vowels in a series of words and to print the counts might look as follows:

START
(1) Get the next word; if no more words, STOP.
(2) Print that word.
(3) Set the tally to zero.
(4) Get the next character of this word; if no more characters remain, go to (7); otherwise go to the next statement.
(5) Determine whether or not this character is an A,E,I,O, or U; if it is not, go back to (4); otherwise go to the next statement.
(6) Add one to the tally which is keeping track of the number of vowels in this word; go back to (4).
(7) Print the value of the tally, which now represents the total number of vowels in the word. Go back to (1) and attempt to get another word.

Note that this program description has been augmented to count the vowels in any number of words, one after another, and to print the counts separately. It would not be useful to write a program to count the vowels in a single word only, as the counting could be accomplished by hand much faster than the program could be written. (However, for more complicated tasks, a program can often be written much more easily than the task can be performed even once by hand; that such a program could then be used again might well be of secondary importance.)

Another method of recording a step-by-step description is to use what is called a "flow chart." In a flow chart the specification of what is to be done next, or the "flow of control," is indicated by means of lines and arrows rather than by phrases of the form "go back to (1)." A flow chart equivalent to the numbered statements just provided might look as follows:

```
     START
       |
       |<--------------------------------------------------------.
       |                                                         |
(1)    v                                                         |
 .-----------.                                                   |
 | get  next |Fail                                               |
 |   word    |-------> STOP                                      |
 '-----------'                                                   |
       |Succeed                                                  |
       |                                                         |
(2)    v                                                         |
 .-----------.                                                   |
 | print the |                                                   |
 |   word    |                                                   |
 '-----------'                                                   |
       |                                                         |
       |                                                         |
(3)    v                                                         |
 .-----------.                                                   |
 | set tally |                                                   |
 |  to zero  |                                                   |
 '-----------'                                                   |
       |                                                         |
       |<-------------------------------------.                  |
       |<-------------------.                 |                  |
(4)    v               (5)  |Fail        (6)  |                  |
 .-----------.         .-----------.       .----------.          |
 | get  next |Succeed  | test  for |Succeed| add  one |          |
 | character |-------->| A,E,I,O,U |------>| to tally |          |
 '-----------'         '-----------'       '----------'          |
       |Fail                                                     |
       |                                                         |
(7)    v                                                         |
 .-----------.                                                   |
 |   print   |                                                   |
 | value  of |---------------------------------------------------'
 |   tally   |
 '-----------'
```

Writing a Snobol Program Text. Now that a detailed method for solving the problem is clearly understood, it may be translated into a set of statements in the Snobol language. Seven Snobol statements are provided below, one for each of the numbered English sentences, or, equivalently, one for each box of the flow chart. These statements are provided here to illustrate the close correspondence between the Snobol statements and the step-by-step description, to give some indication of the appearance of a programming language, and to point out some features of the Snobol language in particular; a complete discussion of the meaning of these statements must be deferred to later chapters of the text. (Comments, beginning with asterisks, have been inserted for spacing and to explain the purpose of the statements.)

```
* STEP 1:  READ IN THE NEXT WORD - IF NO MORE WORDS, STOP
*
READ     WORD  =  TRIM(INPUT)              :  F(END)
*
* STEP 2:  PRINT THE WORD JUST READ IN
*
         OUTPUT  =  WORD
*
* STEP 3:  SET THE TALLY TO ZERO
*
         TALLY  =  0
*
* STEP 4:  GET THE NEXT CHARACTER OF THIS WORD - IF NO MORE
*          CHARACTERS, PRINT THE VOWEL COUNT FOR THIS WORD
*
GETCHAR  WORD  LEN(1) . CHAR  =  NULL  :  F(PRINT)
*
* STEP 5:  SEE IF THIS CHARACTER IS A VOWEL - IF NOT,
*          GO BACK AND GET NEXT CHARACTER
*
         CHAR  ANY('AEIOU')                :  F(GETCHAR)
*
* STEP 6:  CHARACTER IS A VOWEL - ADD ONE TO THE TALLY
*
         TALLY  =  TALLY + 1               :   (GETCHAR)
*
* STEP 7:  PRINT NUMBER OF VOWELS AND RETURN TO
*          READ IN THE NEXT WORD
*
PRINT    OUTPUT  =  TALLY                  :   (READ)
*
END
```

Each Snobol statement consists of three basic parts, any of which may be absent. These parts are called the label, the rule, and the go-to. The label is the first part and serves to identify the statement (as did the numbers in the English description above); the rule is the middle part and specifies some action to be performed; the go-to is the last part and indicates which statement is to be considered next by providing its label in parenthesis. (The F within the first three go-to's above indicates that the go-to is to be taken only if the action specified by the rule preceding it fails; otherwise control is sent to the next statement of the series.)

Input and Output. Before the statements of a program text can be used to instruct a computer, they must first be put in what is called "machine-readable form." For instance, they must be punched on cards to be read into the computer's memory via a card reader, or typed in on a teletype connected to the computer. The data to be manipulated, such as the words whose vowels are to be counted, are seldom explicitly provided within a program text, but are prepared separately and must also be put in machine-readable form before they can be accessed.

The Snobol language provides facilities for reading in units of data, called "records," and for writing out the results of manipulating this data. These are called "input" and "output" facilities. The first statement of the program text above indicates that some input is needed; in particular, it specifies that an indefinite number of words, one at a time, are to be read from a "file" of data which must be supplied with the program. The second statement specifies that some output is to be produced; in particular, that the word just read in is to be printed at the beginning of a new line of printer paper. The last statement specifies that the number of vowels found within that word is to be printed on the following line.

If the file of data to be used as input for the program text above were the following list of words

```
HIPPOPOTAMUS
HIPPOS
HIPPOSIDEROS
HIPPOSPONGIA
HIPPOTIGRINE
HIPPOTOMY
HIPPOTRAGINE
HIPPOTRAGUS
```

then the output produced by the program would be the list

```
HIPPOPOTAMUS
5
HIPPOS
2
HIPPOSIDEROS
5
HIPPOSPONGIA
5
HIPPOTIGRINE
5
HIPPOTOMY
3
HIPPOTRAGINE
5
HIPPOTRAGUS
4
```

Results from executing a program may be printed on paper for personal perusal, written on magnetic storage media, or punched on cards. Since the last two are machine-readable as well as machine-writeable, the output may be used again, without modification, as input data to be further processed by still another program.

Execution of a Snobol Program. It is not enough for a computer to have available to it both a program text and some data in machine-readable form; it must also have available to it a "translator" or "system" to process the language in which the program text has been written. A computer may have available any number of language processors and hence may be able to "understand" any number of languages. A processor itself consists of a program, written in some programming language (often in a language that is basic and unique to a particular computer, but possibly in Snobol). The data which such a system will use is a program text in the language for which it is the processor.

The Snobol system described here consists of two separate parts called the "compiler" and the "interpreter." The compiler uses a Snobol program text as its data, reading in the statements one at a time in the sequential order in which they appear. It prints and numbers each statement to be inspected later by the programmer and tests the statement to determine whether or not it is syntactically correct, that is, whether or not it conforms to all the rules governing the proper structure of a Snobol statement. (This process is analogous to parsing a natural language sentence for grammatical correctness.)

If a statement is well-formed, it is converted by the compiler into a representation ("Code") suitable for later processing by the interpreter; if it is not well-formed, it is flagged as being syntactically incorrect. All statements of the program text are processed, even if incorrect ones occur, so that all syntactic errors are found. The programmer can locate the incorrect statements by inspecting the program listing; he can then correct them and once again submit his program text as data for the compiler to process.

If no compile-time errors occur, the message SUCCESSFUL COMPILATION is written at the end of the program listing. The interpreter then starts processing, using the converted statements of the program text as its data; the entire set of converted statements representing a program text is called a "program." The interpreter executes the program, causing the computer to perform whatever task has been described. It starts by executing the first statement of the program and then proceeds to process the converted statements in the order specified by the go-to's, reading input from a data file and producing output whenever requested. Execution continues until the task is finished (as signified here by the END statement) or until an execution-time error (such as a request to multiply 'CAT' by 'CATALOG') occurs. If this happens, the programmer can inspect the error message printed by the interpreter and can attempt to determine his mistake. He can then modify the program text and submit it once again to the joint processes of compilation and execution.

2A. ASSIGNMENT

A Snobol program text consists of a sequence of statements in the Snobol language. These statements are compiled to produce a series of instructions to the computer, causing it to store data in its memory, to perform operations on this data, and to preserve the results for human inspection and/or for further processing by machine. The data to be manipulated is usually stored externally to the program and is read in by the program as it is needed. A few data values, however, are often written directly in the program text itself. These values may be of several different types, but are most often simply strings of characters.

Literal Values. Strings are sequences of characters which may be of any length and may be composed of any characters in the computer's character set (see Appendix I). Strings whose characters are written directly in the program text are called string literals and are designated by being delimited by either single or double quotes; a string consisting of the five English vowels may be written in a Snobol program text as either

'AEIOU' or "AEIOU"

with exactly the same effect. This permits a string literal to contain whichever quote mark is not being used as the delimiter without confusion. For example,

"LADY¤CHATTERLEY'S¤LOVER"

is a string of 23 characters, while

'"AY!"¤HE¤SAID¤BRIEFLY.'

is a string of 22 characters. Notice that spaces (represented here by the symbol ¤) are treated like any other characters in string literals.

Strings consisting of nothing but digits with perhaps an initial plus sign or minus sign are called numeric strings and are of datatype Integer; all other strings are of datatype String. Those strings which are of datatype Integer, and which do not have an initial sign, may be represented in the program text with or without surrounding quotes. If quotes are not used, as in

669 7449 0 23

then these numeric strings are called integer literals. When an integer literal is stored in the memory, any leading zeroes it may have had are removed; that is, the integer is stored in a "canonical" form. (The canonical form of zero is the single character 0.) Thus 00023 and 23 and '23' all have identical representations in the memory. Leading zeroes may be preserved for non-numeric applications by representing integers in the program text as string literals containing leading zeroes. For example, '00023' would be stored as a five-character string, while '23' would be stored as a two-character string. String literals are always stored within the computer's memory exactly as they are represented in the program, while integer literals are always stored in canonical form. In what follows, the term string will be used to include objects of datatype Integer as well as objects of datatype String.

Variables. Once a value of any datatype is stored within the computer's memory, some method must be provided for referring to it so that it may be used repeatedly throughout the program. Each value is stored by being assigned to a variable, which serves as a reference, or pointer, to the value. Every variable has a name, and any non-null string of characters may be used as the name of a variable. That is, the name of a variable may be of any length and may be composed of any characters of the character set. Those names which begin with a letter and consist of an arbitrarily long sequence of letters, digits, and periods are said to be in "identifier form" and may be written directly in the program text. Thus

```
RHYME1      VOWELS      UNSUCCESSFUL.COGNATES      P.V.C
```

are all valid representations of variables in program texts since they are all identifiers, while

```
1RHYME      ..VOWELS      TEST/3      P-V-C
```

are not, since the first two don't begin with a letter, and the last two contain impermissible characters.

String literals, integer literals, and variables thus have representations in a program text which allow them to be easily differentiated from one another: string literals begin with a quote (and must end with a quote as well), integer literals begin with a digit, and names of variables begin with a letter. (Other ways of representing variables, and particularly variables whose names are not in the form of identifiers, are discussed in Chapter 5 and Chapter 7.)

<u>Assignment Rules.</u> The most fundamental kind of rule in the Snobol language is the assignment rule which is used to assign a value to a variable. The variable is usually represented by an identifier and the value can be a String or an Integer or may be of any other datatype (Real, Pattern, Array, etc.). For example, the assignment rule

```
VOWELS  =  'AEIOU'
```

specifies that the five-character string AEIOU is to be stored in the memory as the value of the variable named VOWELS. Similarly

```
COUNT  =  47
```

specifies that the integer 47 is to be stored as the value of the variable named COUNT.

In general, an assignment rule has the meaning: let the variable represented on the left side of the equals sign refer to the value specified on the right side of the equals sign. (It is obvious that the equals sign does not have its usual arithmetic meaning in an assignment rule; it is being used as an "assignment sign.")

An assignment rule may have a variable name on its right side, rather than a literal. When a variable occurs on the right, it is used to refer to its value. Thus the sequence of rules

```
ALEPH  =  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
ALPHA1  =  ALEPH
LETTERS  =  ALEPH
```

specifies that the variable ALEPH is to have as its value the 26-character string of the alphabet, that the variable ALPHA1 is to have as its value the current value of ALEPH, and so forth. In an assignment rule, when the name of a variable occurs on the left of the assignment sign it stands for the variable; when the name of a variable occurs on the right, it stands for the value of that variable.

The relation between a variable and its value need not be a permanent one. Usually a variable is assigned a variety of different values in the course of executing a single program (hence the term "variable"). A variable named WORD, for example, might be assigned as its successive values each new word encountered in a group of data, thus changing its value 10,000 times for a text 10,000 words in length. Each time a value is assigned to a variable, the previous value of the variable is lost; thus the value of a variable is always the one most recently assigned.

The Null Value. All variables, before they have been assigned any other value, start out with the "empty" or null value. After a variable has been assigned a non-null value, it may be given the null value again by executing an assignment rule with a null value on the right side, such as

```
VOWELS =
```

The null value may also be represented by an "empty" literal, one with no characters in it, as in

```
VOWELS  =  ''
```

or

```
VOWELS  =  ""
```

or by a variable which has a null value, such as

```
VOWELS  =  NULL
```

or

```
VOWELS  =  ANYTHING
```

if the variables NULL and ANYTHING have null values when the rules are executed. (In all examples which follow, wherever the variable NULL occurs it is assumed by convention to have a null value.)

The null value is a special entity in Snobol, distinct from all other values, and has a variety of important uses in the language. Notice particularly that it is distinguished from the strings space and zero. Thus

```
VOWELS  =  '□'

VOWELS  =  '0'
```

and

```
VOWELS  =  0
```

are each assignments which give the variable named VOWELS a non-null value; the first value is of datatype String, while the last two are of datatype Integer. Although the null value is a distinct value, it is not given a special datatype; by convention the null value is of datatype Integer. Thus the general term string, which includes objects of datatype String as well as of datatype Integer, includes also the null value unless specified otherwise.

<u>The Special Variable OUTPUT.</u> Once values have been stored within the computer's memory, they may be printed out by assigning them to the special variable OUTPUT. This variable differs from others in having the following special property: whenever the variable OUTPUT is assigned a string as its value, that value is transmitted to a file to be printed on a line printer which is attached to the computer. Each execution of a rule in which OUTPUT is assigned such a value results in the printing of a new line of information (a record). For example, execution of either

```
        OUTPUT  =  'AEIOU'
```

or

```
        OUTPUT  =  VOWELS
```

(if the current value of the variable VOWELS is the string AEIOU) would cause the five letters AEIOU to be printed at the left margin of the next available line of the output paper.

If OUTPUT is assigned a null value, as in

```
        OUTPUT  =
```

or

```
        OUTPUT  =  NULL
```

the result is a null record, which appears as a blank line on the output paper.

OUTPUT may be assigned a string of any length as its value, but only the first 132 characters, the number of characters available per line on a printer, will be printed. The entire string, however, remains the value of OUTPUT and may thus be assigned as the value of other variables as well. The variable OUTPUT, like any other variable, may be used on either side of an assignment rule, as in the sequence

```
        OUTPUT  =  VOWELS
        OUTPUT  =  OUTPUT
        COPY  =  OUTPUT
```

whose execution would result in the two lines of output

```
AEIOU
AEIOU
```

Note that although the special variable OUTPUT is involved in all three rules, no printing is produced by the third because it does not specify that OUTPUT is to be assigned a value; rather, the value of OUTPUT, which at the time the rule is executed is the string AEIOU, is assigned to the variable COPY.

The Special Variable INPUT. Data may be read into the computer's memory by the use of the special variable INPUT, which differs from other variables in that it has the following property: whenever the value of the variable INPUT is needed for the execution of a statement, INPUT acquires for its value the next record of the input file. For example, in the assignment rule

```
LINE  =  INPUT
```

the value of INPUT is needed, so it can be assigned as the value of LINE; LINE receives as its value the string of characters in the next input record.

It is important to recognize that the value of INPUT cannot be saved or used without assigning it to another variable in the same rule in which it is read. The next use of INPUT will refer, not to its present value, but to the next record of the data. Thus the sequence

```
LINE1  =  INPUT
LINE2  =  INPUT
```

assigns two successive records to the two variables LINE1 and LINE2.

This example illustrates an important difference between the variables INPUT and OUTPUT: INPUT displays its special property (to acquire the next record of an input file as value) every time its value is needed, but not when it is assigned a value; OUTPUT displays its special property (to write a record on an output file) every time it is assigned a value, but not when its value is needed. Thus the last value assigned to OUTPUT is always available for assignment to another variable.

The special variables INPUT and OUTPUT may both be used in a single rule, as in

```
OUTPUT  =  INPUT
```

Execution of this rule will cause the characters of the next data record to be printed by the line printer. Repeated execution of such a rule could be used to make a printed listing of an entire group of data (as will be shown in Chapter 3).

The value of INPUT is always 80 characters long, a convention adopted since that is the width of a card and of lines sent from many remote terminals. If the record being read actually has more than 80 characters, the excess is ignored; if it has fewer than 80 characters, spaces are added at the end to fill out the full length. Executing the rule

```
      VOWELS  =  INPUT
```

where the next data record has the five vowel characters starting in the first position, causes the variable VOWELS to be assigned a string consisting of the 5 characters AEIOU followed by 75 spaces.

Other Forms of Input and Output. The input to a Snobol program may exist in the form of punched cards or it may be stored on a disk file or on magnetic tape. The output from a program may be printed on paper, punched on cards, or written on a disk file or on magnetic tape. Snobol provides the special variable INPUT for reading cards and the special variable OUTPUT for producing printed paper, but provides no other special variables for dealing with the other input and output devices listed above. If the programmer wishes to use these other media, he must cause a variable to be associated with a file for input or output, and then use that variable much as INPUT and OUTPUT are used within his program. Methods of associating program variables with input and output files are described in Appendix A, section II.D.

Procedures. The small amount of Snobol so far presented allows one to enter data into the computer's memory (either by writing it directly in the program text in the form of string and integer literals or by using the special variable INPUT) and then to print it out (using the special variable OUTPUT). However, it is seldom the case that the output is to be the same as the input; that is, some manipulation of the data is usually necessary before the desired results can be obtained. One way of manipulating the data is to invoke what is termed a procedure. Many procedures to perform common tasks are already predefined in the Snobol language; a summary of all the predefined procedures which are available may be found in Appendix A. Besides using these predefined procedures, programmers may define their own procedures and add them to the language within their own programs (see Chapter 6).

A procedure is invoked, or called, by writing a procedure reference consisting of the name of the procedure followed directly by its argument list enclosed within parentheses. This means that the Snobol system is to perform the action of the procedure, using its one or more arguments as data, and is to return the result of carrying out the action as the value of the procedure call.

The TRIM() Procedure. The use of the special variable INPUT almost always results in strings which have spaces at the end of them. Since these spaces are often not wanted, a TRIM() procedure is provided by Snobol which accepts any expression whose value is a string as its single argument; the procedure returns as its value the same string but with all trailing spaces removed. Thus those 75 unwanted spaces which occur in the value of VOWELS when the rule

```
VOWELS  =  INPUT
```

is executed may be trimmed off by using the rule

```
VOWELS  =  TRIM(INPUT)
```

instead. This would give VOWELS the five-character value AEICU.

When the rule

```
VOWELS  =  TRIM(INPUT)
```

is executed, the eighty-character value of INPUT (the next record) is obtained, the trailing spaces are removed from it by the TRIM() procedure, and the shortened string is returned as the value to be assigned to the variable VOWELS.

Although the TRIM() procedure is most often used to trim the value of INPUT, it may be used to return the trimmed value of any string given as its argument. For example, in the rule

```
TEXT1  =  TRIM(TEXT2)
```

the call to the TRIM() procedure returns the trimmed version of the string which is the value of TEXT2, to be assigned to the variable TEXT1. The value of TEXT2 remains unchanged: that is, it still contains any trailing spaces it had when the rule was executed. To trim TEXT2 one could use the rule

```
TEXT2  =  TRIM(TEXT2)
```

Note that although variables and procedures may have the same names, there is no confusion in their use in program texts, since procedure names are always followed immediately by an open parenthesis preceding the argument list. Thus one may write

```
TRIM  =  TRIM(TEXT)
```

to assign to the variable TRIM the trimmed value of TEXT.

<u>The SIZE() Procedure.</u> The length of any string may be determined by a SIZE() procedure, which accepts any expression whose value is a string as its argument; the procedure returns as its value an integer which is the number of characters in that string. That is, executing

```
LENGTH1  =  SIZE(VOWELS)
```

would assign to LENGTH1 the integer value 5, while executing

```
LENGTH2  =  SIZE(INPUT)
```

would assign to LENGTH2 the integer value 80. When the argument of SIZE() is a null value, the result is the integer value zero.

The length of the trimmed value of INPUT may be determined by using the procedures TRIM() and SIZE() together. This may be done by using the two procedures in two different assignment rules, such as

```
SAVE  =  TRIM(INPUT)
LENGTH  =  SIZE(SAVE)
```

or, if the value of INPUT were not to be saved but only its length, by combining both procedures in a single assignment rule, such as

```
LENGTH  =  SIZE(TRIM(INPUT))
```

Here the argument of a procedure reference is still another procedure reference; clearly, these nested procedure calls must be processed from the inside out, since the argument of SIZE() is not known until TRIM() has returned the result of its work. As this example shows, an argument of a procedure reference may be any expression which produces a value the procedure is able to accept.

<u>Operators.</u> Data may also be manipulated by means of a number of different operators provided within the Snobol language. Each operator specifies that some sort of operation is to be performed on its operand(s). Operators having a single operand are termed unary operators;

operators having two operands are termed binary operators. Often the same symbol is used in program texts to indicate both a unary operator and a binary operator with different, perhaps completely unrelated, meanings. The meanings are easily differentiated, however, since a unary operator must always directly precede its operand with no intervening blank; a binary operator must always be bounded by blanks. A summary of all the operators available in Snobol may be found in Appendix C.

<u>The Concatenation Operator.</u> One of the most frequently used operators is the concatenation operator. When the operands of this binary operator are strings, it specifies that the two strings are to be concatenated together, i.e., that the second string is to be appended directly to the first. The symbol for this binary operator, since it occurs so often, is simply a single blank (which requires, therefore, no further blanks to separate it from its operands). For example, the assignment rule

```
ALPHA  =  VOWELS  CONSONANTS  'YW'
```

contains two concatenation operators and specifies that the variable ALPHA is to be assigned a string built up by taking the value of VOWELS, followed by the value of CONSONANTS, followed by the two characters YW. If the variables VOWELS and CONSONANTS have previously been assigned the expected values, then the variable ALPHA will be assigned the value of all the characters of the alphabet, in the indicated order. The values of VOWELS and CONSONANTS are in no way changed by the execution of this rule; likewise, subsequent changes in their values can in no way affect the value of ALPHA, which will change only when another rule specifying an assignment to ALPHA is executed.

The variable appearing to the left of the assignment sign may be used within a concatenation on the right as well, as in the rule

```
VOWELS  =  VOWELS  'YW'
```

This rule appends the characters YW to the string which is the current value of VOWELS and then assigns this resulting string as the new value of the variable VOWELS. The old value of VOWELS is thereby lost.

Rules of this form are often used to collect successive characters in an increasingly long string. Execution of the rule

```
LIST = LIST NEWCHAR
```

would cause whatever new character is the value of NEWCHAR to be appended to those already referred to by the variable LIST, and the re-assignment to the variable LIST of this longer string. If LIST had a null value, as it easily might the first time the rule was executed, then it would simply be assigned the same value as that of NEWCHAR; the concatenation would indeed take place as specified but there would be no evidence that it had occurred since the null value contributes no characters to the string.

Note that no spaces are generated by the concatenation process itself. That is, the new characters are appended to the list in the example above in a contiguous fashion with no intervening spaces. If spaces are desired in the result of a concatenation, they must themselves be concatenated into the string, as in the sequence

```
OUTPUT  =  'A¤ROSE'
OUTPUT  =  OUTPUT '¤IS¤' OUTPUT '¤IS¤' OUTPUT
```

whose execution will produce the following output:

```
A ROSE
A ROSE IS A ROSE IS A ROSE
```

More complicated Snobol expressions may be operands of the concatenation operator; for example, the TRIM() procedure may be used to produce a heading, as in

```
OUTPUT  =  '******¤' TRIM(INPUT) '¤******'
```

or

```
HEAD = TRIM(INPUT) '¤' TRIM(INPUT) '¤' TRIM(INPUT)
```

This last rule specifies that the next three data records are to be read, their trailing spaces (if any) trimmed off, and a single space placed between the trimmed content of successive records. The resulting string is then assigned to the variable HEAD by which it may be referenced in other statements of the program.

If an integer literal is involved in a concatenation, it contributes the string of digits representing its numeric value. Thus

```
SUBST  =  VOWELS  0046
```

and

```
SUBST  =  VOWELS  '46'
```

produce the same string as the new value of SUBST, namely AEICU46.

<u>The Arithmetic Operators.</u> Four binary operators are provided within Snobol for doing the four basic arithmetic operations of addition, subtraction, multiplication, and division. The symbols used to represent these operators in the program text are as follows:

addition	+
subtraction	-
multiplication	*
division	/

Since these are binary operators, they must always be bounded by blanks.

The assignment rules

```
ANSWER  =  669 + 527
ANSWER  =  ((A + B) - (C * (-D))) / E
ANSWER  =  (SUM1 / SUM2) + 3
```

would all assign an integer value to the variable ANSWER, provided the variables to the right of the assignment signs all refer to values of datatype Integer when the rules are executed.

Repeated executions of rules of the form

```
COUNT  =  COUNT + 1
```

are often used to count the number of times a given event occurs. These rules are in some ways analogous to ones of the form

```
LIST  =  LIST  NEWCHAR
```

which cause a new character to be appended to the value of LIST; here a new integer, one larger than its predecessor, becomes the value of COUNT. If COUNT had a null value when the rule was executed, it would acquire the value 1 since the null value is considered equal to zero when it is an operand of an arithmetic operator.

The operands of arithmetic operators must always be numeric; that is, they must be any expressions whose values are integers, real numbers (numbers containing decimal points), or null. Real numbers and integers, however, may not occur together within the same arithmetic expression (i.e., mixed mode arithmetic is not allowed). Further information on Snobol arithmetic, including facts about real numbers, conversion of integers into real numbers and real numbers into strings, truncation on division, etc., may be found in Appendix *F.

<u>A Complete Snobol Program Text.</u> Given below is a complete program text which makes use of only a few of the features of the Snobol language already described: it employs only assignment, concatenation, and the special variable OUTPUT; since all data is provided within the program text, the special variable INPUT is not needed. Comments have been inserted in the program text before some statements to indicate their purpose; a comment is distinguished by having an asterisk (*) as its first character. Instructions for representing program texts on punched cards may be found in Appendix H.

```
* PROGRAM TO PRINT A PARTICULAR DESIGN INVOLVING FISH
*       SET UP THE BASIC COMPONENTS
         LT   =  '<'
         GT   =  '>'
         BL4   =  '□□□□'
         BL10   =  BL4 BL4 '□□'
*
* BUILD FISH WHICH SWIM LEFT, SWIM RIGHT, AND MATE
         LFISH   =  LT GT LT
         RFISH   =  GT LT GT
         MFISH   =  LFISH GT
*
* BUILD LONGER STRINGS COMPOSED OF DIFFERENT KINDS OF FISH
         LSWIM = LFISH BL4 LFISH BL4 LFISH BL4 LFISH BL4
         RSWIM = RFISH BL4 RFISH BL4 RFISH BL4 RFISH BL4
         MSWIM = MFISH BL10 MFISH BL10 MFISH BL10 MFISH
         SCHOOL = RSWIM LSWIM
*
* PRODUCE FOUR LINES OF OUTPUT
         OUTPUT = RSWIM RSWIM
         OUTPUT = LSWIM LSWIM
         OUTPUT = SCHOOL
         OUTPUT = MSWIM
END
```

Output from this program is the design shown below.

```
><>    ><>    ><>    ><>    ><>    ><>    ><>    ><>
<><    <><    <><    <><    <><    <><    <><    <><
><>    ><>    ><>    ><>    <><    <><    <><    <><
<><>          <><>          <><>          <><>
```

3A. THE FLOW OF CONTROL

The statements which make up a Snobol program are seldom designed to be executed in the order in which they are written in the program text. Instead, certain segments of the program, consisting of one or more statements each, are intended to be executed repeatedly until some terminating condition is encountered. This condition may be that a certain pattern of characters has occurred in the data, that the data group is exhausted, that the segment has been executed a certain number of times, etc. Once the terminating condition has been met, then repeated execution of another such segment, or "loop," may begin. The choice of the particular segment to be executed can be made dependent on certain features of the data being processed, so the use of the same program with different data will often result in the execution of a different set of statements from within the program. The actual order in which the statements of a program are executed is called the "flow of control."

The flow of control is specified by means of labels which are given to statements for purposes of reference, and by means of go-to's which indicate the statement to be executed next by making reference to its label. The label of a statement is written to the left of its rule, and the go-to is written to the right, as in

```
ASSIGN    VOWELS  =  'AEIOU'              :  (NEXT)
```

Here the label of the statement is ASSIGN, the rule specifies an assignment, and the go-to specifies that the next statement to be executed after this assignment takes place is the one labelled NEXT. If the go-to part of a statement is absent, it is understood that control flows by default to the following statement of the program.

Labels. Any statement may be given a label so that it may be referred to by other statements of the program, or simply by the programmer for his own convenience. A label must always be an identifier and should be chosen so as to be mnemonically useful. Care must be taken when giving statements labels to see that the same label does not occur twice within a single program, or a compile-time error will occur.

Labels are distinguished from the names of variables in a Snobol statement by their position. A label, if present, must always start in the first character position of a statement and must be separated from the rule, if present, by one or more blanks; if a statement is not labelled, the rule must begin with a blank. Because they are distinguished by position, labels and variable names of the same form may be used freely together without confusion, as in

```
VOWELS   VOWELS  =  VOWELS  'YW'
```

which is a statement labelled VOWELS, whose rule specifies that the variable named VOWELS is to have the characters YW concatenated to its value.

It is sometimes convenient to write a statement which consists solely of a label, as in

```
READ
```

since this makes subsections of the program text easy to locate and makes modifications simpler.

<u>Go-to's.</u> The presence of a go-to within a statement is signalled by the occurrence of a colon which serves as an explicit separator between the go-to and any other part of the statement which may have preceded it. Following the colon (which may optionally be bounded by one or more blanks) the information as to which statement is to be executed next is provided by writing the label of that statement within parentheses. For instance, the statement

```
        :  (TEST)
```

consists of a go-to only (it has no label and no rule) and specifies that the next statement to be executed is the one labelled TEST.

Usually a go-to follows a rule, as in the statement

```
        VOWELS  =  TRIM(INPUT)          :  (TEST)
```

which specifies that after the assignment is performed, the next statement to be executed is the one labelled TEST.

The form of the go-to's just shown is called unconditional, because execution of the statement in which they occur will always cause a transfer of control to the statement labelled TEST. More commonly, go-to's are conditional upon the possible failure of the rule which precedes them in the same statement. This causes a choice, or branch, to occur in the flow of control and allows the data to determine which path through the program will be followed next. (Ways in which rules may fail will be indicated presently.)

Conditional go-to's are written like unconditional go-to's, with the addition of a prefixed F (for failure) or S (for success). The statement

```
TEST     LINE  =  INPUT                    :  F(WRITE)
```

specifies that control be transferred to the statement labelled WRITE only if the rule LINE = INPUT fails. Similarly, the statement

```
TEST     LINE  =  INPUT                    :  S(READ)
```

specifies a transfer to the statement labelled READ <u>unless</u> the rule fails (i.e., if it succeeds). In either statement, if the condition for transfer is not met, control will pass by default to the next statement of the program. Thus a conditional go-to always embodies both a success and a failure transfer, even though one of them may be expressed implicitly rather than explicitly. Both a success and a failure transfer may be written explicitly in a single statement as in

```
TEST     LINE  =  INPUT                    :  F(WRITE) S(READ)
```

Since both cases are provided for explicitly, control will never pass to the following statement by default. The order of the success and failure transfers is immaterial and the space between them is optional; the only important requirement is that no blank may intervene between an F or an S and its following open parenthesis.

<u>The Special Transfer END.</u> A go-to specifying a transfer to END is used to terminate execution of a program. This transfer has a special system definition, and constitutes a request to the Snobol system to stop executing. Any number of statements in a program may contain go-to's specifying transfers to END, and the first such transfer to be taken ends execution of the program.

An alternative way of terminating execution is to execute the statement which stands last in the program text, without taking a transfer from it back to some other statement of the program.

There is no restriction against using END as the label of any statement of the program text, but if this is done its special system definition is lost. The convention adopted here is to terminate every program text with a statement consisting solely of the label

```
END
```

A transfer to END causes this last statement to be executed and the flow of control continues on to the next statement; since there is no next statement, the program terminates and the effect is the same as if the system definition of END had not been overridden.

Failure of the Rule. Failure of the rule is not an error and does not cause execution of the program to cease. Rather, it is used to direct the flow of control and to prevent the rule which has failed from continuing execution. When a rule fails, control is sent immediately to the go-to part of the statement so no further processing of the rule is undertaken; in particular, the assignment specified by an assignment rule does not occur. If the statement in which the failure occurs has no go-to, control passes by default to the next statement of the program; if the go-to is conditional (as would usually be the case) the failure transfer, expressed explicitly or implicitly, is taken; if the go-to is unconditional, this unconditional transfer is used.

Failure of INPUT. There are a variety of ways in which a rule can fail. Of the rules presented so far, however, only those which call for the reading of data — those in which the value of INPUT is needed — have any possibility of failing. Such a rule will fail when an end-of-group record is read, i.e., when there are no more data records in the group to become the new value of INPUT. The ability to test for an end-of-group mark, and to direct the flow of control if it is encountered, makes it possible to specify that some process is to be performed on all the records of a data group without having to specify how many records that might be. For example, all the records of a data group, no matter how many there are, may be printed by executing the following very simple complete program text.

```
READ     OUTPUT  =  INPUT                 :  S(READ)
END
```

Every time the statement labelled READ is executed, INPUT acquires the value of the next data record. If that value is not an end-of-group mark, it is assigned to the variable OUTPUT and hence printed. Since the rule has not failed, control is sent back to READ and the process is performed again. This single statement, a one-statement loop, will be executed repeatedly until the end-of-group mark is encountered, causing the rule to fail. In this case the assignment will not take place and the value of OUTPUT will remain unchanged. Control will then flow by default to the statement labelled END, terminating the program.

More than one data group may be processed by a single program since the reading of an end-of-group mark does not prevent further reading of data. The following program text prints two data groups, the first in single-spaced format (as above) and the second in double-spaced format (with a blank line following each record). It prints a message at the end of the first group.

```
READ1    OUTPUT  =  INPUT                   :  S(READ1)
         OUTPUT  =  'END□OF□GROUP□ONE.'
READ2    OUTPUT  =  INPUT                   :  F(END)
         OUTPUT  =  NULL                    :   (READ2)
END
```

The one-statement loop labelled READ1 fails when INPUT acquires the value of the first end-of-group mark, but the next use of INPUT (in the two-statement loop starting at READ2) causes it to acquire the value of the first data record in the second group. Eventually a failure of INPUT will occur in this statement as well, when a second end-of-group mark is read, sending control to END and thus terminating the program.

Evaluation Rules. A rule in a program text consisting of a single expression only is called an evaluation rule. The statement

```
         INPUT                              :  F(DONE)
```

consists of an evaluation rule and a go-to. When such a statement is executed, the single expression of the rule is evaluated, often causing success or failure of the rule to be determined; then the go-to part of the statement, if any, is processed. The statement above indicates that a record is to be read from the input file, and a transfer taken to DONE if that record is an end-of-group mark. No provision is made for preserving the data which is read, but there are some applications in which the data is not needed. The two complete program texts below provide examples of such applications: the first is a program to count the number of records in a group and to print the result; the second prints every other data record in a group, starting with the second record.

```
*  PROGRAM TO COUNT THE NUMBER OF RECORDS IN A GROUP
READ     INPUT                                :  F(DONE)
         COUNT  =  COUNT + 1                  :   (READ)
DONE     OUTPUT  =   COUNT  '□RECORDS'
END

*  PROGRAM TO PRINT EVERY OTHER RECORD STARTING WITH THE 2ND
READ     INPUT                                :  F(END)
         OUTPUT  =  INPUT                     :  S(READ)
END
```

Evaluation rules are commonly used to direct the flow of control through failure of the rule; they can also be used to cause a variable to have a special input or output association attached to it, to define a new procedure, etc., in ways to be described later; in these cases failure of the rule is not involved.

Test Procedures. Failure of the rule may also be caused by the failure of a procedure call which occurs within the rule. Snobol provides nine predefined procedures, called test procedures, which are used primarily to direct the flow of control. Each test procedure accepts two arguments and tests to see whether or not some specified relation, such as equality, holds between them. If the test succeeds, the test procedure returns the null value and execution of the rule continues. If the test fails, the rule of which it is a part fails as well and control is sent immediately to the go-to part of the statement where the failure transfer will be taken.

The Test Procedures IDENT() and DIFFER(). IDENT() and DIFFER() may have arguments of any datatype; they are used to determine whether or not the values of their arguments are identical. In order to be identical, two values must be of the same datatype; if both arguments are of datatype String or both of datatype Integer, then they are tested for character for character identity. Note that the null value is not identical to zero, since zero is represented by a single character, even though the null value is considered equal to zero when used in arithmetic operations. IDENT() and DIFFER() perform exactly the same test but return opposite results: IDENT() fails if its two arguments are not identical, while DIFFER() fails if its two arguments are identical. Thus the following statements are equivalent:

```
         IDENT(STRING1,STRING2)           :  S(SAME)
         DIFFER(STRING1,STRING2)          :  F(SAME)
```

Spaces, of course, must be considered as any other character in the data, so if the rules

```
        STRING1  =  'KING□LEAR'
and
        STRING2  =  'KING□LEAR□'
```

had just been executed, the rule with IDENT() above would fail while the rule with DIFFER() would not.

It is often important, for reasons which will be indicated presently, to know whether or not a given variable has a null value. This can be determined by the execution of

```
        IDENT(STRING,'')                    :  S(EMPTY)
or
        DIFFER(STRING,NULL)                 :  F(EMPTY)
```

or something similar. Since any missing argument of a procedure reference is assumed to be null, the simplest (if not perhaps the clearest) way to write the above statement is in the form

```
        IDENT(STRING)                       :  S(EMPTY)
```

__The Test Procedure LGT().__ LGT() compares two strings to determine whether or not the first is "Lexicographically Greater Than" the second — that is, whether the first __follows__ the second in alphabetical order. For example, the sequence

```
        STR1  =  'ABB'
        STR2  =  'ABC'
        LGT(STR2,STR1)                      :  S(WRITE)
```

will send control to WRITE since ABC alphabetizes after ABB.

The string values being compared may be of any length and may be composed of any characters; the "alphabetic order" of non-alphabetic characters is determined by the order of the computer's character set (see Appendix I). Although the character "space" has special significance in most written languages, it is treated as any other character by the computer, so its relative position within the character set must be taken into account when alphabetizing material containing spaces.

If either of the values being compared by LGT() is not a string, an execution-time error will result.

Arithmetic Test Procedures. The remaining six predefined test procedures compare two numeric values for the following arithmetic relationships:

procedure	relationship
EQ(X,Y)	X equal to Y
NE(X,Y)	X not equal to Y
LT(X,Y)	X less than Y
LE(X,Y)	X less than or equal to Y
GT(X,Y)	X greater than Y
GE(X,Y)	X greater than or equal to Y

All these procedures fail if the indicated relationship does not hold.

EQ() and NE() are very similar to IDENT() and DIFFER(), except that here arithmetic identity, rather than character for character identity, is required. Thus EQ(23,'+00023') will not fail since both arguments have the numeric value of 23, while IDENT(23,'+00023') will fail since character for character identity cannot be found between two strings of different lengths. The expression EQ(NULL,0) succeeds since the null value and zero are arithmetically identical.

If either argument of an arithmetic test procedure has a non-numeric value, an execution-time error results.

Test Procedures within Assignment Rules. Any number of references to test procedures may be embedded within the right-hand side of an assignment rule where they are used not only to direct the flow of control but also to determine whether or not the assignment is to be executed. For example, the statement

```
STRING1 = IDENT(STRING1,NULL) STRING2   :  F(SKIP)
```

specifies that STRING1 is to be given the value of STRING2 only if STRING1 has a null value when the rule is executed. If it is non-null, then the IDENT() procedure will signal failure, sending control to SKIP before the assignment takes place, so the value of STRING1 will remain unchanged.

Several arithmetic test procedures may be used in conjunction with one another to specify a range of acceptable values. The following rule for example, allows the printing of a record having from 2 to 10 characters only.

```
OUTPUT  =  GE(SIZE(REC),2)  LE(SIZE(REC),10)  REC
```

If either of the test procedures signals failure, no output is produced.

The following single statement employs two references to test procedures to specify that a transfer is to be taken to LOOP2 if the value of N is either 0 or 1; if N has neither value, then whatever value it has is increased by 1 and control flows by default to the next statement.

```
N  =  DIFFER(N,0)  DIFFER(N,1)  N + 1  :  F(LOOP2)
```

The desired condition here is that the value of N be either 0 or 1, so there is no need to differentiate the two cases. However, it is often necessary to know which part of the rule has signalled failure and to take different transfers accordingly. Consider, for instance, the problem of giving STRING, if it is null, the value of the next data record. The statement

```
STRING  =  IDENT(STRING)  TRIM(INPUT)  :  F(SKIP)
```

will send control to the statement labelled SKIP if STRING is non-null but also if an end-of-group record is encountered, making no differentiation between the two cases. Different transfers will usually be needed for these two situations, so in this case it will be necessary to express the process in two statements, each having a failure transfer, such as the following:

```
NEXT   =  TRIM(INPUT)          :  F(DONE)
STRING = IDENT(STRING) NEXT    :  F(SKIP)
```

The placement of a reference to a test procedure within the right side of an assignment rule implies that the value which the procedure returns is to be concatenated with any other right-side values before assignment occurs. All test procedures return null values, so the result of such concatenation is never visible; the null value concatenated with any other value leaves that value unchanged.

<u>Loops.</u> Any useful program will contain at least one (and usually many) loops which are to be executed repeatedly until some terminating condition is encountered. These loops may consist of any number of statements (they are typically longer than the one and two-statement loops which have been the only examples presented so far), and may overlap or be nested within one another. The terminating condition may be that an end-of-group record is read (as in the earlier

examples), that some other feature of the data is encountered, or that the loop has been entered a certain number of times. Every time a loop is entered it is necessary to perform some test, often with the use of a test procedure, to determine whether or not the terminating condition has been met; if it has, control is sent out of the loop to some other part of the program. If the test is accidentally omitted, or set up wrongly, then there may be no way to leave the loop and the set of statements of which it is composed will be executed repeatedly until the program is terminated by the computer's operating system. When this happens, the program is said to be in an "infinite" loop.

<u>Loops Controlled by Data Conditions.</u> The terminating condition for a loop may be that a record of a certain form is encountered in the data. If this record is an end-of-group mark, then the test for its existence can be made by simply providing a failure transfer on a statement in which the value of INPUT is needed. However, it is often useful to divide the data into "subgroups," each of which is terminated by a record having a special pattern of characters, such as one consisting of asterisks as the first six characters, followed by spaces. If each subgroup is to be processed separately, then a test must be made for this special signal each time a record is read, and a transfer taken accordingly.

IDENT() or DIFFER() can be used to make this kind of test. For example, the following program segment reads and prints all data records until one with asterisks as the first six characters and no other non-space characters is encountered; when that record is read, control is sent to STARS which may be the initial statement of another loop.

```
READ       RECORD  =  TRIM(INPUT)          :  F(ERROR)
           IDENT(RECORD,'******')          :  S(STARS)
           OUTPUT  =  RECORD               :   (READ)
```

Note that provision is made for the possibility that a record consisting of six initial asterisks will not be found in the group, i.e., that the program is processing the wrong data. This condition may be treated by transferring to a statement labelled ERROR when an end-of-group mark is read. Here an appropriate error message may be written and control sent either to END or to some other part of the program, depending on the sort of tasks which still remain to be done. If such an error exit were not provided there might be no indication from the program that anything was wrong, and it might attempt the processing of many groups of erroneous data. In any event, the program has entered an infinite loop since it is persistently seeking a terminating condition which will never be found.

<u>Loops Controlled by Counts.</u> Arithmetic test procedures are often used to control the number of times that a loop is to be entered before control is sent to some other part of a program; that is, the terminating condition for such a loop will be that it has been executed a given number of times. Using the EQ() procedure, for example, one may write a loop to print 5 data records, and then go on to the rest of the program. (If there are less than 5 records to be read, control is sent to ERROR where an appropriate error message can be printed.)

```
LOOP     OUTPUT  =  INPUT                  :  F(ERROR)
         COUNT  =  COUNT + 1
         EQ(COUNT,5)                       :  F(LOOP)
```

A similar loop may be written by using the LT() procedure and embedding it within the second assignment rule, as follows:

```
LOOP     OUTPUT  =  INPUT                  :  F(ERROR)
         COUNT = LT(COUNT,4) COUNT + 1 :  S(LOOP)
```

In this segment it has been necessary to use 4 as the test value rather than 5 since the procedure call is executed <u>before</u> the value of COUNT is incremented, rather than after as in the earlier example. In both segments, COUNT is assumed to have the null value when the segment is executed for the first time.

Information as to the number of times that something is to be done may be found on a data record or computed during the course of execution, rather than being written directly into the program text. For example, the following segment would cause the LOOP to be entered as many times as there were characters in each data record that it was processing.

```
READ     RECORD  =  TRIM(INPUT)            :  F(ENDDATA)
         N  =  SIZE(RECORD)
LOOP     N  =  NE(N,0)   N - 1             :  F(READ)
      [series of statements to process record]
                                           :   (LOOP)
```

Here the test has been placed at the beginning of the loop instead of at the end, and the counting has been done by subtraction rather than by addition. It might seem clearer and more intuitive to perform the process first and to test for the terminating condition afterwards (as in the two previous examples). For instance, the program text

```
READ    RECORD  =  TRIM(INPUT)           :  F(ENDDATA)
        N   =  SIZE(RECORD)
LOOP  [series of statements to process record]
        N   =  NE(N,1)  N - 1            :  S(LOOP)  F(READ)
```

might seem to be equivalent to the one given above, in the sense of always producing the same result. An examination of the case of a one-character record shows that the program appears to work properly. In this case it would perform the process once, find that N was equal to 1 and then leave the loop correctly by transferring to READ and reading in the next record.

The difference between the two programs becomes apparent when one attempts to process a record consisting solely of spaces which when trimmed becomes null. The program which tests before processing will handle records of size zero appropriately by failing the first time the loop is entered and returning immediately to read the next record. The program which processes first and then tests will perform the process once (erroneously) and then will test to see whether the value of N is equal to 1. Since it is zero, the value of N will be decreased by 1 to become -1, and control will be sent back into the loop so the process will be performed again. Henceforth the value of N will never equal 1, but a series of constantly decreasing negative numbers. The terminating condition will thus never be met and the program has entered an infinite loop.

4A. PATTERN MATCHING

The process of searching a string of characters to determine whether or not it contains one of a specified set of strings is called pattern matching. The pattern being sought may be something very particular, such as a certain character or a certain number of characters, or it may be something much more general , such as one of a choice of characters or all characters preceding one of a choice of characters. Like calls to test procedures, pattern matches either succeed or fail, causing the rules in which they occur to succeed or fail as well. Thus pattern matching may be used to direct the flow of control.

<u>The Pattern-Matching Rule.</u> The pattern-matching rule consists of two main parts: the string reference, whose value is to be searched, and the pattern. These two parts must be separated in the program text by one or more blanks. The very simple pattern-matching statement

```
        VOWELS  'E'                         :  S(YES)
```

specifies that the current value of VOWELS is to be searched for an instance of the character E, and that a transfer is to be taken to the statement labelled YES if the search is successful. If the search fails, then control will flow by default to the next statement of the program. Whether the search succeeds or fails, the value of VOWELS is in no way affected.

The pattern part may be in the form of a variable, rather than a literal, and may have a value consisting of more than one character. For example, the sequence

```
        PAT   =  'IOU'
        VOWELS  PAT                         :  S(YES)
```

specifies a search through the value of VOWELS for the three-character string IOU. This pattern match will succeed (if VOWELS has the value AEIOU) with the third, fourth, and fifth characters of the string reference being matched, and control will be sent to YES.

The search for the pattern always begins with the first character of the string reference and continues through the rest of the string from left to right until either a match is found or all characters have been tested. Note that if the first statement above had read

```
        PAT  =  'OUI'
```

the search would have failed. The characters OUI are indeed present within the string reference, but not in the indicated order.

The string reference part of a pattern-matching rule may be any expression which gives a string when evaluated. Thus executing the statement

```
        TRIM(TEXT)  '¤THE¤'               :  S(YES)
```

will cause the expression TRIM(TEXT) to be evaluated, and its value to be searched for an instance of the word THE, surrounded by spaces. Similarly, the use of the variable INPUT within the string reference will cause it to acquire the value of the next data record, since this value will be needed for the execution of the statement. A statement of the form

```
        TRIM(INPUT)  '¤THE¤'              :  S(YES)
```

however, is not likely to be useful since (1) the value of INPUT has not been assigned to another variable and hence will be lost, and (2) no distinction is made between failure of INPUT and failure of the pattern match.

<u>The Replacement Rule.</u> The replacement rule specifies a pattern which is to be sought in the string reference, and also a replacement for that part of the string which is matched by the pattern if the search is successful. For example, the replacement statement

```
        WORD  'A'  =  'Y'                 :  S(FOUNDA)
```

specifies that the character A is to be sought within the value of WORD and that the first A which is found, if any, is to be replaced by a Y. This new string, with Y in place of A, is stored within the memory and assigned to the variable WORD; the old value of WORD is lost.

Note that the search succeeds, replacement occurs, and control is sent to the go-to part of the statement as soon as the first (leftmost) instance of the pattern is found, so successive instances of the pattern remain unfound and unaltered. In order to change, for example, all A's within a string reference to Y's, one would write a loop of the form

```
SELF    WORD   'A'  =  'Y'                :  S(SELF)
```

When this rule failed, any A's which had been within the original value of WORD would all have been changed to Y's. If WORD referred to the value SASSAFRAS when the loop was first entered, its new value would be the string SYSSYFRYS.

The replacement for a matched substring may be shorter or longer than the string it replaces. Thus one may write a rule to replace a double vowel by a single one, as in

```
          WORD    'EE'  =  'E'
```

or a single vowel by a double one, as in

```
          WORD    'E'  =  'EE'
```

While it is perfectly safe to write the first of these replacement statements in a loop, so that all double (or triple, etc.) E's are reduced to a single E, execution of the statement

```
SELF      WORD    'E'  =  'EE'            :  S(SELF)
```

to make all single E's into double ones will send the program into an infinite loop if the value of WORD contains an E. Care must always be taken when writing replacement statements in a loop to insure that the pattern is not contained within its replacement, unless some terminating condition other than pattern match failure is used.

Deletion of a matched pattern may be accomplished by providing a null value to the right of the assignment sign. Thus one may delete all E's from a string reference by executing a statement of the form

```
DELETE    WORD    'E'  =  NULL            :  S(DELETE)
```

which will fail only when no E's remain within the value of WORD.

The replacement rule, which is syntactically a combination of a pattern-matching and an assignment rule, is the last of the four types of rules in the Snobol language. If the rule part of a statement is non-null, it must call for either an assignment, an evaluation, a pattern match, or a replacement.

The Alternation Operator. The alternation operator, a binary operator designated by the symbol | , is used to specify alternatives within a pattern. The pattern-matching statement

```
        WORD   'A'  |  'E'                 :  S(YES)
```

specifies that the value of WORD is to be searched for either an A or an E, and if either is found a transfer is to be taken to YES.

More than one alternation operator may be used within a pattern, as in the statement

```
        WORD  'A' | 'E' | 'I' | 'O' | 'U'  :  S(YES)
```

which will succeed if the value of WORD contains any of the five vowels. The search for a match proceeds as follows: the first character of WORD is checked successively for being A, E, I, O, or U; if it is none of these the second character is checked beginning with the A alternative, and so on. As soon as any one of the alternatives is found, transfer is made to YES. The pattern matching fails only when all characters of WORD have been examined and no alternative of the pattern has been found.

The alternatives may consist of any number of characters, not just a single character as in the example above. One may search a line to determine whether or not it contains one of a number of words, where a word is defined as a sequence of characters surrounded by spaces, by employing a statement of the form

```
        LINE '□A□' | '□' WORD1 '□' | '□' WORD2 '□' : S(YES)
```

The values of WORD1 and WORD2 may be strings of any length. An alternative way of writing this pattern is used in the statement

```
        LINE   '□'  ('A' | WORD1 | WORD2)  '□'     : S(YES)
```

Here, parentheses are necessary since the concatenation operator takes precedence over the alternation operator; if the parentheses were missing, the statement would be equivalent to

```
        LINE   '□A' | WORD1 | WORD2 '□'            : S(YES)
```

which is not what was intended.

<u>The Pattern Procedures ANY() and NOTANY().</u> Snobol has a number of predefined procedures for use solely in contructing patterns. The pattern procedures ANY() and NOTANY() provide an efficient way of expressing alternation, where the alternatives are single characters only. The

pattern-matching statement

```
         WORD  'A' | 'E' | 'I' | 'O' | 'U'  :  S(YES)
```

which employs four instances of the alternation operator may be written instead as

```
         WORD  ANY('AEIOU')              :  S(YES)
```
or
```
         WORD  ANY(VOWELS)               :  S(YES)
```
or
```
         WORD  ANY(TRIM(INPUT))          :  S(YES)
```

(if both VOWELS and TRIM(INPUT) have the value AEIOU). ANY() accepts for its single argument any expression whose value is a string, and returns as its value a pattern which will match any single character of that string. The pattern returned by ANY() contains only a single test for each character of the argument string, no matter how many instances of that character the string contains. That is, the pattern returned by ANY('SAGAS') is equivalent to that of 'S' | 'A' | 'G' .

The companion procedure to ANY() is NOTANY() which returns a pattern to match any single character <u>not</u> represented in its argument. Thus

```
         WORD  NOTANY('AEIOU')           :  S(YES)
```

will match the first character within the value of WORD which is not a vowel. This match will succeed if any character of the complete character set, except A, E, I, O, or U, is found.

It is always better to use ANY() or NOTANY() where single character alternatives are involved, but it will be necessary to use the alternation operator for alternatives of more than one character. Both methods of expressing alternation may be used together as in the statement

```
         WORD  'YW' | 'YI' | ANY('AEIOU')  :  S(GOOD)
```

The alternation operator and pattern procedures may be used within replacement rules as well as within pattern-matching rules. For example, the replacement rule

```
         WORD ANY('AEIOU')  =  'X'
```

specifics that the first vowel within the value of WORD is to be replaced by an X; the rule

```
        WORD  NOTANY('0123456789')  =  NULL
```

specifies that the first non-digit is to be deleted. Either rule may be written in a loop to specify that all vowels are to be replaced by X's

```
LOOP1   WORD  ANY('AEIOU')  =  'X'    :  S(LOOP1)
```

or that all non-digits are to be deleted

```
LOOP2   WORD NOTANY('0123456789') = NULL  :  S(LOOP2)
```

<u>The Conditional Assignment Operator.</u> It is often important when using a pattern which will match any one of a number of strings to preserve the information as to exactly what has been matched in the search. This may be done by assigning the matched substring as the value of a variable with the conditional assignment operator, a binary operator whose symbol is a period. The pattern-matching statement

```
        WORD ('AW' | 'AY' | ANY('AEIOU')) . SAVE  :  F(NO)
```

specifies that the value of WORD is to be searched for the alternatives, and that the part of the string reference which satisfies the pattern is to be assigned to the variable SAVE. If the value of WORD does not contain any of these alternatives, then the match fails and no assignment takes place, i.e., the value of SAVE remains unchanged.

(Note that these particular two-character alternatives must be expressed before the one-character alternatives; once an A is found the rule succeeds, so a search for AY or AW would never be undertaken if they were not the first alternatives to be tried.)

More than one conditional assignment operator may be used to assign the same value to more than one variable. The statement

```
        WORD ANY('AEIOU') . SAVE1 . SAVE2 . SAVE3  :  F(NO)
```

assigns the first vowel within the value of WORD to the variables SAVE1, SAVE2, and SAVE3.

If the variable OUTPUT is used, as in

```
        LINE  (WORD1 | WORD2 | WORD3) . OUTPUT
```

the successful match will be printed. The use of parentheses is necessary here since the conditional assignment operator associates itself with the single pattern element immediately to its left; if the parentheses were missing, OUTPUT would be assigned a value only if the value of WORD3 was the pattern alternative which caused the rule to succeed. (If that is what is intended, of course, then the parentheses should be omitted.)

The conditional assignment operator is useful within replacement rules in which the matched pattern is to form part of the replacement. If the first vowel found is to be reduplicated, one may use a statement of the form

```
WORD ANY('AEIOU') . SAVE = SAVE SAVE   :   F(NOVOWEL)
```

since the value assigned to SAVE is immediately available for use on the right side of the rule. If the pattern fails, control is sent directly to the go-to part of the statement, so no assignment can occur, either to SAVE or to WORD.

<u>Concatenation of Patterns.</u> The concatenation operator can be used with operands which are patterns, as well as with strings. For example, in the statement

```
WORD  ANY('AEIOU')  'Y' = 'Y'  :  F(NOVOWELY)
```

the operands of the concatenation operator are the pattern values returned by a call to the ANY() procedure and the string Y. The result is a pattern which will match any vowel which is followed by a Y; if this pattern is found it is to be replaced by a Y alone (i.e., the vowel is to be deleted). If instead the Y were to be deleted, a statement of the form

```
WORD ANY('AEIOU') . SAVE 'Y' = SAVE   :  F(VOWELY)
```

could be used. Here only a part of the matched pattern (the first vowel directly preceding a Y) is to be assigned to the variable named SAVE. Note, however, that the entire pattern must be found before such assignment can occur.

It is often useful to assign the different matched parts of a string reference to different variables. For example, a pattern to search for clusters of three consonants, and to assign each consonant to a different variable, is employed in the rule

```
WORD   ANY(C) . C1  ANY(C) . C2  ANY(C) . C3
```

(It is assumed here that the value of C is a string of consonants.) The pattern in this rule is the concatenation of three pattern elements, each of which consists of a reference to ANY() and a conditional assignment. The three-consonant string may be assigned to the variable CCC as well, by placing the entire pattern within parentheses and using one more conditional assignment operator, as follows:

```
WORD  (ANY(C) . C1 ANY(C) . C2 ANY(C) . C3) . CCC
```

None of the variables will acquire a new value unless the entire pattern is successfully matched.

<u>The Immediate Assignment Operator.</u> The immediate assignment operator is a binary operator whose symbol is a dollar sign ($). It is very similar to the conditional assignment operator except that it causes the immediate assignment of any matched substring to a variable, whether the remaining elements of the pattern are matched successfully or not. Thus if the rule above were rewritten as

```
WORD (ANY(C) $ C1 ANY(C) $ C2 ANY(C) . C3) . CCC
```

then C1 and C2 would acquire new values each time partial matches occurred, but C3 and CCC would acquire new values only when a substring of three contiguous consonants was found. For example, if WORD had the value ADIEU then C1 would acquire the value D when the match was attempted, while the rest of the variables remained unchanged; if WORD had the value CHATEAU then C1 would acquire the successive values C, H, and T, and C2 would acquire the value H, as repeated (but unsuccessful) attempts were made to find the pattern. Thus the immediate assignment operator may be useful in determining how much of a pattern was successfully matched before failure occurred.

Both the conditional and immediate assignment operators may be applied to the same pattern element, as in the rule

```
WORD  ANY(VOWELS) $ SAVE1 . SAVE2 'T'
```

which specifies a search for any vowel which is followed directly by a T. (The order in which the immediate and conditional assignment operators occur is immaterial.) If the pattern match succeeds, then both SAVE1 and SAVE2 will refer to the same value, that of the first vowel encountered which occurred directly before a T. If WORD contained one or more vowels, but not one occurring before a T, then the match will fail and the value of SAVE2 will be unchanged, but SAVE1 would acquire as successive values all vowels within the value of WORD which were encountered in the attempts to find the pattern.

The variable OUTPUT may be used in conjunction with the immediate assignment operator to produce a printed trace of the progress of the pattern-matching operation. For example, if the variable OUTPUT were written in place of SAVE1 above, producing the rule

```
        WORD  ANY(VOWELS) $ OUTPUT . SAVE2 'T'
```

and the value of WORDS was the string ECCLESIASTICAL, then the following output would be produced:

```
E
E
I
A
I
A
```

When a transfer was taken to the next statement, the value of OUTPUT would be A and the value of SAVE2 would not have been changed, since the pattern match did not succeed.

<u>The Pattern Procedures SPAN() and BREAK().</u> SPAN() and BREAK() are procedures which match not just a single character but a string of characters of indefinite length. SPAN() returns a pattern which matches a string composed solely of the characters specified within its argument. For example, a string consisting of one or more vowels may be specified by the pattern

```
        SPAN('AEIOU')
```

BREAK() returns a pattern which matches a string composed of any characters <u>except</u> those specified in its argument. Thus a string consisting of anything but vowels may be specified by the pattern

```
        BREAK('AEIOU')
```

Both SPAN() and BREAK() must find a character from their argument strings in order to succeed. SPAN() will match that character along with any other acceptable characters which are contiguous; BREAK() will match everything up to such a character, leaving the "break character" itself unmatched.

Note that the pattern returned by BREAK() may match the null value, as in

```
        WORD  =  'IDLE'
        WORD  BREAK('AEIOU') . SAVE
```

Here SAVE will be assigned the null value since BREAK() matches all characters preceding the first vowel, or in this case no characters. SPAN() can never match the null value since it must match at least one of the characters of its argument.

SPAN() and BREAK() are often used together to break data into significant units, such as words. If a word is defined as a string of characters terminated by any number of spaces, periods, or commas, then the following program segment can be used to assign to the variable WORD each new word of the data.

```
READ     LINE  =  TRIM(INPUT)  '¤'        :  F(DONE)
LOOP     LINE BREAK('¤.,') . WORD SPAN('¤.,') = NULL
+                                          :  F(READ)
      [sequence of statements to process WORD]
                                           :   (LOOP)
```

In the replacement statement labelled LOOP, BREAK('¤.,') matches all characters until a space, period, or comma is encountered. The sequence of characters which have been matched is assigned to the variable WORD. SPAN('¤.,') will then match the character which caused BREAK('¤.,') to succeed, and any other spaces, periods, or commas which may be contiguous. This entire pattern is then replaced by the null value (removed from LINE), the value of WORD is processed in some way, and control sent back into the loop again. The replacement rule fails only when no more words remain to be processed and a new value for LINE is read in. Note that a space has been concatenated to the trimmed value of each data record to insure that BREAK('¤.,') will be able to find a "break character" at the end of the last word, and SPAN('¤.,') will have at least one character to match.

<u>The Pattern Procedure LEN().</u> The pattern procedure LEN() accepts any non-negative integer argument, and returns a pattern to match as many characters as its argument specifies. Thus LEN() matches strings of predictable length but unpre-dictable content, while BREAK() and SPAN() match strings of predictable content but unpredictable length.

LEN() is useful between two pattern elements to specify the exact number of characters which must lie between them for the match to succeed. Thus the search for four-character strings within parentheses might be specified by the statement

```
LINE '(' LEN(4) . INSIDE ')' : F(OUT)
```

Note that the strings matched by the three concatenated pattern elements must be contiguous for the match to succeed. Thus the above rule does not mean "at least four characters between parentheses" but "exactly four." If this rule is successful, the first string of four characters found between parentheses will be assigned to the variable INSIDE.

LEN() is often used at the beginning of patterns to match an initial field of the data, such as an identification number. The statement

```
LINE LEN(10) . IDNUMBER LEN(40) . DATA : F(SHORT)
```

assigns the first 10 characters of LINE to the variable IDNUMBER, and the next 40 characters to the variable DATA. The rule will fail only if LINE contains less than 50 characters.

Statements of the form

```
LINE LEN(10) . IDNUMBER 'A' : S(ALINE)
```

are often erroneously used to specify a search for lines with A as the eleventh character. While it is true that all such lines will be found by the above rule, many other lines may be found as well. The rule will succeed if a string of 10 characters preceding an A can be found anywhere within the value of LINE, not necessarily in initial position.

The ANCHOR() Procedure. The ANCHOR() procedure may be used to "anchor" all searches so that they succeed only in initial position. In anchored mode, if a pattern does not match beginning with the first character of the string reference, failure is recorded immediately and no further pattern searching occurs.

The normal, unanchored, mode of pattern matching can be changed to anchored mode by executing an evaluation rule of the form

```
ANCHOR('ON')
```

or

```
ANCHOR('XXX')
```

or

```
ANCHOR(VOWELS)
```

or any other rule in which the ANCHOR() procedure is called with a non-null argument. Executing the sequence

```
ANCHOR('ANCHORITE')
LINE  LEN(10) . IDNUMBER 'A'  :  S(ALINE)
```

would cause a transfer to ALINE only when the eleventh character of LINE was indeed an A.

The anchored mode remains in effect until another rule is executed in which the ANCHOR() procedure is called with an argument having a null value, such as

```
ANCHOR()
```

or

```
ANCHOR(NULL)
```

The original unanchored mode of pattern-matching is then restored.

<u>The Pattern Procedures TAB() and RTAB().</u> The pattern procedures TAB() and RTAB() specify pattern matching not in terms of character content or of length, but in terms of position within the string reference. Both TAB() and RTAB() accept a single argument which must be a non-negative integer and return a pattern to match all the characters up to that position within the string reference, matching as always from the left. The difference between TAB() and RTAB() is that they use opposite conventions for numbering the string positions (and thus for interpreting their arguments): TAB() works in terms of numbers counted from the left, RTAB() in terms of numbers counted from the right, as shown in the following charts:

For TAB(),

```
       character:    1   3     6 7
                     |   |     | |
 string_position:   0|1  |3    |6|7
                    |||  ||    ||||
                     C A M Y L O T
```

For RTAB(),

```
       character:    7 6     3   1
                     | |     |   |
 string_position:   7|6|    3|  1|0
                    ||||    ||  |||
                     C A M Y L O T
```

Notice that although there is no zero-th character, there is a zero-th string position — just before the first character or just after the last one, depending on whether TAB() or RTAB() is being used. This prevents confusion when thinking about characters in terms of their string positions: TAB(2), "everything up to string position 2," matches the first two characters; RTAB(1), "everything up to string position 1 counting from the right," matches all the characters but one. Although the argument of RTAB() is an integer to be used in counting from the right, this does *not* imply that pattern-matching is done from the right; pattern-matching always proceeds from the left.

TAB() and RTAB() may be used for breaking up strings into fixed fields; the rule

```
LINE  TAB(15) . ID  TAB(70) . TEXT
```

assigns the first 15 characters of LINE to ID, and the next 55 characters (those remaining up to string position 70) to TEXT. This is exactly equivalent to the rule

```
LINE  LEN(15) . ID  LEN(55) . TEXT
```

If the first field were of varying length, terminated by a space, then

```
LINE  BREAK('□') . ID  '□'  TAB(70) . TEXT
```

would assign everything up to the first space to ID, and all characters after the space but before string position 70 to TEXT. Note that this is *not* equivalent to

```
LINE  BREAK('□') . ID '□' LEN(70) . TEXT
```

in which all characters up to the first space are assigned to the variable ID (as before) but a full 70 characters following the space are assigned to the variable TEXT. TAB() may match strings of varying length ending at a definite string position, while LEN() will always match a definite number of characters ending at varying string positions.

RTAB() can be used like TAB() for patterns in which the string position terminating the match is better expressed as a count from the right rather than from the left. RTAB(0) is particularly useful; it will always match everything from the current position in a pattern search up to the end of the string — the "remainder" of the string after any other pattern elements have been matched.

Both TAB() and RTAB() can match the null value; but if either attempts to match up to a string position to the left of one which has already been matched by a preceding pattern element, or a string position which does not exist (because the string is too short), the pattern match will fail.

The Pattern Procedures POS() and RPOS(). The pattern procedures POS() and RPOS() return patterns which match no characters at all (the null value); they match only the single string positions specified by their single non-negative integer arguments. POS() uses the numbering system of TAB(), RPOS() of RTAB(). Their use is to restrict successful matches by other pattern elements to certain positions in string references; this provides a more flexible form of "anchoring."

A pattern which begins with POS(0) is anchored in the usual way. The rule

```
LINE  POS(0) '******'
```

will succeed only if the value of LINE contains asterisks as its first six characters. (The advantage over turning on the ANCHOR() procedure is that the restriction applies to this single rule only.) Similarly, the rule

```
LINE  POS(7) '******'
```

will succeed only if the value of LINE contains asterisks as characters 8 through 13.

RPOS() permits the same kind of anchoring, counting from the right; the rule

```
LINE  '******'  RPOS(0)
```

will match only if the value of LINE ends with six asterisks, and

```
LINE  POS(0) '******'  RPOS(0)
```

will succeed only if the value of LINE is precisely a six-character string of asterisks. That is, the above pattern-matching rule is equivalent to the evaluation rule

```
IDENT(LINE,'******')
```

The Pattern Procedure ARBNO(). ARBNO() is the only pattern procedure which accepts a pattern as its argument. It returns a pattern which will match zero or more occurrences of the pattern given in its single argument. Note that matching zero occurrences is the same as matching the null value; since this is always the first choice for the ARBNO() procedure, a call to it always succeeds. ARBNO() will match as many occurrences of the specified pattern as will cause the remainder of the pattern to succeed.

A string is a simple form of a pattern, so the argument of ARBNO() may be a single character or characters. A pattern to match zero or more A's may be specified as

```
      ARBNO('A')
```

This differs from

```
      SPAN('A')
```

in that the SPAN() procedure must always match at least one character, so the pattern which is the value of SPAN('A') matches <u>one</u> or more A's instead.

A pattern which will match any number of characters, including none, enclosed within parentheses (rather than exactly 4, or some other number) can be specified with the use of ARBNO() as follows:

```
      LINE  '(' ARBNO(LEN(1)) . INSIDE ')'  :  F(NOPAREN)
```

This pattern will match strings of the form

```
      ()
      (1)
      (AB)
      (XXX)
```

The null value or the characters within the parentheses will be assigned to the variable INSIDE.

A more complicated illustration of the use of ARBNO() is provided by a consideration of the following set of sentences:

The dog ran.
The old dog ran.
The old, gray dog ran.
The old, gray, barking dog ran.

The similarity among these sentences may be characterized in terms of some pattern which would succeed when applied to any of them. Such a pattern may be written with the use of

ARBNO() as follows:

'THE¤' ARBNO(BREAK('¤,') LEN(1)) 'DOG¤RAN.'

When this pattern is applied to the first sentence, the ARBNO() procedure matches zero instances of its argument, or the null value, since the literal strings within the pattern account for the entire sentence. In the second sentence, ARBNO() matches one instance of its pattern, the string OLD¤. In the third sentence, ARBNO() matches three instances of its pattern, the string OLD,¤GRAY¤. This is three instances since BREAK() first matches everything up to the comma, then up to the space following the comma, then up to the space following GRAY. In the last sentence, ARBNO() matches five instances of its pattern, the string OLD,¤GRAY,¤BARKING¤. The pattern matching in the last sentence occurs as follows:

(1) the opening literal matches to begin with and ARBNO() matches no instances of its pattern (or the null value); but then the closing literal cannot be matched, so an instance of the ARBNO() pattern is sought with

(2) BREAK() matching everything up to the comma (the string OLD), and LEN() matching the comma; when the final literal cannot be matched, successive instances of the ARBNO() pattern are tried with

(3) BREAK() matching everything up to the blank (the null value) and LEN() matching the blank, then

(4) BREAK() matching everything up to the next comma (the string GRAY) and LEN() matching the comma, then

(5) BREAK() matching everything up to the following blank (again the null value) while LEN() matches the blank, and finally

(6) BREAK() matching everything up to the next blank (the string BARKING) and LEN() matching the blank. At this point the final literal can be matched and the entire pattern matching is completed.

These successive attempts by ARBNO() to match the number of instances of its argument which will cause the remainder of the pattern to succeed could be observed by using the immediate assignment operator in conjunction with the variable OUTPUT as described earlier.

<u>Assigning Patterns to Variables.</u> Patterns may be assigned as the values of variables just as strings are assigned as the values of variables. This may be done with an assignment rule of the usual form, such as

```
        PAT  =  'IOU'
```

or

```
        ID.PAT  =  LEN(1) . IDNUMBER  LEN(40) . DATA
```

or

```
        DOG = 'THE¤' ARBNO(BREAK('¤,') LEN(1)) 'DOG¤RAN.'
```

The variable which refers to the pattern, rather than the pattern itself, may then be used within the pattern part of a rule as in

```
        VOWELS  PAT                       :  S(YES)
```

or

```
        LINE  ID.PAT                      :  F(SHORT)
```

or

```
        DOGLINE  DOG                      :  F(NODOG)
```

When these statements are executed, the current values of PAT, ID.PAT, and DOG are obtained; thus the pattern matching and the conditional assignment are performed exactly as if the patterns themselves were expressed.

The value of the variable PAT is of datatype String, but it may be used as the pattern part of a pattern-matching rule, as indicated at the very beginning of this chapter, since a string is a trivial form of a pattern. The values of ID.PAT and DOG are of datatype Pattern, since they are concatenations of values of calls to procedures which return patterns. Any expression containing a reference to a pattern procedure, an alternation operator, a conditional or immediate assignment operator, or a deferred evaluation operator (described below), has a value of datatype Pattern. The values of such expressions cannot be assigned to the special variable OUTPUT, since only strings can be printed. (Ways of printing the value of an expression of datatype Pattern are indicated in Appendix A, section II.B, s.v. "PROTOTYPE()".) The variables ID.PAT and DOG are of course in no way restricted to having only Patterns as their values, but may be assigned values of any datatype in other parts of the program.

If a pattern occurs within a rule which is to be executed more than once, or if the same pattern occurs in more than one rule, a considerable increase in program efficiency can be obtained by assigning the pattern as the value of a variable. The use of a variable within the rule makes it unnecessary to construct the pattern every time the rule is executed.

When a pattern is assigned to a variable, as in the rule

```
        ALTPAT  =  X | Y
```

any variables occurring within the pattern (X and Y above) are evaluated when the assignment rule is executed. Thus if X had as its value the string A and Y the string B, the value of ALTPAT after the above rule had been executed would be equivalent to 'A' | 'B' .

There are often applications, however, in which one wants the variables of the pattern to be evaluated only when the pattern is used in a pattern-matching rule, not when the assignment occurs. For example, a loop to search the value of WORD for one of two substrings, each to be read from the input file, may be written as follows:

```
LOOP1   X  =  TRIM(INPUT)                  :  F(DONE)
        Y  =  TRIM(INPUT)                  :  F(ERROR)
        WORD  X | Y                        :  S(FOUND)   F(LOOP1)
```

Since the efficiency of the program can be increased by using a variable which refers to a pattern, rather than the pattern itself, one would like to be able to write the loop as

```
        ALTPAT  =  X | Y
LOOP2   X  =  TRIM(INPUT)                  :  F(DONE)
        Y  =  TRIM(INPUT)                  :  F(ERROR)
        WORD  ALTPAT                       :  S(FOUND)   F(LOOP2)
```

If this is done, however, the loop will not have the same meaning as before. The new values of X and Y which are acquired from the input file on each iteration of the loop will not affect the value of ALTPAT; rather its value will remain unchanged at 'A' | 'B' (if A and B were the values of X and Y when the assignment occurred).

<u>The Deferred Evaluation Operator.</u> The deferred evaluation operator, a unary operator whose symbol is an asterisk (*), may be used within patterns to take care of the above situation. It may be written directly before the name of a variable to indicate that its evaluation is to be deferred until its value is needed during a pattern-matching operation. For instance, the assignment rule

```
        ALTPAT  =  *X | *Y
```

may be used to indicate that both X and Y are variables which are to be re-evaluated each time a pattern-matching rule is executed in which ALTPAT is used within the pattern part. Thus the sequence

```
        ALTPAT  =  *X | *Y
LOOP3   X  =  TRIM(INPUT)              :  F(DONE)
        Y  =  TRIM(INPUT)              :  F(ERROR)
        WORD  ALTPAT                   :  S(FOUND)   F(LOOP3)
```

will produce the same results as the LOOP1 example above, but more efficiently.

The unary * operator is also useful in patterns in which the value of one pattern element is dependent on the successful match of an earlier element of the same pattern. Consider, for example, the problem of searching a word to determine whether or not it contains two identical contiguous vowels. This pattern may be expressed using the * operator as

```
        VOW2PAT  =  ANY(VOWELS) $ V  *V
```

When this pattern is used, as in the statement

```
        WORD  VOW2PAT                  :  S(YES)
```

it specifies a search through the value of WORD for any of the five vowels, immediate assignment of the vowel found to the variable V, and then a search of the next character for another instance of that same vowel.

A more general pattern in the same vein is one which searches for two identical contiguous characters. This may be expressed as

```
        CHARPAT  =  LEN(1) $ CHAR  *CHAR
```

and works as described above. Without the use of deferred evaluation, these patterns would be cumbersome to define.

The unary * operator may be used only before names of variables, not before references to pattern procedures. An expression composed of a deferred evaluation operator and a variable name is of datatype Pattern and so may be used only where a pattern value is appropriate; hence such an expression may not be used as the argument of any of the pattern procedures except ARBNO(). The loop

```
        ARBPAT  =  'S'  ARBNO(*X) . SAVE  'S'
LOOP4   X  =  TRIM(INPUT)              :  F(DONE)
        WORD  ARBPAT                   :  S(FOUND)  F(LOOP4)
```

specifies a search through WORD for zero or more instances of whatever string is specified on the next data record, bounded by an S on either side, and the assignment of the substring matched by ARBNO() to the variable SAVE. If the search fails, another data record is read, causing a different pattern to be sought.

<u>The Special Pattern Variables ARB and REM.</u> There are six variables which have predefined patterns as their values, assigned by the Snobol system; these are the only six variables in Snobol which do not have the null value when execution of a program begins. The values of these variables may be changed in a program by assigning them new values in the usual way, but then of course the predefined values are lost. The six special pattern variables are ARB, REM, BAL, FAIL, FENCE, and ABORT. Only ARB and REM will be discussed here. (The remaining four pattern variables are described in Appendix B.)

The variable ARB has as its predefined value a pattern equivalent to ARBNO(LEN(1)) — that most arbitrary pattern which will match the null value or any string of characters. ARB, like ARBNO(LEN(1)), matches the longest string of characters left for it by surrounding pattern elements; thus the pattern to match any parenthesized string could have been written as

```
        LINE  '(' ARB . INSIDE ')'     :  F(NOPAREN)
```

Execution of this statement would cause the variable INSIDE to be assigned the zero or more characters occurring between a pair of parentheses.

The variable REM has as its predefined value a pattern which will match "all the remaining (none-or-more) characters." Another pattern equivalent to this is RTAB(0). For example, a statement to match all characters after the sixth may be written as

```
        LINE  LEN(6) REM . A6          :  F(NOTSIX)
```

Execution of this statement will cause LEN(6) to match the first six characters in LINE and will cause all remaining characters to be assigned to the variable A6. If the value of LINE is exactly six characters long, the pattern match will succeed and the variable A6 will be assigned the null value. If the value of LINE is less than six characters long the pattern match will fail, A6 will not acquire a new value and control will be sent to the statement labelled NOTSIX.

Since the predefined pattern values of both ARB and REM are equivalent to patterns which may easily be written in other ways, ARB and REM may be regarded merely as convenient predefined abbreviations for longer pattern specifications.

<u>A Program to Illustrate Pattern-Matching.</u> The program text provided below reads an indefinitely long text which has line numbers in the first six positions of each data record, and words occurring in free form, but never broken across records, in the remaining positions. A word is defined as a string of characters followed by a space or a punctuation character. Any number of spaces and/or punctuation characters may occur between words (and before the first word on a card). The program looks for words within the text which begin and end with the same character (one letter words excluded). If such words are found, they are printed following the line number of the record in which they occurred. Thus the two records

```
000001    EFFICIENCY IS IMPORTANT BUT
000002    ELEGANCE IS TO BE DESIRED
```

would produce the output

```
000002    ELEGANCE    DESIRED
```

since the first line contains no words which begin and end with the same character, but the second line contains two. All patterns are assigned to variables for the sake of efficiency.

```
*   PROGRAM TO FIND AND PRINT ALL WORDS THAT
*     BEGIN AND END WITH THE SAME CHARACTERS
*
*   SET UP THE PATTERNS NEEDED FOR THE PROGRAM
*
         PUNC  =  '□.,:;'
         WORD.PAT  =  BREAK(PUNC) . WORD  SPAN(PUNC)
         ID.PAT  =  LEN(6) . ID  (SPAN(PUNC) | NULL)
         SAME.PAT  =  POS(0)  LEN(1) $ CH  RTAB(1)  *CH
*
*   READ THE NEXT RECORD OF THE DATA - APPEND A SPACE
GETLINE  LINE  =  TRIM(INPUT)  '□'      :  F(END)
*
*   REMOVE ID NUMBER - IGNORE RECORDS SHORTER THAN 6 CHARS
         LINE  ID.PAT  =  NULL          :  F(GETLINE)
```

```
*  GET THE NEXT WORD - IF NO MORE WORDS, CONSIDER PRINTING
GETWORD LINE  WORD.PAT  =  NULL          :   F(PRINT)
*
*  SEE IF THIS WORD HAS SAME FIRST AND LAST CHARS -  IF NOT,
*     THEN GET THE NEXT WORD
        WORD  SAME.PAT                   :  F(GETWORD)
*
*  WORD TO BE PRINTED - APPEND IT TO THE OUTPUT LINE
        OUT  =  OUT  '□□□□'  WORD        :  (GETWORD)
*
*  PRINT VALUE OF OUT IF IT CONTAINS ANY WORDS
*    PRECEDE THE WORDS BY THE APPROPRIATE LINE NUMBER
PRINT   OUTPUT = DIFFER(OUT,NULL)  ID OUT  : F(GETLINE)
*
*  IF NECESSARY, ASSIGN OUT A NULL VALUE BEFORE PROCEEDING
        OUT  =  NULL                     :  (GETLINE)
END
```

5A. INDIRECT REFERENCING

The fact that a single variable may be used to refer to a number of different values during the course of program execution makes it possible to write a general rule which can have the effect of many specific ones. For example, the single rule

```
OUTPUT  =  WORD
```

specifies in general that the current value of the variable named WORD is to be printed, whatever that value may be. If the above rule is part of a loop in which WORD is being assigned a new value every time the loop is entered, then the rule sends different specific characters to the output file every time it is executed. Without this ability to express a process in general terms rather than in specific ones, no useful programs could be written.

The ability to generalize is further extended in Snobol by the use of indirect referencing. This operation allows one to specify a variable without writing its name into the program text; rather, one specifies a variable by writing an expression whose value is a variable. Just as WORD in the rule above may refer to a number of different values during the course of program execution, so this expression involving indirect referencing may refer to a number of different variables during the course of the program, each variable's value changing independently. In neither case do the specific values need to be known when the program text is written. Hence the use of indirect referencing allows another level of generality to be introduced.

<u>The Indirect Referencing Operator.</u> Indirect referencing is accomplished by means of the indirect referencing operator, a unary operator whose symbol is a dollar sign ($). This operator takes a single string-valued operand (or one of datatype Name as described in Chapter 7) and returns as its value the variable named by that string. In the simplest case, the operand is a literal as in the rule

```
OUTPUT  =  $'WORD'
```

which produces the same effect as

```
OUTPUT  =  WORD
```

Both will cause the current value of the variable WORD to be printed since the variable returned by the $ operator above is the one whose name is WORD. There is no advantage to using the $ operator in this way, since it is simpler to write WORD than to write $'WORD'.

However, there are many variables which cannot be referred to by writing their names in program texts since they consist of strings of characters which are not identifiers. As indicated in Chapter 2,

```
1RHYME      ..VOWELS      TEXT/3      P-V-C
```

are all the names of variables, but they are not valid representations of these variables within a program text. These variables may be represented with the use of the $ operator, since they are, respectively, the values of the expressions

```
$'1RHYME'    $'..VOWELS'    $'TEXT/3'    $'P-V-C'
```

Although these expressions are useful in a way that $'WORD' is not, they introduce no generality into the program since each specifies a single, fixed, variable.

Generality is introduced when the operand of the $ operator is some string-valued expression other then a literal. Thus the rule

```
OUTPUT  =  $WORD
```

can cause the values of different variables to be printed when it is executed at different times, since the variable whose value is to be printed depends on the current value of WORD. If the rules

```
WORD  =  'SASSAFRAS'
```

and

```
SASSAFRAS  =  'TREE'
```

have been executed, then execution of the rule

```
OUTPUT  =  $WORD
```

will cause the characters TREE to be printed. First WORD is evaluated to yield the string SASSAFRAS; then the $ operator returns the variable named by that string. Thus the effect is as though

```
OUTPUT  =  $'SASSAFRAS'
```

or, equivalently,

```
OUTPUT = SASSAFRAS
```

had been executed.

Similarly, the rule

```
$VOWEL = $VOWEL + 1
```

can cause the value of many different variables to be incremented by 1. If the value of VOWEL is the string A, then the rule is equivalent to

```
$'A' = $'A' + 1
```

or

```
A = A + 1
```

but if the value of VOWEL is a different vowel, say E for example, then the rule is equivalent to

```
E = E + 1
```

instead. Thus executing the same rule at different times in the program may result in incrementing the value of different variables. A single rule of this form could be used to count how many of each vowel occurred in a text.

(Notice that a variable returned by the indirect referencing operator is treated in the execution of rules exactly like a variable whose name is written in the program text; variables occurring to the right of an assignment sign, or within a pattern or a string reference, must be evaluated when the rule in which they occur is executed.)

<u>The Operand of the Indirect Referencing Operator.</u> The operand of an indirect referencing operator may be an expression of any complexity; the only restriction is that this expression yield a non-null string (or a Name) when it is evaluated. Thus the operand of a $ operator may itself contain one or more $ operators (as in the expression $$CURRENT), as long as the variable returned by each inner $ operator refers to a value which is a string. These nested $ operators, like nested procedure calls, must be evaluated from the inside out since the variable returned by an inner $ is needed to form the operand of an outer $. For example, if the assignments

```
CURRENT = 'VOWEL'
```

and

```
VOWEL = 'A'
```

have been executed, then the rule

```
$$CURRENT = $$CURRENT + 1
```

is equivalent to

```
A = A + 1
```

The evaluation of the rule involving double indirect referencing proceeds as follows: first the value of CURRENT is determined, providing the string VOWEL as the operand of the inner $ operator and making the expression $$CURRENT equivalent to $$'VOWEL'; when the inner $ is applied to the string VOWEL the variable VOWEL is returned, making $$'VOWEL' equivalent to $VOWEL; the outer $ is then applied, giving $'A', in turn equivalent to A, as above. Examples of how multiple indirect referencing can be useful are provided by two program texts given at the end of this chapter.

Similarly, a reference to any procedure which returns a string as its value may be used within the operand. As a simple example, the rule

```
$SIZE(WORD) = $SIZE(WORD) + 1
```

could be used in a loop, analogously to the rule

```
$VOWEL = $VOWEL + 1
```

above, to count how many words of each length occurred in a text. If the current value of WORD at some point during execution is the nine-character string SASSAFRAS, then the above rule is equivalent to

```
$'9' = $'9' + 1
```

Thus the variable whose name is 1 would be assigned the count of the one-character words, the variable named 2 the count of the two-character words, etc. Although the names of these variables may not be written in the program text, the variables may be specified by means of indirect referencing, since the $ operator may be applied to any string of characters to return the variable named by that string.

The null value may not be used as the operand of the $ operator since the name of a variable must be at least one character long. It is a common mistake, however, to use as the operand of the $ operator a variable which at some time during the course of execution will have a null value. Such an error cannot occur in the example above, since there is no way for the operand to be null. If WORD has a null value, then SIZE(WORD) returns the integer zero as its value. Hence the count of all null values is referred to by the variable whose name is 0. (If WORD has a value which is not a string, then an execution-time error will result when the SIZE() procedure is called, before an attempt to apply the $ operator can be made.)

A Program to Produce a Character Count. As an example of the power of indirect referencing, consider this simple character-counting program, which prints out a table giving the number of times each letter occurred within a text.

```
*   PROGRAM TO MAKE A CHARACTER COUNT
*   SET UP CHARACTER-FINDING PATTERN
         CHAR.PAT = LEN(1) . CHAR
*
*   READ IN THE DATA
READ     LINE  =  TRIM(INPUT)           :  F(OUT)
*
*   FIND THE NEXT CHARACTER - ASSIGN IT TO THE VARIABLE CHAR
LOOP1    LINE  CHAR.PAT = NULL          :  F(READ)
*
*   ADD ONE TO THE COUNT FOR THAT CHARACTER
INC      $CHAR  =  $CHAR + 1            :   (LOOP1)
*
*   SPECIFY THE ALPHABET FOR RECOVERING COUNTS
OUT      ALPHA  =  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
*
*   GET THE NEXT LETTER WHOSE COUNT IS TO BE RECOVERED
*   ASSIGN IT TO THE VARIABLE CHAR
LOOP2    ALPHA CHAR.PAT = NULL          :  F(END)
*
*   IF LETTER DID NOT OCCUR, GIVE IT THE VALUE ZERO, NOT NULL
         $CHAR  =  IDENT($CHAR,NULL)  0
*
*   PRINT LETTER AND ITS COUNT
         OUTPUT  =  CHAR '□□□□' $CHAR  :   (LOOP2)
END
```

Output from this program would be a list of the form

```
A    129
B    58
C    32
```

and so on.

This program uses the pattern which is the value of CHAR.PAT to assign each successive character of the text to the variable CHAR; indirect referencing is then used to return the variable named by that character. Depending on which character has been found, the rule part of the statement labelled INC might be equivalent to

```
        A  =  A + 1
```

or

```
        B  =  B + 1
```

or

```
        $','  =  $',' + 1
```

or whatever.

When all the text has been read, printing of the counts begins. This is done with the use of the variable ALPHA, whose value is a string containing all the characters for which counts are to be printed, given in the desired order. (In this case, only letters have been chosen.) These letters, one by one, are again assigned to the variable CHAR (although any other variable would have done as well) by means of the CHAR.PAT pattern. Using indirect referencing, the variable named by the character is tested to determine whether or not it has a null value; if it is null, then that character was never encountered in the text and so the variable is given the value zero for output purposes. The output statement prints the value of CHAR (the character A the first time the output loop is entered) and the value of $CHAR (in this case the value of the variable A, or 129).

This scheme for specifying the printing permits the programmer to choose the order of the output — alphabetical order, rather than text order — and to be selective; the program causes counts to be stored for all characters (numbers, punctuation, spaces, etc.), but only the counts for the letters are recovered for printing.

<u>Concatenation within the Operand.</u> The concatenation operator is needed within the operand of the indirect referencing operator in applications in which variables having "successive" names are to be used. For example, execution of a loop of the form

```
NLOOP   N  =  N + 1
        OUTPUT  =  TRIM(INPUT)        :  F(ALLGONE)
        $('LIST' N)  =  OUTPUT        :  (NLOOP)
ALLGONE
```

will cause an entire group of data to be read, printed, and stored, with successive records being assigned as the values of the variables named LIST1, LIST2,..., $('LIST' N). When the loop terminates through failure of INPUT, the value of N is an integer one greater than the number of lines of data which have been read. Since these lines of data are now stored in the memory they may be processed in some way, for example subjected to pattern-matching and replacement, and eventually printed out again in an altered form. The following loop may be used to print out all the lines, reversing their line numbers in the output, so that the last record read in is numbered 1, the next-to-last numbered 2, etc., until the first record read in is numbered N-1.

```
        M  =  N
MLOOP   M  =  GT(M,1)  M - 1            :  F(DONE)
        OUTPUT  =  N - M  '□□□□' $('LIST' M)  :  (MLOOP)
DONE
```

In the above example, a single set of successively-named variables were being assigned values (those whose names all begin with the characters LIST). This process can be made more general if several sets of successively-named variables are assigned values by the same program segment. If, for example, a file contained intermixed records of various types, each type distinguished by the first character of the record, then the following segment of program text would cause each record to be assigned to the variable named by the concatenation of its first character (the type-code) and the number of records of that type encountered so far.

```
READ    RECORD = TRIM(INPUT)            :  F(DONE)
*
*  DETERMINE TYPE-CODE OF RECORD
        RECORD  LEN(1) . CODE           :  F(READ)
*
*  ADD ONE TO COUNT FOR THIS TYPE
        $CODE  =  $CODE + 1
*
*  STORE RECORD IN NEXT "SUCCESSIVE" VARIABLE OF ITS TYPE
        $(CODE $CODE)  =  RECORD        :    (READ)
DONE
```

The first record found beginning with an E would become the value of the variable named E1, for example, and the twenty-fifth record found beginning with a colon would become the value of the variable named :25. If the distinct type-codes are stored by the program as they are encountered, then the records have effectively been sorted in terms of their first characters, since the records of each type can now be found as the values of different sets of successively-named variables.

Variables having "successive" names are also useful in printing data in tabular format, where a varying number of spaces, or other characters such as dots or dashes, will be needed to make the data line up properly. The variable named 1□, for example, could be assigned the value of a single space, while the variable named 2□ would have the value of two spaces, etc. In general, variables can be given names which indicate their values, where the first part of the name indicates the number of instances of some character, and the second part indicates the character in question. Thus the variable named 52X would have as its value a string of 52 X's.

The short segment of program text below causes such variables to be assigned appropriate values. The value of MAX is the largest number to be used as the first part of any name and is the maximum length of any string to be assigned as value; the value of CHAR is the particular character to be used as the second part of each name and is the character of which all string values are to be composed.

```
FORMLOOP N  =  LT(N,MAX)   N + 1          :  F(DONE)
         $(N CHAR) = $(N - 1 CHAR) CHAR   :    (FORMLOOP)
DONE
```

If MAX has the value 10 and CHAR has the value of a single dash, then execution of the loop causes the set of variables named 1-,2-,...,10- to be assigned the respective values -,--,...,----------.

A program may begin by executing the FORMLOOP segment repeatedly for each pair of values of CHAR and MAX needed to generate the strings which may be required for formatting within the remainder of the program. Then whenever, say, a string of 42 spaces is needed it may be represented by the expression $(42 '□'), and whenever 10 periods are needed they may be represented by the expression $(10 '.'), provided the FORMLOOP segment has been executed when the value of MAX was at least 42 and the value of CHAR was a space, and when the value of MAX was at least 10 and the value of CHAR was a period. If an expression of this form is written in which the numeric part lies outside the range specified (from 1 to the value of MAX) when the set of variables involved was given value, or in which the character part is not a character which was the value of CHAR when the FORMLOOP segment was executed, then the null value is likely to result; a variable will always be returned from an expression of this form, but not necessarily one to which a value has been assigned.

Concatenation within the operand is also useful as a safeguard against conflicts which occur when a variable returned by the $ operator turns out unexpectedly to be the same as one written directly in the program text as an identifier, and used for some unrelated purpose. In the character-counting example above, the writing of any one-character name within the program text would have produced a conflict of usage if that character had occurred within the text being processed. In that particular case, only variables with one-character names could be returned so the restriction could be made that no one-character names be written in the program text. Often, however, there is no way of knowing which variables will be returned by indirect referencing. Consider the case of counting words, rather than characters, in a text; if the same scheme is employed, then each word of the text will be used as the name of a variable, and there is often no restriction on which words may occur, so a conflict in the use of variables is likely.

Such conflicts may be avoided by using concatenation within the operand of the $ operator to produce a string which is not an identifier; then the variable returned by applying the $ operator to this string will necessarily be one whose name can never be written in the program text. This has been done in the formatting example above by always using a number as the first part of the name, so these names are never in identifier form. Similarly, if the expression $('*' CHAR) were used in place of $CHAR throughout the character-counting program text above, the restriction against the use of one-character names within the program text could be removed; the number of A's in the text would then be referred to by the variable named *A, the number of B's by *B, etc. The two complete program texts which follow in this chapter both rely on concatenation of this form to insure against the possibility of error due to conflict.

<u>A Program to Produce a Frequency Table.</u> The usefulness of multiple indirect referencing is illustrated in the following program, which is similar to the character-counting program but produces instead a frequency table specifying how many letters failed to occur in the text, how many occurred once, how many twice, etc. The program begins in the same way as the character-counting program, by using a variable named by a character to refer to the number of times that character occurred within the text. When all the text has been read in, the character counts themselves are used as the operands of the $ operator to return variables whose names are 0,1,2,...,etc.; the values of these variables are increased by one for each character which occurred that many times within the text.

Concatenation is used in this example to prevent the conflict of variable usage which would occur if the text contained any digits. If concatenation were not used and the text contained, for example some 3's, then the variable named 3 would be used in the first part of the program to refer to the number of 3's occurring in the text; in the second part, when the frequency table was being formed, the variable named 3 would be used to refer to the number of characters which occurred exactly three times in the text. Since the variable named 3 would then already have a value indicating the number of 3's in the text, the frequency table for 3 occurrences would be incorrect. (The program would appear to run correctly and the only indication of error might be an abnormally high count.) Thus concatenation is used to return a variable whose name is 3* for the first part of the program; the frequency table for characters occurring 3 times can then safely be made with a variable whose name is simply 3.

```
*   PROGRAM TO MAKE A FREQUENCY TABLE
*
         CHAR.PAT  =  LEN(1) . CHAR
READ     LINE  =  TRIM(INPUT)              :  F(CHARS)
LOOP1    LINE CHAR.PAT  =  NULL            :  F(READ)
         $(CHAR '*') = $(CHAR '*') + 1 :   (LOOP1)
*
*   SPECIFY THE CHARACTERS WHOSE FREQUENCIES ARE TO BE FOUND
CHARS    ALPHA  =  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
LOOP2    ALPHA CHAR.PAT  =  NULL           :  F(PRINT)
*
*   GIVE MAX THE VALUE OF THE LARGEST COUNT SO FAR FOUND
         MAX = GT($(CHAR '*'),MAX)  $(CHAR '*')
*
*   CHANGE ANY NULL VALUE TO ZERO
         $(CHAR '*') = IDENT($(CHAR '*'),NULL)  0
*
*   USE DOUBLE INDIRECT REFERENCING TO MAKE A COUNT OF COUNTS
FREQ     $$(CHAR '*') = $$(CHAR '*') + 1  :  (LOOP2)
*
*   PRINT THE FREQUENCY TABLE
PRINT    COUNT  =  0
*
*   IF NO LETTERS OCCURRED COUNT TIMES, SKIP IT
LOOP3    IDENT($COUNT,NULL)                :  S(SKIP)
         OUTPUT = $COUNT '¤LETTERS¤OCCURRED¤' COUNT '¤TIMES'
*
*   INCREASE THE VALUE OF COUNT UNTIL THE MAXIMUM IS REACHED
SKIP     COUNT = LT(COUNT,MAX) COUNT + 1  :  S(LOOP3)
END
```

Output from this program would be of the form

```
2 LETTERS OCCURRED 0 TIMES
4 LETTERS OCCURRED 1 TIMES
2 LETTERS OCCURRED 4 TIMES
7 LETTERS OCCURRED 6 TIMES
```

and so on. Such a table would have at most 26 entries; all 26 would be present only if each letter had a different character count associated with it.

The statement labelled FREQ uses double indirect referencing to form variables from these character counts. Its rule represents assignments of the form

```
$'0'  =  $'0' + 1
$'1'  =  $'1' + 1
$'2'  =  $'2' + 1
```

The value assigned to each of these variables is increased by one every time a character is found which occurred that many times in the text.

(Note that it is necessary to assign the value zero rather than the null value to variables representing characters which did not appear in the text. If this were not done, the rule part of the statement labelled FREQ would attempt to represent a rule of the form

```
$''  =  $'' + 1
```

if the value of $(CHAR '*') was null, and an execution-time error would result.)

<u>A Program to Produce a Word Count.</u> As a further example of the use of both multiple indirect referencing and concatenation, consider the following word-counting program which works on the same principle as the character-counting program; it uses each word as the name of a variable and increases the value of that variable by one whenever the word occurs within the text. The process of printing out the words once the counts have been formed, however, is necessarily more complicated than that of printing a character count. While it is possible to specify all the characters which may occur in a text, it is seldom possible to specify all the words. If counts are desired for only certain words, then a list of those words can be supplied as data to the program; but if all words are to be counted, or all words except those specified, then some record must be kept by the program of all different words encountered so

they may be retrieved. In this program, concatenation is used to assign each new word to a variable whose name is of the form W/1, W/2, W/3, etc., so that all words of the text may be recovered for printing with the use of these "successive" variables.

```
*   PROGRAM TO MAKE A WORD COUNT
*   SET UP WORD-FINDING PATTERN
*
         PUNC   =  '□.,:;'
         WORD.PAT  =  BREAK(PUNC) . WORD SPAN(PUNC)
*
*   READ TEXT AND FIND WORDS
READ     LINE  =  TRIM(INPUT) '□'          :  F(OUT)
LOOP1    LINE WORD.PAT  =  NULL            :  F(READ)
*
*   USE CONCATENATION IN FORMING THE WORD COUNT
         $('*' WORD)  =  $('*' WORD) + 1
*
*   TEST TO SEE WHETHER THIS IS A NEW WORD
*     IF NOT, RETURN TO LOOP1
         EQ($('*' WORD),1)                 :  F(LOOP1)
*
*   NEW WORD - ASSIGN IT TO A VARIABLE NAMED W/1, W/2, ETC.
         N  =  N + 1
         $('W/' N)  =  WORD                :   (LOOP1)
*
*   ALL DATA HAS BEEN READ IN - PRINT WORD COUNT TABLE
OUT      M  =  LT(M,N) M + 1               :  F(END)
         OUTPUT  =  $('W/' M) '□□□□' $('*' $('W/' M))
                                           :   (OUT)
END
```

The words are printed in the order of their first occurrence in the text. Output for a well-known six-word text would be

```
TO    2
BE    2
OR    1
NOT    1
```

In the processing of this short text, the rule

```
         $('*' WORD)  =  $('*' WORD) + 1
```

at different times is equivalent to rules of the form

```
$'*TO'  =  $'*TO' + 1
$'*BE'  =  $'*BE' + 1
$'*OR'  =  $'*OR' + 1
$'*NOT' = $'*NOT' + 1
```

and the like, while the rule

```
$('W/' N)  =  WORD
```

is equivalent to

```
$'W/1'  =  'TO'
$'W/2'  =  'BE'
$'W/3'  =  'OR'
$'W/4'  =  'NOT'
```

When the first line of the output is printed, the output statement

```
OUTPUT  =  $('W/' M) '□□□□' $('*' $('W/' M))
```

is equivalent to

```
OUTPUT  =  $'W/1'  '□□□□'  $('*' $'W/1')
```

or

```
OUTPUT  =  $'W/1'  '□□□□'  $'*TO'
```

or

```
OUTPUT  =  'TO□□□□2'
```

<u>Indirect Referencing within the Go-to.</u> The indirect referencing operator may be used within the go-to part of a statement as well as within the rule. When the $ operator is used within the go-to, it takes the string which is its operand and returns the label which is that string. Thus the go-to's

```
                                    :  ($'READ')
```

and

```
                                    :  (READ)
```

have the identical effect of causing a transfer to be taken to the statement labelled READ.

(Note that the $ operator must appear inside the parentheses rather than outside, since the only characters which may appear between the colon and the open parenthesis of the go-to are an S or an F. Thus the go-to : $('READ') is syntactically incorrect. Inner parentheses, such as : ($('READ' N)) are permissible.)

As before, the power of indirect referencing becomes visible only when the operand consists of something besides a literal. The statement

```
LINE  LEN(6) . CODE             :  S($CODE)
```

illustrates the usefulness of the $ operator within the go-to. It causes the first six characters in the value of LINE, if there are that many, to be assigned to the variable CODE, and then, on success, transfers to the label specified by those six characters. (The value of CODE which was obtained in the rule part of the statement is immediately available for use within the go-to.) The single general go-to : ($CODE) may thus represent a great many specific go-to's, one for each possible value of CODE. These values which CODE may acquire must all be in identifier form, since an individual label must actually exist within the program for every possible transfer which is taken. (The indirect referencing operator may not be used in the label field, so there is no way of using a label which is not an identifier.) If an attempt is made to transfer to a non-existent label, an execution-time error will result.

If the special variable INPUT occurs within a go-to in which an indirect referencing operator is used, as in

```
EQ(X,Y)                         :  S($(TRIM(INPUT)))
```

it is assigned as value the next data record, since this string value is needed as the operand of the $ operator. If the next data record had the characters NOUN as its first four characters, followed by spaces, the go-to shown above would send control to the statement labelled NOUN if the rule preceding the go-to succeeded. If INPUT fails, or any other failure occurs in a go-to, then an execution-time error results, since no information will be available as to which statement is to be executed next.

Concatenation is often used within the go-to to send control to "successive" labels of the program. For example, the statement

```
N  =  SIZE(WORD)                :  ($('RULE' N))
```

assigns to N the integer length of the value of WORD, and then transfers control to a label specified by concatenating the characters RULE and this integer; if WORD has as its value any one-character string, a transfer would be taken to the statement labelled RULE1; if WORD has as value a two-character string, then control would be sent to RULE2, etc.

(The statements starting at RULE1 would presumably specify some process to be performed on one-character words, which would be different from the process at RULE2 for two-character words, etc.) The same effect could be achieved by writing

```
                    :  ($('RULE' SIZE(WORD)))
```

Note that some device such as the concatenation of an alphabetic literal is necessary in the above example, since one may not write simply

```
                    :  ($N)
```

or

```
                    :  ($SIZE(WORD))
```

These go-to's would send control to labels of the form 1, 2, 3, etc., and such labels do not exist since they may not be written in the program. Indirect referencing within the go-to is often useful, but is more limited than indirect referencing within the rule: the string designating a label must always be in identifier form and a corresponding label must exist in the program text in order for the transfer to be taken; on the other hand, the string designating the name of a variable may be composed of any characters, since any string names a variable, and there is no need for that variable to have been used in any prior statement of the program.

6A. PROGRAMMER-DEFINED PROCEDURES

In addition to supplying a number of useful predefined procedures, Snobol provides a mechanism which allows a programmer to define any procedure of his own choosing. This permits the task which a program is to perform to be expressed as a series of separate processes of varying degrees of complexity, each of which is defined as a procedure. The more complex procedures may consist mainly of calls to simpler procedures which have been defined earlier; many of these procedures, in turn, will make use of the predefined procedures supplied by the Snobol system. Once the necessary procedures have been written, the writing of a program to perform some task is simplified since it can make reference to the highest-level, most powerful procedures. Program texts written in this fashion are easier to write (and incidentally easier to read) because their organization reflects the structure of the process embodied in the program.

Defining a Procedure. A definition of a new procedure requires two parts: first, the name of the procedure being defined and the form of future references to that procedure must be declared to the Snobol system; second, a description (in Snobol) of what the procedure is to do must be provided, which will be executed each time the procedure is called.

The declaration of a programmer-defined procedure is accomplished by executing a predefined procedure, DEFINE(), which in its simplest form has a single argument consisting of a string which is a sample reference to the procedure. For instance

```
DEFINE('REPEAT(N,OBJECT)')
```

declares a new procedure, REPEAT(), which is defined to have two arguments, represented by the names N and OBJECT. The description of what the REPEAT() procedure is to do can be anything expressible in Snobol. If its purpose is to concatenate some object to itself n times, this might be expressed as follows.

```
REPEAT          N  =  GT(N,0)   N - 1          :  F(RETURN)
                REPEAT  =  REPEAT OBJECT       :   (REPEAT)
```

This section of program text, termed a "procedure body," is written in accordance with a number of conventions which are the subject of the following sections of this chapter. It is identified as the procedure body for the REPEAT() procedure by the label REPEAT, which has the same form as the name of the procedure. The names N and OBJECT are used both in the declaration and in the procedure body to represent the two arguments with which the REPEAT() procedure will be called. The value of N indicates how many times the value of OBJECT is to be concatenated to itself to form the value to be returned by the REPEAT() procedure.

The first statement of the procedure body specifies that the value of N is to be decremented by one if it is still greater than zero; the second statement specifies that the value of OBJECT is to be concatenated to the value of REPEAT, initially null, every time N is successfully decremented. When the value of N becomes zero, then the desired number of concatenations have been performed and the failure transfer to RETURN is taken; this represents not any fixed location in the program, but rather a request to the Snobol system to return to whatever statement contained the call to the REPEAT() procedure. The REPEAT() procedure returns as its value the current value of the variable named REPEAT (again with the same form as the name of the procedure) when the transfer to RETURN is taken.

Once the REPEAT() procedure has been declared and a procedure body provided for it, then it may be invoked by a procedure reference anywhere in the program text. For instance, one might write the assignment rule

```
OUTPUT = REPEAT(10,'X')
```

to specify that a string of 10 X's is to be printed.

The REPEAT() procedure provides a simpler method of producing the varying length strings needed for formatting than the scheme involving indirect referencing described in Chapter 5. Here it is not necessary to store values with a set of successively-named variables in advance of their use in order to insure that a string of the right length will be available; rather the needed string is generated by the procedure call. Using REPEAT(), the alternate records of a data group may be printed in a two-column format, such that the first record of a pair is printed starting in column 1 and the second starting in a column which is the value of N, with a sufficient number of the formatting character which is the value of CH printed in between. The following program segment may be used for that purpose.

```
LOOP    REC1  =  TRIM(INPUT)              :  F(DONE)
        REC2  =  TRIM(INPUT)              :  F(ERROR)
        OUTPUT = REC1 REPEAT((N - 1) - SIZE(REC1),CH) REC2
+                                         :   (LOOP)
```

Since patterns may be concatenated to one another as well as strings, the REPEAT() procedure may take a pattern as its second argument and will then return a pattern as its value. For example, the pattern-matching rule

```
WORD  REPEAT(3,ANY(VOWELS))    :  S(YES3)
```

will succeed and send control to YES3 if the value of WORD contains at least three contiguous vowels.

Procedure names may be defined more than once in a program and even the names of predefined procedures may be redefined (although there is seldom any reason for doing so). In each case, it is the most recent definition which establishes the current meaning of the procedure name, and any preceding definition is lost.

<u>The DEFINE() Procedure.</u> The predefined procedure DEFINE() will accept two arguments, both strings. The basic form of the first argument consists of the name of the procedure being defined followed by a parenthesized list of names of "formal variables" (or "dummy variables") which are used in the procedure body to represent the arguments with which the procedure will be called; in the example above, DEFINE('REPEAT(N,OBJECT)'), the procedure REPEAT() is declared with the two formal variables N and OBJECT.

Procedure names and names of formal variables may be freely invented by the programmer, subject to the usual restriction that they be identifiers. They may be the same as names used elsewhere in the program text for other purposes, because all the names in the first argument of the DEFINE() procedure are used in a special way: when a procedure is called, these names are all made to refer to new variables, "internal" to the procedure call, which are distinct from the variables to which the names previously referred; they will continue to refer to these internal variables until a return from the procedure call is made. (This mechanism will be described in detail in following sections of this chapter.) It turns out to be useful to have other names which are made to refer to internal variables for the duration of each procedure call; these names of additional internal variables, if used, are written immediately following the closing parenthesis of the formal variable list. A definition of a PRINT() procedure, which has three additional internal variables, could be

```
DEFINE('PRINT(N,NAME)M,W,P')
```

The internal variables M, W, and P could then be used within the procedure body where they might be assigned some values, such as tallies, needed only during execution of the procedure call. Notice that the list of additional internal variables is an extension of the string which is the first argument; no embedded blanks are permitted in this string. There is no limit to the number of formal variables and additional internal variables with which a procedure may be declared.

It is also possible to declare a procedure with no formal variables, as in

```
DEFINE('RECORDS()')
```

if the process which the procedure is to perform is not dependent on an argument list. The RECORDS() procedure, for example, might be used to count all records in a group of data read from the input file. Even though there is no argument, the pair of empty parentheses must still appear, both in the declaration and in every reference to the procedure in a program text.

The second argument of the DEFINE() procedure is a string which is the label of a statement in the procedure body which is to be executed first whenever the procedure is called; this label is termed the "entry label." If the second argument is null or missing (and thus null by default), as it has been in all previous examples, the entry label is taken to have the same form as the procedure name. Thus the declaration

```
DEFINE('RECORDS()','RECORDS')
```

would have precisely the same effect as the preceding example, of defining the entry label to be RECORDS.

More commonly, the second argument of DEFINE() is used to insure that the entry label for a procedure body is different from any label which may happen to appear elsewhere in the program text, since all the labels of a program must be unique. Thus the convention may be adopted of forming all entry labels by preceding the name of the procedure with the string PR.; the evaluation rule

```
DEFINE('RECORDS()','PR.RECORDS')
```

declares that the entry label for RECORDS() is the label PR.RECORDS, and the first statement to be executed in the procedure body for the RECORDS() procedure must bear that label. (The labels of the other statements of a procedure body should also be protected from conflicts by adopting some similar conventions.)

The DEFINE() procedure itself returns the null value when it is executed.

Procedure Bodies. A DEFINE() procedure declares to the Snobol system the name of a programmer-defined procedure, the names of its formal variables, additional internal variables, and its entry label, but gives no indication of its effect; that information is supplied by a procedure body, which consists of a series of Snobol statements to be executed whenever the procedure is invoked. A procedure body may consist of any number of Snobol statements, one of which (not necessarily the first) must have the label declared by the DEFINE() as the entry label for this procedure. The statements of a procedure body may be of any kind; they may include procedure declarations and references to other procedures, or even to the procedure being defined. A procedure whose body contains a reference to itself is termed a "recursive procedure"; examples of recursive procedures may be found in Chapter 8.

The statements of a procedure body should be executed only in response to a procedure call, so procedure bodies should be located within a Snobol program text in such a way as to be outside the flow of control of the "main program"; the main program consists of all statements except those of procedure bodies.

The specification of a procedure's action is made general rather than specific by using the names of the formal variables within the procedure body. In the definition of the COUNT() procedure shown below, the formal variables PAT and LINE are used to represent the many different arguments with which this procedure may be called on different occasions.

```
         DEFINE('COUNT(PAT,LINE)','PR.COUNT')   : (END.COUNT)
PR.CCUNT   LINE PAT  =  NULL                    :  F(RETURN)
           COUNT  =  COUNT + 1                  :   (PR.COUNT)
END.COUNT
```

The first statement of the procedure body specifies that the value of the second argument LINE is to be searched for an instance of the first argument PAT; the second statement of the procedure body increments the value of COUNT each time a pattern is found and sends control back to the first statement to institute another search. COUNT() is thus generally defined as a procedure which counts the number of occurrences of some pattern within some string; information as to what pattern and what string are to be used will be supplied to the procedure body by the arguments each time the procedure is called. (Notice how the procedure body has been removed from the flow of control of the main program by the unconditional transfer following its DEFINE() statement.)

The internal variable named COUNT, rather than any other variable, is assigned the result because of a convention which exists for the returning of values: when a success return from a procedure is taken, the last value assigned within the procedure body to the variable whose name is the same as that of the procedure is returned as the value of the procedure call. If that variable, which is termed the "result variable," is assigned no value during the execution of the procedure body, the null value is returned. A value of any datatype may be returned as the value of a procedure call.

<u>The Returns RETURN, NRETURN, and FRETURN.</u> The logical end of a procedure body is signalled by a go-to specifying a transfer to RETURN (the standard success return), to NRETURN (another success return, for returning a variable rather than a value), or to FRETURN (the failure return). These transfers have special system definitions and constitute requests to the Snobol system to return control to the statement from which the procedure was called. Any number of statements in a procedure body may contain transfers to RETURN, NRETURN, or FRETURN; the first such transfer to be executed ends execution of the procedure call. If either success return (RETURN or NRETURN) is executed, the value of the result variable is returned as the value of the procedure call and execution of the calling statement resumes at the point of the call; if the failure return FRETURN is executed, no value is returned but control is sent directly to the go-to of the calling statement where the failure transfer will be taken.

There is no restriction against using RETURN, NRETURN, or FRETURN as the label of any statement within the program text, but if this is done the special system definition of that return is lost. Hence RETURN, NRETURN, and FRETURN must not be used as labels within any program which employs them to return from a programmer-defined procedure, or else a transfer to RETURN, for example, from a procedure body will send control not to the calling statement but to the statement labelled RETURN.

The example below presents another way to write the COUNT() procedure, in which the procedure body includes both RETURN and FRETURN transfers. (An example of a procedure which uses NRETURN may be found toward the end of this chapter.) As before, the procedure is designed to count the number of occurrences of some pattern within some string; here, however, if no instances of the pattern are found, the procedure does an FRETURN, causing failure of the rule from which it was called, rather than returning the null value.

```
          DEFINE('COUNT(PAT,LINE)','PR.COUNT')   : (END.COUNT)
PR.COUNT    LINE PAT  =  NULL              :  F(OUT.COUNT)
            COUNT  =  COUNT + 1            :   (PR.COUNT)
OUT.COUNT   IDENT(COUNT,NULL)    :    S(FRETURN) F(RETURN)
END.COUNT
```

As in the earlier definition of COUNT(), the counting loop is executed until the pattern match fails. When this happens, however, control is sent to the statement labelled OUT.COUNT which tests COUNT to see whether or not it has been incremented. If it has not -- if the pattern match failed on the first attempt -- then COUNT has a null value, the test will succeed, and the procedure will do an FRETURN causing failure of the procedure call; if COUNT is non-null, then the procedure will do a RETURN, returning the value of COUNT as the value of the procedure call. Often, as here, a success transfer may lead to an FRETURN, and a failure transfer to a RETURN.

<u>Procedure Calls.</u> When an assignment statement such as

```
          NUMBERA  =  COUNT('A',RECORD)      :   F(NONE)
```

is executed, the procedure call must be processed before the assignment can take place; hence, execution of the calling statement is temporarily suspended while the Snobol system executes the procedure call.

To carry out the call, the Snobol system begins by taking several automatic actions. First the names in the first argument of the DEFINE() statement are made to refer to new variables which are internal to this call of the procedure. The procedure name now refers to the internal result variable, and the formal variable names refer to internal formal variables. Next the internal variables to which these names now refer are assigned the values needed for the execution of this call: the result variable (COUNT in this case) is assigned the null value, the formal variables are assigned the values of their corresponding arguments (in this example, the formal variable PAT is assigned the character A and the formal variable LINE is assigned the value of the variable RECORD). Since there is no way to make reference to a variable except by using its name, this means that the variables formerly referred to by the names COUNT, PAT, and LINE are inaccessible during the execution of this procedure call.

After this preparation is completed, control is sent to the entry label and execution of the procedure body begins. The action of the procedure is carried out using the values of the arguments provided to the procedure call, since these have just been assigned as the values of the formal variables. The statements of the procedure body are executed in the usual way, until a request for the system to do a return is encountered.

Any return automatically reverses the actions of the preparation process; the names of the procedure and of the formal variables are made to refer to the same variables which they named just before the procedure call was executed, and thus the internal variables, having served their purpose, become in turn inaccessible. The flow of control reverts to the calling statement — on a RETURN, to the point of the procedure call; on an FRETURN, to the go-to.

The Passing of Arguments. When a procedure is invoked, the values of the arguments in the procedure reference are said to be "passed" to become the values of the formal variables. The values of the arguments are assigned to the corresponding formal variables on a one-to-one, left-to-right basis. Any procedure, predefined or programmer-defined, may be called with more or fewer arguments then its definition provides for. Missing arguments are taken to have the null value; extra arguments are evaluated before the procedure call is executed, but are otherwise ignored.

In Snobol, all arguments are passed "by value"; that is, the arguments are evaluated and the resulting values are passed to the procedure body. (In fact, the mechanism for passing arguments has the same effect as if a Snobol assignment rule were executed, with the formal variable on the left side and the argument on the right.) This method of passing arguments assures that the values of variables in the arguments are not affected by execution of the procedure call. For instance, in the call

```
NUMBERA = COUNT('A',RECORD)     : F(NONE)
```

it is the value of the variable RECORD which is passed as the value of the second argument. The procedure will use, not the variable RECORD, but only the internal formal variable LINE which has been assigned the value of RECORD at the time of the call. Thus the value of RECORD is always the same before and after a call of the COUNT() procedure is executed.

The arguments used in a procedure reference may be any expressions having values which the procedure body will handle properly. A call to COUNT() such as in the statement

```
NUMBERV = COUNT(ANY('AEIOU'),RECORD)   : F(NONE)
```

would pass the pattern returned as the value of the procedure call ANY('AEIOU') to be the value of the variable PAT. Since PAT is used in the pattern part of a statement, a pattern value is appropriate and the number of vowels within the value of RECORD will be returned as the value of this call to the COUNT() procedure.

While the first formal variable, PAT, may acquire either a string or a pattern value, the second formal variable, LINE, may acquire only a string as value, since it is used within the procedure body as a string reference. Execution of a procedure call of the form

```
NUMBERV = COUNT(RECORD,ANY('AEIOU'))   : F(NONE)
```

(in which the programmer has presumably forgotten the correct order of the arguments) will pass the formal variable LINE a pattern value; when the procedure body is entered an execution-time error will result, since the first field in a replacement rule cannot be a pattern.

<u>Additional Internal Variables.</u> The names of variables which are to be internal to a procedure call (in addition to the result variable and any formal variables) are also made to refer to distinct internal variables at each procedure call, thus making the variables previously referred to by those names temporarily inaccessible; the names are restored to their former significance when a return from the procedure call is taken. The internal variables which they name are initially null at every call of the procedure just like the result variable. There are thus two possible reasons for declaring additional internal variables: to prevent their names from conflicting with names used elsewhere for other purposes, and to take advantage of the automatic null initialization at each call. Any number of additional internal variables may be declared by writing their names in the first argument of a DEFINE() procedure.

As an example of the usefulness of additional internal variables, consider the LONGER() procedure which employs four of them. This procedure compares the two strings given as the values of its first two arguments to determine which contains the longer sequence of the characters specified by the value of its third argument; it returns as its value the string containing the longer sequence. If the size of the longest sequence in both strings is the same, then by convention the first string is returned as the value of the procedure call; if neither string contains a character given by the third argument, a transfer to FRETURN is taken causing failure of the procedure call. Thus execution of the assignment statement

```
         OUTPUT = LONGER('HILARIOUS','TREACHEROUS','AEIOU')
+                                            :  F(NOVOWEL)
```

would cause the string HILARIOUS to be printed since its longest vowel sequence is longer than any vowel sequence in the string TREACHEROUS.

```
         DEFINE('LONGER(S1,S2,SEQ)T1,T2,SAVE,LONGEST',
+             'PR.LONGER')                   :    (END.LONGER)
*  MAKE COPIES OF THE TWO STRINGS TO BE COMPARED
PR.LONGER  T1  =  S1
           T2  =  S2
*  FIND THE LONGEST SEQUENCE IN THE FIRST STRING
*    ASSIGN ITS SIZE TO THE INTERNAL VARIABLE NAMED LONGEST
T1.LONGER  T1 SPAN(SEQ) . SAVE = NULL  :  F(T2.LONGER)
           LONGEST = GT(SIZE(SAVE),LONGEST)  SIZE(SAVE)
+                                      :    (T1.LONGER)
*  SEE IF THERE IS A SEQUENCE IN THE SECOND STRING
*    WHICH IS LONGER THAN THE LONGEST SEQ IN THE 1ST STRING
*    IF SO, ASSIGN THE SECOND STRING AS THE VALUE OF THE
*    RESULT VARIABLE AND RETURN
T2.LONGER  T2 SPAN(SEQ) . SAVE = NULL  :  F(OUT.LONGER)
           LONGER = GT(SIZE(SAVE),LONGEST)  S2
+                            :  S(RETURN)  F(T2.LONGER)
*  IF NO SEQUENCE WAS FOUND IN EITHER STRING, FAIL
*    OTHERWISE RETURN THE FIRST STRING AS VALUE OF THE CALL
OUT.LONGER LONGER  =  DIFFER(SAVE,NULL)  S1
+                            :  S(RETURN)  F(FRETURN)
END.LONGER
```

This procedure uses four additional internal variables named T1, T2, SAVE, and LONGEST. T1 and T2 are needed because the method used for determining the longest vowel sequence in S1 and S2 deletes each vowel sequence which is found. Since the original strings must be preserved to be returned as the value of the procedure call, the replacement statements T1.LONGER and T2.LONGER use the variables T1 and T2 rather than S1 and S2, allowing the values of S1 and S2 to remain unchanged. The internal variable SAVE is assigned each vowel sequence which is found. The fact that SAVE is given the null value initially allows the test in the statement labelled OUT.LONGER to determine whether or not any vowel sequences have been found; if SAVE still has its null value, then neither string contains a vowel and an FRETURN is taken. The internal variable LONGEST is used to keep track of the size of the currently longest vowel sequence as each is successively found within the first string. When the determination of the size of the longest sequence has been completed, this number is then compared with the size of each vowel sequence as it is found in the second string until either a longer sequence is found (in which case the second string is returned as the value of the procedure call) or until all vowel sequences have been considered (in which case either the first string is returned or failure is signalled).

Since in this procedure body the internal variables T1 and T2 are assigned the values of the arguments as soon as the procedure body is entered, the only reason for declaring them to be internal is to prevent conflicts with other uses of the names T1 and T2. The internal variables SAVE and LONGEST are similarly protected, but also take advantage of the fact that they are initialized to null each time the LONGER() procedure is called.

Note that the use of the additional internal variable LONGEST is not really necessary since the result variable LONGER may be substituted for it wherever it occurs. Result variables have exactly the properties of additional internal variables until a success transfer is taken, so they are often assigned temporary values which are needed during the processing of a procedure call. When the final value of a call has been determined, it can then be assigned to the result variable and a return made to the statement in which the procedure call occurred.

<u>References to External Variables.</u> The principle of a programmer-defined procedure is that of a "sub-program," independent of the program with which it is used; it receives values through its arguments, performs some process using those values, and returns the result. If temporary values are needed, the procedure assigns them to additional internal variables, so that it avoids changing the values of any variables not internal to itself, i.e., those whose names do not appear within the first argument of the DEFINE() statement for the procedure.

Procedures written in such a way as to make reference to no values other than those of their internal variables (or to literals within their own bodies), and which assign values only to their own internal variables, are desirable for many reasons. They are easy to move from program to program since they will operate correctly regardless of their environment, and they are easy to use because they can influence that environment only through the result which they return (including, of course, the possible "result" of failing).

At the same time, there are sometimes good reasons for relaxing this discipline, in pursuit of the same goals for which procedures are written in the first place: to make programs easier to write and clearer to read. One example of such a motivation has already come up in some of the examples; in the procedure body for the LONGER() procedure, for example, the statement

```
T1.LONGER     T1 SPAN('AEIOU') . SAVE = NULL : F(T2.LONGER)
```

occurs. Here NULL is the name of a variable which is external to the call of the LONGER() procedure; since the name NULL is not included in its declaration, it receives no special treatment when this procedure is called; it continues to refer to the same variable before, during, and after a call to LONGER(). Thus, if LONGER() were to be called from a program which had assigned some non-null value to the variable named NULL, it would not work as intended.

In this case there are several ways to restore the independence of the LONGER() procedure; the identifier NULL can be replaced in its body by a literal null string (two adjacent quotation marks), or by nothing, or the name NULL can be declared as naming an additional internal variable for LONGER(), thus assuring that NULL will refer to a variable initialized to the null value each time LONGER() is called. For this procedure such precautions seem extreme, but they might make sense if LONGER() were a much more complicated procedure, and were intended for use by people other than its programmer.

As another motivation for making reference to external variables, consider a programmer-defined test procedure which determines whether or not the string given as its argument is a palindrome, that is, whether it reads the same from left to right as from right to left. The complete program presented below uses the PALIN() procedure to perform this test. The program reads all trimmed records of a group of data but prints only those which are palindromes.

```
*        PALINDROME-FINDING PROGRAM
*
*  SET UP PATTERN NEEDED BY THE PALIN() PROCEDURE
*    ASSIGN IT TO A MAIN-PROGRAM VARIABLE
         PAL.PAT = POS(0) LEN(1) $ CH RTAB(1) . CAND *CH
         DEFINE('PALIN(CAND)CH','PR.PALIN') :  (END.PALIN)
*
*  IF CANDIDATE NOW CONSISTS OF 1 OR 0 CHARACTERS, SUCCEED
*    OTHERWISE APPLY THE PATTERN AGAIN
PR.PALIN LE(SIZE(CAND),1)                 :  S(RETURN)
         CAND  PAL.PAT       :  S(PR.PALIN)  F(FRETURN)
END.PALIN
*
READ     RECORD  =  TRIM(INPUT)           :  F(END)
PRINT    OUTPUT = PALIN(RECORD) RECORD :   (READ)
END
```

Output from this program could be strings of the form

```
HANNAH
I
ROTOR
...
NOON
SAGAS
*
103595301
YREKABAKERY
><>  <><>  <><
```

The PALIN() procedure uses virtually the same pattern as that shown at the end of Chapter 4 for finding words with identical first and last characters; the pattern is changed only by the re-assignment of the substring matched by RTAB(1) to the variable named CAND. Thus, on each iteration of the loop the string being searched is shortened by the loss of its first and last characters; a new set of first and last characters is then tested for identity. The loop is executed until either (1) the end characters being tested are found to be different, upon which an FRETURN is taken signifying that the string is not a palindrome, or (2) the size of the string is reduced to zero or one, in which case a RETURN is taken since this indicates that all characters have been tested and that the string is a palindrome. Note that the rule in the statement labelled PR.PALIN will succeed immediately if the size of the argument is either zero or one, meaning that strings of one or no characters are palindromes by definition. The PALIN() procedure returns the null value on success, since the result variable PALIN is not assigned a value within the procedure body.

Here the pattern on which PALIN() relies is constructed once, in the statement just above the DEFINE(), and assigned to the variable PAL.PAT. The reason for doing this is clear: since internal variables are internal to a single call of a procedure and their values never persist between calls, if PAL.PAT were declared to be the name of an additional internal variable of PALIN() then the pattern assignment would have to be moved into the procedure body, and thus the pattern would have to be constructed anew at each call of the PALIN() procedure -- a substantial amount of unnecessary effort.

It is true that PALIN() will not work properly if the program calling it inadvertantly assigns a different value to the variable PAL.PAT. It might seem that this kind of error could be avoided by rewriting PALIN() to accept the pattern as another argument, rather than merely using the value of an external variable; but that turns out not to be true. A call to such a re-written PALIN() procedure would be something like

```
        PALIN(POS(0) LEN(1) $ CH RTAB(1) . CAND *CH,RECORD)
```

Apart from the bother of writing the invariant pattern in every reference to PALIN(), the pattern is once again being constructed at each call of PALIN() -- in the evaluation of the argument, rather than within the procedure body. The calling program can avoid the repeated evaluation of the pattern by executing the assignment statement

```
        PAL.PAT  =  POS(0) LEN(1) $ CH RTAB(1) . CAND *CH
```

and then making references to the procedure in the form

```
        PALIN(PAL.PAT,RECORD)            :  F(NOPALIN)
```

But now, just as before, the calling program is responsible for assuring that PAL.PAT has the correct value at the time of the call. So the original PALIN() procedure cannot be improved upon in this way, and has the additional merit of requiring only one argument instead of two. The conclusion to be drawn is that a pattern used by a procedure must either be constructed at each procedure call, or else must be assigned as the value of an external variable so that it will be available for use by repeated procedure calls.

Notice, however, how the pattern which is the value of the main-program variable PAL.PAT can cause assignments to the internal formal variable named CAND and to the additional internal variable named CH within the PALIN()

procedure. The pattern PAL.PAT calls for immediate assignment to whatever variable is currently referred to by the name CH, and conditional assignment to whatever variable is currently referred to by the name CAND -- it specifies nothing about which variables those must be. If PAL.PAT is used in a statement of the main program, then it will cause assignments to the main-program variables named CH and CAND. At a call of the PALIN() procedure, though, those two names are made to refer to different variables, internal to the procedure call; so if PAL.PAT is used (as above) in a statement within the body of PALIN(), it will cause assignments to the two variables internal to the call.

<u>Side-effects of Procedures.</u> Just as there are sometimes reasons for making reference to the values of external variables, so are there reasons for altering their values as well. A procedure call which alters the value of a variable not internal to the call is said to have a "side-effect." This terminology exists because of the presumption that the main effect of a procedure is to return a value or to direct the flow of control; in fact, however, procedures are often written solely for the purpose of producing side-effects.

One reason for defining a procedure which produces a side-effect is to keep some sort of record of occurrences inside and outside of procedure calls. For instance, the COUNT() procedure presented earlier could be changed so that in addition to its former action of returning as its value the number of instances of some pattern within some string, it also increments an external counter by that number. This new version of COUNT(), TCOUNT(), could be written as follows.

```
          DEFINE('TCOUNT(PAT,LINE)','PR.TCOUNT')        : (END.TCOUNT)
PR.TCOUNT LINE PAT  =  NULL                         :  F(OUT.TCOUNT)
          TCOUNT  =  TCOUNT + 1                     :   (PR.TCOUNT)
OUT.TCOUNT TALLY  =  TALLY + TCOUNT                 :   (RETURN)
END.TCOUNT
```

Aside from the systematic replacement of COUNT by TCOUNT, this procedure definition is the same as that of the first version of COUNT(), except that before returning the procedure increments the value of the external variable TALLY by the value of the result variable. Since TALLY is not an internal variable, its value can be increased throughout a program over repeated calls to TCOUNT(), and thus represent a total of the results of many invocations of that procedure; for that matter, TALLY might also be incremented by other assignments in the main program or by calls to other procedures as well.

The inclusion of the side-effect involving TALLY specializes the COUNT() procedure, and the same record could be kept without recourse to side-effects by keeping the tally entirely in the main program, as in the segment

```
              RESULT  =  COUNT('A',RECORD)
              TALLY  =  TALLY + RESULT
```

and so forth. But that requires that the tally-incrementing statement be written once for every reference to the procedure; if there are many references to COUNT() in a program, then the whole text can be shortened considerably by writing the statement which increments TALLY once in the TCOUNT() procedure body and permitting the side-effect to occur.

Another reason for changing the value of an external variable in a procedure body is to take advantage of an output association which that variable may have. A SKIP() procedure can be defined, for example, to "skip" the number of lines specified by its argument by assigning the null value repeatedly to the main-program variable named OUTPUT.

```
            DEFINE('SKIP(NUM)','PR.SKIP')  :   (END.SKIP)
PR.SKIP     NUM  =  GT(NUM,0)  NUM - 1     :  F(RETURN)
            OUTPUT  =  NULL                :   (PR.SKIP)
END.SKIP
```

If SKIP() is called in the sequence

```
              OUTPUT  =  HEAD1
              SKIP(3)
              OUTPUT  =  HEAD2
```

then the first heading, the three empty lines, and the second heading are all written to the same file, the one with which the variable OUTPUT is associated, since the variable referred to by the name OUTPUT is the same both inside and outside the procedure call. Note that SKIP() would not work as intended if OUTPUT were declared to refer to a variable internal to the procedure call, since the association is with the main-program variable, not with the name OUTPUT.

Quite a different motivation for side-effects arises when a procedure does not have a fixed name of an external variable in its procedure body, but rather can change the values of different variables when it is called with different arguments.

One way to do this is to define a procedure which has a string as its argument and which uses indirect referencing within its procedure body to refer to an external variable named by that string, or by a string derived from it. Consider the following STORE() procedure, whose purpose is to store the string which is its first argument as the value of one of a set of successively-named variables; the name of the variable which is to be used is formed by concatenating the length of the string to be stored, then the value of the second argument of STORE(), then the index number of the next available successively-named variable of the set. If the procedure reference

```
STORE('CAT','LIST')
```

is written, for instance, and CAT is the first three-letter word to be stored, then it will become the value of the variable named 3LIST1. If STORE() were called repeatedly with the string LIST as its second argument, then it would store one-character strings as the values of the variables 1LIST1, 1LIST2, ..., $(1 'LIST' N), two-character strings as the values of 2LIST1, 2LIST2, ..., $(2 'LIST' N), etc. The STORE() procedure further keeps track of the last used index number for each 'list' by storing these numbers as the values of the variables 1LIST, 2LIST, ..., $(N 'LIST'). Note that all names formed by the STORE() procedure depend on the value of its second argument, but all begin with a number and so are necessarily distinct from any names which may be written in the program text.

The definition of the STORE() procedure could be

```
        DEFINE('STORE(WORD,NAME)','PR.STORE') : (END.STORE)
*
*  ADD ONE TO THE INDEX NUMBER FOR THIS SIZE WORD LIST
PR.STORE $(SIZE(WORD) NAME)  =  $(SIZE(WORD) NAME) + 1
*
*  STORE THE WORD AS THE VALUE OF THE "NEXT" VARIABLE
         $(SIZE(WORD) NAME  $(SIZE(WORD) NAME))  =  WORD
+                                            :   (RETURN)
END.STORE
```

STORE() is thus a procedure which always succeeds, returning the null value. Its purpose is always to have the side-effect of changing the value of one of the great many external variables whose names are dependent on the various values of its second argument.

Levels of Internal Variables. When a procedure call is to use variables other than those internal to itself, either to refer to their values or to assign new values to them, then the particular relation between names and variables at any time becomes important. In the preceding sections the examples have assumed that a procedure was called from a main program, and thus all names either referred to variables internal to the procedure call, or else to variables associated with the main program. But the situation may be more complicated than this, because one procedure may be called and then it may call another procedure: if the second procedure makes reference to variables other than its own internal variables, the possiblity exists that it may use a name which refers to one of the internal variables of the procedure which called it, rather than to a main-program variable external to both of them. Sometimes this is what was intended and sometimes not; care must be taken to insure that the names used by procedures will always refer to the intended variables.

The number of sets of internal variables which have become temporarily accessible at any point in time during execution is termed the "level" of execution. When a program begins executing, it is at level zero and the statements executed at level zero are the technical definition of the main program. If a statement of the main program calls a procedure, the statements of that procedure's body will be executed at level one; if that procedure calls a second procedure before returning, then the statements of the second procedure's body will be executed at level two. When the second procedure does a return, the first procedure will resume execution at level one; when it returns, the main program will resume execution at level zero. It may then call another procedure which will execute at level one, and so forth. Any number of levels may be attained; there is no level lower than zero, however, so any attempt to do a return from a statement of the main program (caused by allowing control to flow into a procedure body by accident rather than through a procedure call) will cause an execution-time error. Such an error can be caused by neglecting to write an unconditional transfer following a DEFINE() procedure in any of the above examples.

At different times a procedure may be executed at different levels, depending on the length of the chain of calls by which it was reached. The only change in executing at different levels is in the variables to which names refer. A procedure executing at level three, for example, will be executing in an environment in which most names refer to main-program variables, but some names refer to variables internal to whatever procedure call is at level one, some names refer to variables internal to whatever procedure call is at level two, and some names refer to its own internal variables at level three. If this same procedure is later called directly from a statement of the main program, then all names except those of its own internal variables will refer to main-program variables. This difference in environment must be considered to assure that a procedure will refer to and assign values to the intended external variables, no matter from what level it is called and no matter which procedure (and thus what names of internal variables) are at levels below it in any particular chain of calls.

As an illustration of the same name referring in different environments to variables at three different levels, consider an improved version of the PALIN() procedure, PALIND(), which would delete all spaces and punctuation characters from its argument before testing it for being a palindrome, thus allowing strings of the form DOC, NOTE. I DISSENT. A FAST NEVER PREVENTS A FATNESS. I DIET ON COD to be accepted. In the complete program below the name CAND is used to refer to the trimmed record read from the input file, to the formal variable of the PALIND() procedure, and to a formal variable of the DELETE() procedure which is called by the PALIND() procedure to perform the deletion. Nevertheless, there is no possibility of the name CAND referring to a variable at the wrong level: within the PALIND() procedure (in this example) it always refers to an internal variable at level one, while within the DELETE() procedure it always refers to an internal variable at level two. The level zero variable named CAND can thus be referred to only by statements of the main program.

```
            DEFINE('PALIND(CAND)CH','PR.PALIND')
*
*  SET UP PATTERN NEEDED BY THE PALIND() PROCEDURE
*    ASSIGN IT TO A MAIN-PROGRAM VARIABLE
            PAL.PAT = POS(0) LEN(1) $ CH RTAB(1) . CAND *CH
+                                      :    (END.PALIND)
*
*  CALL DELETE() TO REMOVE SPACES AND PUNCTUATION FROM ARG
PR.PALIND   CAND  =  DELETE(ANY('□.,:;'),CAND)
*
*  PROCEED AS IN THE PALIN() PROCEDURE
LOOP.PALIND LE(SIZE(CAND),1)                :  S(RETURN)
            CAND PAL.PAT     :  F(FRETURN) S(LOOP.PALIND)
END.PALIND
*
```

```
             DEFINE('DELETE(PAT,CAND)','PR.DELETE')
+                                                 :    (END.DELETE)
*
*  REMOVE ALL PATTERNS FROM THE CANDIDATE
PR.DELETE    CAND  PAT  =  NULL                      :  S(PR.DELETE)
             DELETE  =  CAND                         :   (RETURN)
END.DELETE
*
*  MAIN PART OF PROGRAM
*
*  READ ALL RECORDS BUT PRINT ONLY THE PALINDROMES
READ         CAND  =  TRIM(INPUT)                    :   F(END)
PRINT        OUTPUT  =  PALIND(CAND)  CAND           :    (READ)
END
```

In this program the two DEFINE() statements, the assignment to PAL.PAT, the READ statement, the PRINT statement, and the END statement constitute the complete main program. These statements are executed in the order specified by the go-to's until an attempt is made to perform the assignment in the PRINT statement; before this assignment can occur, the value of the call to the PALIND() procedure must be obtained. This call causes the variable named CAND, internal to level one, to be assigned the same value as the main-program variable CAND, that is, the candidate to be tested, and a transfer to be taken to PR.PALIND. Before the assignment specified in this statement can be performed, however, a call to the DELETE() procedure must be processed. This causes the variable named CAND internal to the level two call of DELETE() to be assigned the same value as that of the level one variable CAND, the string to be tested. This string is searched repeatedly for spaces and punctuation characters and when all have been deleted the resulting, possibly shortened, string is returned to the statement PR.PALIND where it is assigned as the new value of the level one variable CAND. The value of this variable is then searched, perhaps repeatedly, for the PAL.PAT pattern; each time the search is successful, the value of the level one variable CAND is shortened by the loss of its first and last characters. If the candidate is indeed a palindrome, then the final value of the level one variable CAND will be a string of one or zero characters, the PALIND() procedure will take the success return and transfer back to the statement labelled PRINT. Here the value of the level zero variable named CAND, the original string as it was read from the input file, is printed whenever PALIND() succeeds.

Output from this program could be strings such as

```
CIVIC
SUMS ARE NOT SET AS A TEST ON ERASMUS.
ROTOR
DEIFIED
DENNIS AND EDNA SINNED.
****** ***** **** *** ** *
```

There are two different ways of classifying variables, which are useful in different descriptions of procedures. On the one hand, there are main-program variables, at level zero, as opposed to the internal variables at higher levels; it is the level zero, or main-program, variables which have the lasting values associated with all names, while internal variables at all higher levels become accessible only temporarily during procedure calls and are initialized anew at each call. On the other hand, from the viewpoint of discussing any particular procedure call, the distinction is between names of internal variables which are always its own, as opposed to external variables which may be different variables when the procedure executes at different levels.

The important special case in which these two descriptions are equivalent is for procedures executing at level one; at level one, the external variables are all main-program variables. The fact that external variables cannot be guaranteed to be main-program variables at level two and above without a painstaking check of the names of all internal variables through all possible chains of calls, is one reason for avoiding unnecessary references to external variables in procedure bodies.

<u>The Use of NRETURN to Return a Variable.</u> Any procedure call which returns a non-null string (or an object of datatype Name) may occur to the left of an assignment sign as the operand of an indirect referencing operator. This was indicated in Chapter 5 with the rule

```
          $SIZE(WORD)  =  $SIZE(WORD) + 1
```

and may be further illustrated by the rule

```
     $COUNT(ANY(VOWELS),WORD) = $COUNT(ANY(VOWELS),WORD) + 1
```

which adds one to the value of the variable named by the number of vowels found within a word. As another example, the statement

```
        $TRIM(INPUT) = LINE1            : F(DONE)
```

assigns the value of LINE1 to the variable named by the characters of the next trimmed data record, or causes an execution-time error if the trimmed record is null.

Programmer-defined procedures can be written specifically for the purpose of returning a string which will be used as the operand of the $ operator to return a variable. Consider, for example, the problem of determining the first null-valued variable of the set LIST1, LIST2, ..., $('LIST' N), described in Chapter 5, and then assigning that variable the value of the next data record. A procedure named NEXTNULL() might be written to determine the first null-valued variable as follows.

```
             DEFINE('NEXTNULL(NAME)N','PR.NEXTNULL')
+                                       :    (END.NEXTNULL)
PR.NEXTNULL  N  =  N + 1
             NEXTNULL = IDENT($(NAME N),NULL) NAME N
+                              :  S(RETURN) F(PR.NEXTNULL)
END.NEXTNULL
```

The NEXTNULL() procedure cannot fail so it may be used in a statement of the form

```
        $NEXTNULL('LIST') = TRIM(INPUT)   :   F(NODATA)
```

The procedure is called with a string-valued argument representing that part of the name which is common to all the variables. This string is concatenated to the value of the variable N internal to the procedure call, and the $ operator is applied to the result of this concatenation to return a variable. If the value of this variable is null, a string representing the name of the variable is formed by concatenation and assigned as the value of the result variable; this string is returned as the value of the procedure call where it is used as the operand of the $ operator which returns the variable needed to perform the assignment.

Since N is declared as internal, it is assigned the null value every time the NEXTNULL() procedure is called, hence the search for the "next" variable always begins from one. If the search were to begin from the value given N the last time the procedure returned, i.e., from the last variable located, then N should not be declared as internal so that it would retain its value from one procedure call to the next.

A procedure can be caused to return a variable, rather than a string which can be used by the $ operator to return a variable, with the use of the name return NRETURN. This return may be used only if the value of the result variable is a string (or a Name); it effectively applies the $ operator to the value of the result variable, causing the variable named by that value to be returned as the value of the procedure call. Using NRETURN, the NEXTNULL() procedure may be written as follows.

```
               DEFINE('NEXTNULL(NAME)N','PR.NEXTNULL')
+                                             :    (END.NEXTNULL)
PR.NEXTNULL    N  =  N + 1
               NEXTNULL = IDENT($(NAME N),NULL) NAME N
+                                   :  S(NRETURN)  F(PR.NEXTNULL)
END.NEXTNULL
```

This version of NEXTNULL() is exactly the same as its predecessor except that NRETURN has been written instead of RETURN in the last statement of the procedure body, causing the variable named by the string formed by concatenating the value of NAME and N to be returned, rather than that string. A reference to this new NEXTNULL() procedure would have the form

```
               NEXTNULL('LIST')  =  TRIM(INPUT)  :  F(NODATA)
```

The $ operator is now not wanted before the procedure reference since NRETURN has effectively applied it already.

NRETURN is provided for convenience only; its effect may always be obtained by using RETURN within the procedure body to return the name of a variable, and by placing a $ operator directly before the procedure reference. Further examples of the use of NRETURN may be found in Chapters 7 and 8.

<u>The APPLY() Procedure.</u> A procedure reference in a program text is composed of a procedure name followed directly by an argument list enclosed within parentheses. Although these arguments may be represented by arbitrarily complex expressions, which when evaluated yield appropriate values, the procedure name may not be so represented but must be an identifier.

There are some applications, however, in which the programming would be much simplified if one could indicate generally, rather than specifically, which procedure is to be called. Consider, for example, a series of procedures named FIX1, FIX2, FIX3, etc., each one designed to "fix" a word of the indicated length. A procedure call something like $('FIX' SIZE(WORD))(WORD) is what is needed in order to call the appropriate procedure for any given word, but this expression is syntactically incorrect.

Assigning an expression representing the procedure name to another variable, as in

```
TEMP = 'FIX' SIZE(WORD)
```

and then applying the $ operator as in $TEMP(WORD) gives an expression which is syntactically correct but does not produce the desired result; in this case the procedure call TEMP(WORD) is evaluated, and its value used as the operand of the $ operator. (Of course, if no procedure TEMP() were defined — the most likely case -- an execution-time error would result when it was called.)

A way of calling a procedure, in which the name of the procedure to be called is determined at execution-time, is provided by the predefined procedure APPLY() whose first argument may be any expression which yields a string naming the procedure to be called, and whose remaining arguments are any expressions representing the arguments to be supplied to that procedure. APPLY() may be applied to predefined procedures as well as to programmer-defined ones; thus

```
WORD = APPLY('TRIM',INPUT)
```

is equivalent to

```
WORD = TRIM(INPUT)
```

and

```
OUTPUT = APPLY('LONGER',STRING1,STRING2,VOWELS)
```

is equivalent to

```
OUTPUT = LONGER(STRING1,STRING2,VOWELS)
```

More usefully, the designation of the appropriate procedure from the set FIX1, FIX2, FIX3, etc., could be made with the evaluation rule

```
APPLY('FIX' SIZE(WORD),WORD)
```

which is equivalent to the rule

```
FIX3(WORD)
```

if WORD has a value three characters long. Similarly, executing the statement

```
APPLY(TRIM(INPUT),ARG1,ARG2)    :   F(ERROR)
```

calls the procedure whose name is specified on the next data record, giving it the two arguments ARG1 and ARG2.

The value returned by APPLY() is the value returned by the procedure which it calls, and APPLY() returns with whatever return (RETURN, NRETURN, or FRETURN) is used by that procedure.

Note that APPLY() is defined to have a varying rather than a fixed number of arguments, always one more than that of the procedure specified in its first argument. However, the usual rules about missing and extra arguments pertain: if the number of arguments beginning with the second exceeds the number of formal variables specified for the procedure being called, the extra arguments are evaluated but otherwise ignored; if there are fewer arguments than formal variables, each remaining formal variable is assigned the null value.

Although the name of the procedure may be represented by an expression of any complexity, that expression must yield a string which is an identifier when evaluated. This restriction comes about because all the names in the first argument of the DEFINE() procedure must be identifiers; all predefined procedures, of course, have names which are in identifier form.

<u>Using a Library of Procedures.</u> Most tasks which a program is to perform divide themselves naturally into a series of smaller tasks, some of which are so basic as to be repeated many times during the course of the program. If each basic part is written as a procedure, then the organization of the program can be clearly seen; the body of each procedure need occur within the program text only once, but it may be referred to whenever it is needed. Once a procedure has been thoroughly tested, it may form part of the programmer's "library" to be used, just as the predefined procedures are used, as a part of many different programs.

The complete program text below begins by providing the library of procedures to which it will refer; with the exception of the PRINT() procedure, these procedures have all occurred earlier in this chapter with the same definitions. After the library comes the main program, which consists largely of references to these procedures. The purpose of the program is to read data from the input file, isclate the words, and store them in "lists" according to their size. When all the words have been read in and stored, the lists are printed, in crder of increasing word size, with the words in each list in the order in which they were encountered. In addition, each word of a list which is a palindrome is underlined by printing a row of hyphens beneath it on the succeeding line. At the end of each list, numbers are printed indicating the number of words in the list and the number of palindromes; when all the lists have been printed, the total number of words and of palindromes is also provided.

The main program begins by determining the characters which are to be considered as punctuation by reading them in frcm the first record of the input data. It then proceeds to read each subsequent data record, which consists of words separated by spaces and punctuation and appearing in no fixed format, except that no word is broken across a record. As each word is found, the STORE() procedure is invoked to store the word in the list appropriate to its size. When all the words have been processed, the PRINT() procedure is called to print the lists, shortest words first, and to underline each word which is a palindrome. The PRINT() procedure invokes the PALIN() procedure to determine whether or not the word is a palindrome, the REPEAT() procedure to form an underline of the needed length, and the SKIP() procedure to produce blank lines. The PRINT() procedure counts the words and palindromes occurring in each list by incrementing the values of the internal variables W and P, printing their values before it returns. It also adds to the total count of words and palindromes by incrementing the values of the main-program variables WORDS and PALINS; these values persist and increase through successive calls to PRINT().

```
*   PROCEDURE TO CONCATENATE A STPING OR PATTERN N TIMES
*
            DEFINE('REPEAT(N,OBJECT)','PR.REPEAT')
+                                           :    (END.REPEAT)
PR.REPEAT  N  =  GT(N,0)   N - 1            :  F(RETURN)
           REPEAT  =  REPEAT OBJECT         :    (PR.REPEAT)
END.REPEAT
*
*   TEST PROCEDURE TO FIND PALINDROMES (FAILS IF NOT A PALIN)
*
            DEFINE('PALIN(CAND)CH','PR.PALIN')
```

```
*   SET UP PATTERN NEEDED BY THE PALIN() PROCEDURE
*     ASSIGN IT TO A MAIN-PROGRAM VARIABLE
         PAL.PAT = POS(0) LEN(1) $ CH RTAB(1) . CAND *CH
                                               :   (END.PALIN)
*   IF CANDIDATE NOW CONSISTS OF 1 OR 0 CHARACTERS, SUCCEED
*     OTHERWISE APPLY THE PATTERN AGAIN
PR.PALIN LE(SIZE(CAND),1)                      :  S(RETURN)
         CAND   PAL.PAT         :  S(PR.PALIN)   F(FRETURN)
END.PALIN
*
*   SIDE-EFFECT PROCEDURE TO TO SKIP N LINES ON OUTPUT FILE
*
         DEFINE('SKIP(NUM)','PR.SKIP')    :   (END.SKIP)
PR.SKIP  NUM  =  GT(NUM,0)   NUM - 1      :  F(RETURN)
         OUTPUT  =  NULL                  :   (PR.SKIP)
END.SKIP
*
*   SIDE-EFFECT PROCEDURE TO STORE WORDS IN LISTS BY SIZE
*
         DEFINE('STORE(WORD,NAME)','PR.STORE') : (END.STORE)
*
*   ADD ONE TO THE INDEX NUMBER FOR THIS SIZE WORD LIST
PR.STORE $(SIZE(WORD) NAME)  =  $(SIZE(WORD) NAME) + 1
*
*   STORE THE WORD AS THE VALUE OF THE "NEXT" VARIABLE
         $(SIZE(WORD) NAME $(SIZE(WORD) NAME))  =  WORD
+                                         :   (RETURN)
END.STORE
*
*   PROCEDURE TO PRINT WORDS, UNDERLINE PALINS, KEEP COUNTS
*
         DEFINE('PRINT(N,NAME)M,W,P','PR.PRINT')
+                                         :   (END.PRINT)
PR.PRINT OUTPUT  =  'LIST□OF□'  N  '-LETTER□WORDS'
         SKIP(1)
*
*   TEST FOR END OF LIST - IF NOT END, PRINT NEXT WORD
UP.PRINT M  =  LT(N,$(N NAME))  M + 1  :  F(DONE.PRINT)
         OUTPUT  =  $(N NAME M)
*
*   ADD ONE TO THE WORD COUNT FOR THIS SIZE
         W  =  W + 1
*
*   UNDERLINE WORD IF IT IS A PALINDROME
         OUTPUT = PALIN(OUTPUT) REPEAT(N,'-') : F(UP.PRINT)
*
*   ADD ONE TO THE PALINDROME COUNT FOR THIS SIZE
         P  =  P + 1                      :   (UP.PRINT)
*
*   ALL WORDS HAVE BEEN PRINTED - PRINT THE COUNTS
```

```
DONE.PRINT   SKIP(1)
         OUTPUT  =  W '□□□'  N  '-LETTER□WORDS'
         OUTPUT  =  IDENT(P,NULL)  '0□□□'  N  '-LETTER'
+                   '□PALINDROMES'      :  S(W.PRINT)
         OUTPUT  =  P  '□□□'  N  '-LETTER□PALINDROMES'
*
*   ADD THESE TOTALS TO THE COUNTS FOR ALL SIZES
         PALINS  =  PALINS + P
W.PRINT  WORDS  =  WORDS + W
         SKIP(2)                        :    (RETURN)
END.PRINT
*
*   MAIN PART OF PROGRAM
*
*   INITIALIZE BY DETERMINING THE PUNCUTATION CHARACTERS
*   AND FORMING A WORD-FINDING PATTERN
         PUNC   =  '□'  TRIM(INPUT)     :   F(ERROR)
         WORD.PAT  =  BREAK(PUNC) . WORD  SPAN(PUNC)
*
*   MAIN READ LOOP - GET THE NEXT RECORD
READ     RECORD  =  TRIM(INPUT)  '□'    :   F(LIST)
*
*   REMOVE ANY INITIAL SPACES OR PUNCTUATION
         RECORD  POS(0)  SPAN(PUNC) = NULL
*
*   GET THE NEXT WORD
NEXTWORD RECORD  WORD.PAT  =  NULL      :   F(READ)
*
*   SAVE LENGTH OF LONGEST WORD IN MAX
         MAX  =  GT(SIZE(WORD),MAX)  SIZE(WORD)
*
*   STORE THE WORD IN THE LIST FOR ITS SIZE
         STORE(WORD)                    :    (NEXTWORD)
*
*   PRINT THE LISTS, SHORTEST ONES FIRST
LIST     N  =  LT(N,MAX)  N + 1         :   F(FINAL)
*
*   IF THERE ARE WORDS OF LENGTH N, PRINT THEM
         (DIFFER($(N 'LIST'),NULL)  PRINT(N,'LIST'))
+                                       :    (LIST)
*
*   PRINT SOME FINAL STATISTICS, PREPARED BY PRINT()
FINAL    OUTPUT = 'TOTAL□NUMBER□OF□WORDS□--□'  WORDS
         OUTPUT = 'TOTAL□NUMBER□OF□PALINDROMES□--□' PALINS
+                                       :    (END)
*
ERROR    OUTPUT = 'NO□DATA'
END
```

If the input to this program were the question

DID THE NAME ADA REFER TO A VARIABLE AT LEVEL 1 OR LEVEL 2

then the output would be as follows.

```
LIST OF 1-LETTER WORDS

A
-
1
-
2
-

3    1-LETTER WORDS
3    1-LETTER PALINDROMES


LIST OF 2-LETTER WORDS

TO
AT
OR

3    2-LETTER WORDS
0    2-LETTER PALINDROMES


LIST OF 3-LETTER WORDS

DID
---
THE
ADA
---

3    3-LETTER WORDS
2    3-LETTER PALINDROMES


LIST OF 4-LETTER WORDS

NAME

1    4-LETTER WORDS
0    4-LETTER PALINDROMES
```

```
LIST OF 5-LETTER WORDS

REFER
-----
LEVEL
-----
LEVEL
-----

3    5-LETTER WORDS
3    5-LETTER PALINDROMES


LIST OF 8-LETTER WORDS

VARIABLE

1    8-LETTER WORDS
0    8-LETTER PALINDROMES


TOTAL NUMBER OF WORDS -- 14
TOTAL NUMBER OF PALINDROMES -- 8
```

7A. ARRAYS

The programming of some problems can be greatly simplified with the use of sets of successively-named variables, such as those described in Chapters 5 and 6. There, indirect referencing was used to refer to variables with some set of names such as LIST1,LIST2,...,$('LIST' N). The variables could be thought of as forming a set because their names were composed of two parts, where one part was common to all names of the set and the other part varied; the variables were said to be successively-named because the varying part was an integer which differed by one for each member of the set. The notion that the variables with names differing in this way were logically associated was, of course, simply a convention adopted by the programmer. But the idea of a set of variables associated together, with the selection of any one of them dependent on the value of an arithmetic expression, is so useful that data structures of this sort are predefined in Snobol, under the name of Arrays. An array is used very much like a set of variables with successive names, except that the convention that the variables constitute a set is not the programmer's alone, but is shared by the Snobol system. Thus it is possible to treat the set of variables as a single aggregate in some cases, and to make reference to specific variables in the set on other occasions.

Creating an Array. An array is created by executing a call to the predefined procedure ARRAY(). The ARRAY() procedure has a single string-valued argument, which in its simplest form is used to specify the number of variables of which the array is to be composed. For example, execution of the rule

```
LIST = ARRAY('1000')
```

causes an array of 1000 variables to be created; this array is returned as the value of the ARRAY() procedure and the entire aggregate is assigned as the value of the variable named LIST.

The variables forming an array are distinct from other variables in that they do not have names which can be written directly in program texts. Rather, they are usually represented in a program text by expressions which are composed of two parts: the first part consists of the name of a variable whose value is the entire "family" of variables that make up the array; the second part, called the "selector," consists of at least one integer-valued expression, called an index, enclosed within square brackets and immediately following the family part of the name. Consecutive integer selectors are assigned to each variable of the array and serve to select a particular variable from the set. Thus variable number three of the 1000-variable array which is the value of LIST may be referred to as LIST[3].

When the rule

```
LIST = ARRAY('1000')
```

is executed, the 1000 variables LIST[1], LIST[2], ..., LIST[1000] become available for use. Each of these variables initially has the null value, like any other variable, when the array is created. These variables may acquire new values by the usual means of assignment, as in the statements

```
LIST[1] = TRIM(INPUT)          : F(DONE)

LIST[1] POS(0) SPAN('□') = NULL
```

and

```
RECORD ANY(VOWELS) . LIST[7] : F(NOVOWEL)
```

Although all variables of an array are often assigned values of the same datatype, there is no requirement that this be done: some may be assigned Strings as values, and some Patterns, for instance; such a variable may even have an Array as its value, including the array of which it is itself a member.

Array Items and Item References. The variables forming an array are called "array items"; references to these variables in program texts, expressions of the form LIST[N], are called "item references." It is important to remember that the variables referred to by these item references do not have names in the form of strings. That is, the string LIST[1] is not the name of variable number one of the array which is the value of LIST. For one thing, such a string cannot be written in a program text to represent a name since it is not in identifier form. Nevertheless, every string is the name of a variable, so the string LIST[1] is indeed the name of some variable, which may be represented in a program text as $'LIST[1]'; however, this variable has no intrinsic connection with any array.

The variables with strings as names are all available to a programmer when execution of a program begins, and are called "natural" variables; in contrast, variables which are array items must be explicitly created by a call to the ARRAY() procedure, and in consequence are called "created" variables. They have names which are not strings -- necessarily, since every possible string is the name of a natural variable. If the name of a variable which is an array item is needed (so that it may be passed as an argument to a procedure, for example), a special kind of non-string Name must be generated by the use of the name operator described toward the end of this chapter.

The family part of an item reference, LIST in the example above, must always be an identifier and must refer to a variable whose value is an array. However, natural variables whose names are not in identifier form, such as the one represented by $(CHAR '*'), and created variables, such as the one represented by LIST[3], may be assigned arrays as values. Special methods, described later in this chapter, must then be used to form references to the items of these arrays. Note that references to all items of an array are always formed with the use of a single name, that of a variable whose value is the array to which they belong.

<u>Comparison with Indirect Referencing.</u> A set of successively-named variables formed with the use of indirect referencing constitutes a sort of simulated array. These simulated arrays have some advantages over the predefined array structures provided by Snobol.

When indirect referencing is used, it is not necessary to specify in advance how many variables will belong to the set. That is, in the loop

```
NLOOP    N  =  N + 1
         OUTPUT  =  TRIM(INPUT)          :  F(ALLGONE)
         $('LIST' N)  =  OUTPUT          :   (NLOOP)
```

the maximum value of N is determined only by the number of data records read, which may vary with each use of the program.

There is also no restriction that N be incremented only by 1 -- any interval may be used, not necessarily the same one on each iteration of the loop. Thus the statement labelled NLOOP above may read

```
NLOOP    N  =  N + 2
```

or

```
NLOOP    N  =  N + SIZE($('LIST' N))
```

or whatever.

Further, there is no necessity to use numeric values at all in forming the varying part of a name. For example, the "successively-named" variables LISTA, LISTB, ..., LISTZ could be used by writing the loop

```
        ALPHA  =  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
        CHARPAT  =  LEN(1) . CHAR
LOOP    ALPHA  CHARPAT  =  NULL          :  F(DONE)
        $('LIST' CHAR) = TRIM(INPUT)     :  S(LOOP)
```

For that matter, there is no need for the variables of a simulated array to have names which are obviously "successive." Thus, the varying part of each name could be formed from a list of words which might have no obvious relation to one another. Using a word as a "selector" of a simulated array item provides much more information than the use of an often arbitrary number. Lastly, no difficulties arise if the "family" part of the names is not in identifier form.

On the other hand, there are some advantages to using the predefined array structure. The principal one is that the array items are recognized as being related by the Snobol system, so the whole aggregate can be assigned as the value of a variable, passed as an argument to a procedure, and so forth. Also, the variables which are array items are distinct from all other variables since they do not have names in the form of strings, so inadvertant conflicts of variable usage are easily avoided; and sometimes an item reference in a program text gives a more intuitive picture of the process being programmed than does an expression involving indirect referencing.

An array is a particularly useful data structure to employ when the numeric order of its items is significant, e.g., when the n-th item of some list is needed. For data which does not lend itself well to being processed in terms of numeric ordering, other types of data structures are probably more useful. Ways of creating data structures of one's own choosing are indicated in the following chapter.

Multi-dimensional Arrays. It is often intuitively useful to think of the items of an array as being arranged in more than the single dimension of the LIST example above. One might want, for example, to simulate the moves on a chessboard by using an 8x8 array which is the value of a variable named BOARD. Such a two-dimensional, 64-item array could be created by executing the rule

```
BOARD = ARRAY('8,8')
```

The first row of the chessboard could then be represented by giving values to the items referred to as BOARD[1,1], BOARD[1,2], ..., BOARD[1,8]. The programmer is of course free to decide which dimension is to be thought of as indicating the rows and which as indicating the columns. If he prefers the opposite convention, then the first row would be the items BOARD[1,1], BOARD[2,1], ..., BOARD[8,1].

Similarly, a three-dimensional tic-tac-toe board having a 5x5 square on each of its three planes could be simulated by using the array created by executing the rule

```
TIC3 = ARRAY('5,5,3')
```

The central cell of this structure is the array item TIC3[3,3,2].

Although it is difficult to symbolize or conceptualize arrays of more than three dimensions, they present no programming problems. For each new dimension, another number within the argument of the ARRAY() procedure is needed for the creation of the array; similarly, another index is needed within the selector to form an appropriate reference for any given array item. There are no limitations on the number of dimensions which an array may have, or on the number of items to be associated with each dimension.

Arrays of many dimensions can be used to arrange data elements which differ from one another along many numeric scales. Each "dimension" is thought of as an "attribute," and a data element is assigned to a particular array item according to the numeric value of all its attributes. The data elements may then be accessed in an orderly manner along each "dimension" of the arrangement.

<u>The ARRAY() Procedure.</u> The predefined procedure ARRAY() requires a single string-valued argument which provides a prototype of the array, specifying (implicitly or explicitly) the number of dimensions the array is to have and the range of index numbers which may be used to select items of this array in each dimension. Unless otherwise specified, it is assumed that the indexing in each dimension starts with 1. However, if the arrays described above as being the values of LIST, BOARD, and TIC3 were to be indexed from zero instead of from one, but were still to have the same number of items as before, this could be specified by executing the rules

```
LIST  =  ARRAY('0:999')
BOARD  =  ARRAY('0:7,0:7')
TIC3  =  ARRAY('0:4,0:4,0:2')
```

The colon within the argument is used to separate the lowest index number from the highest index number for each dimension; the comma is used to separate the different dimensions from one another; no embedded blanks are permitted.

Negative numbers may be used within the prototype of an array, and consequently within the selectors of its items. Execution of the rule

```
NEGARR  =  ARRAY(-50:-5)
```

creates a 46-element array whose items may be referred to as NEGARR[-50], NEGARR[-49], ..., NEGARR[-5]. (Note that these references are arranged, as always, in ascending arithmetic order.)

Information about the range of index numbers in each dimension may be provided in terms of any expressions which give the desired numbers when evaluated. These indices may be positive, negative, or zero, but the upper bound for any dimension must always be greater than or equal to the corresponding lower bound; consequently an array must always be composed of at least one item. Thus the rules

```
ARRAY1 = ARRAY(SIZE(WORD1) ',' SIZE(WORD2))
ARRAY2 = ARRAY(M1 ':' N1 ',' M2 ':' N2)
ARRAY3 = ARRAY(A + B ',' C + D)
```

may each specify the creation of a two-dimensional array, if the expressions within the argument of each ARRAY() procedure have appropriate numeric values at the time the rules are executed.

Note that the commas and colons are placed within quotes to indicate that they are literal characters to be concatenated into the string being formed to provide the single argument. If the commas were not placed within quotes, each comma would indicate the presence of another argument for the ARRAY() procedure; all arguments after the first would be evaluated but otherwise ignored, since ARRAY() requires only one argument. The array procedure returns as its value an array created to the specifications of its argument. Thus the variables named ARRAY1, ARRAY2, and ARRAY3 in the above example would all be assigned values of datatype Array.

Selectors. Selectors may also consist of any expressions which yield the desired index (or indices) when evaluated. Thus

```
LIST[1]
LIST[A + B]
LIST[SIZE(TRIM(CARD))]
LIST[$LIST[2]]
LIST[LIST[LIST[2]]]
```

are all item references which may be used to refer to variable number one of the array which is the value of LIST if the expressions A + B and SIZE(TRIM(CARD)) and $LIST[2] and LIST[LIST[2]] all have the value 1 when the rules in which the above expressions appear are executed.

Although the prototype of the array is expressed as a string, note that the selector of an item reference is not; rather the expressions representing the indices are separated by commas, much like the arguments of a procedure reference. Thus BOARD[X,Y] is an appropriate item reference for a two-dimensional array, while BOARD[X ',' Y], which specifies a non-integer index, is not. An execution-time error will occur if a non-integer results from the evaluation of the index for any dimension, or if the number of dimensions indicated by the selector is not the same as the number specified by the prototype for that array.

Failure of an Item Reference. An attempt to evaluate an item reference may fail, causing failure of the rule in which the evaluation occurs. An item reference fails when its family part refers to a variable whose value is an array, but its selector yields an index for any dimension which falls outside the range specified by the prototype of that array. Thus the rule

```
        OUTPUT  =  LIST[N]              :  F(DONE)
```

will fail and send control to DONE for values of N which are less than 1 or greater than 1000 for the value of LIST described at the beginning of this chapter. The simple two-statement loop

```
LOOP    N  =  N + 1
        OUTPUT  =  LIST[N]              :  S(LOOP)   F(DONE)
```

can therefore be used to print the values of all items of the array referred to by LIST (provided these values are all strings). Here the fact that the item reference can cause failure of the rule eliminates the need for a statement of the form

```
        N  =  LT(N,1000)  N + 1        :  F(DONE)
```

to terminate the loop and so somewhat simplifies the programming. (Note that the values of all the items of an array cannot be printed by a rule of the form OUTPUT = LIST, since LIST has an array as its value, and only strings can be printed.)

Often reliance on the failure of an item reference rather than on the failure of some test procedure does not simplify the programming and may lead to logical errors. For example, the loop

```
FILL1   N  =  N + 1
        LIST[N]  =  TRIM(INPUT)        :  F(FULL)  S(FILL1)
```

will fail and send control to FULL (1) when the value of N becomes greater than 1000 or (2) when the data is exhausted, without making the (often necessary) distinction between the two cases. The fact that an item reference can cause failure of the rule must always be kept in mind to prevent the writing of rules which may fail for more than one reason.

<u>Special Problems Concerning Item References.</u> It is possible to assign an array as the value of a variable whose name cannot be represented in identifier form, either because it contains impermissible characters, as in

```
        $'A/1'  =  ARRAY('1000')
```

or because it is a created variable, as in

```
        LIST[1]  =  ARRAY('1000')
```

or because it is unknown, as in

```
        $WORD  =  ARRAY('1000')
```

Although each of the above rules creates an array of 1000 items and assigns it as the value of some variable as in all previous examples, the items of these arrays may not be referred to in the usual manner, since there is a restriction that the family part of an item reference must be a name in identifier form. Thus if one attempts, for the first two cases above, to write rules of the form

```
        $'A/1'[1]  =  TRIM(INPUT)
```

and

```
    .   LIST[1][1]  =  TRIM(INPUT)
```

then compile-time errors result.

Writing, for the third case, the rule

```
        $WORD[1]  =  TRIM(INPUT)
```

does not result in a compile-time error, but does not give the desired result either. Here, the operand of the indirect referencing operator is not the variable WORD, as is desired, but rather the item reference WORD[1]. The evaluation of WORD[1] should cause an execution-time error, since the variable WORD was intended as the operand of the indirect referencing operator, and thus its value should be a string or a Name, not an array.

All of these cases may be taken care of by simply assigning each array to another variable, one whose name may be represented by an identifier. Each of the erroneous rules presented before can thus be replaced by a pair of rules, such as the following:

```
        TEMP1  =  $'A/1'
        TEMP1[1]  =  TRIM(INPUT)

        TEMP2  =  LIST[1]
        TEMP2[1]  =  TRIM(INPUT)

        TEMP3  =  $WORD
        TEMP3[1]  =  TRIM(INPUT)
```

Note that assigning an array to a second variable does not cause a new array to be created, but merely allows two (or more) variables to have the same array as their values.

The ITEM() Procedure. The ITEM() procedure provides another method of referring to the items of an array when the array has been assigned to a variable whose name cannot be written in identifier form. The ITEM() procedure, like the APPLY() procedure described in Chapter 6, has a varying number of arguments, usually one more than the number of dimensions of the array involved. The first argument must be an expression whose value is an array; the remaining arguments may be any integer-valued expressions, usually one for each dimension of the array, given in the appropriate order. ITEM() returns as its value (by NRETURN) the variable specified by using its first argument to indicate a family and its remaining arguments together to form a selector. Thus the expression ITEM(LIST,1) is equivalent to the expression LIST[1], and ITEM(BOARD,8,8) is equivalent to BOARD[8,8]. More usefully, the rules

```
ITEM($'A/1',1)  =  TRIM(INPUT)

ITEM(LIST[1],1)  =  TRIM(INPUT)
```

and

```
ITEM($WORD,1)  =  TRIM(INPUT)
```

could all be used in place of the rules involving TEMP1, TEMP2, and TEMP3, above.

A procedure reference to ITEM() may be written wherever an item reference may appear. Thus the rule

```
OUTPUT  =  TIC3[X,Y,Z]
```

may be written as

```
OUTPUT  =  ITEM(TIC3,X,Y,Z)
```

with the same effect. ITEM() fails, in just the way that an item reference fails, if the index for any dimension within the selector which is formed falls outside the range specified by the prototype of the array involved.

Although the selector part of an item reference must consist of a list of indices separated by commas, as in TIC3[X,Y,Z], and may not be expressed as a concatenated string, as in TIC3[X ',' Y ',' Z], the ITEM() procedure allows the selector to be represented by either method and even by combinations of the two. Furthermore, ITEM() does not require that the proper number of index expressions be present in its arguments. It uses only as many indices as are appropriate for the array given as its first argument; it assumes the value zero for missing indices, and evaluates but otherwise ignores the expressions for extra indices. Thus the number of arguments with which ITEM() may be called can vary not only with the number of dimensions of the array being indexed but also with the choice of representation for each index. The four-argument call

```
ITEM(TIC3,X,Y,Z)
```

has the same effect as either of the three-argument calls

```
ITEM(TIC3,X ',' Y,Z)
```

or

```
ITEM(TIC3,X,Y ',' Z)
```

or the two-argument call

```
ITEM(TIC3,X ',' Y ',' Z)
```

Each returns the item TIC3[X,Y,Z] as its value. The importance of this feature is illustrated by an example at the end of this chapter.

<u>The PROTOTYPE() Procedure.</u> The PROTOTYPE() procedure can accept as its single argument any expression whose value is of datatype Array, and returns as its value a string giving the prototype of that array. This prototype will be the same as the one specified in the call to the ARRAY() procedure which caused the array to be created, except that the lower bound for each dimension is always explicitly expressed, and the integers specifying the bounds are in canonical form (a sign retained only for negative numbers, leading zeroes suppressed, and zero represented by the single character 0). Thus if the rules

```
BOARD  =  ARRAY('08,08')
TIC3  =  ARRAY('5,5,3')
LIST  =  ARRAY('0:999')
NEGARR  =  ARRAY('-50:+5')
```

have been executed, then execution of the rules

```
OUTPUT  =  PROTOTYPE(BOARD)
OUTPUT  =  PROTOTYPE(TIC3)
OUTPUT  =  PROTOTYPE(LIST)
OUTPUT  =  PROTOTYPE(NEGARR)
```

will cause the strings

```
1:8,1:8
1:5,1:5,1:3
0:999
-50:5
```

to be printed. Such strings may be investigated with a pattern-matching rule to determine the structure of the array; this may be useful in cases where the dimensions have not been given as literals within the ARRAY() procedure's argument, but have been specified by more complicated expressions or supplied from the data. For example, an array could be created by executing the rule

```
BOXES  =  ARRAY(DIM1 ',' DIM2)
```

Although the value of BOXES appears to be a two-dimensional array, this is not necessarily the case since the values of DIM1 and DIM2, perhaps acquired from the input file, may contain any number of commas, each indicating another dimension. The number of dimensions of this array may be determined by the following simple program segment which searches the string returned by PROTOTYPE() to determine how many commas it contains; the number of dimensions is always one more than the number of commas.

```
         STRING  =  PROTOTYPE(BOXES)
LOOP     STRING BREAK(',')  ','  REM . STRING  :  F(DONE)
         COMMA  =  COMMA + 1             :    (LOOP)
DONE     DIMENS  =  COMMA + 1
```

The PROTOTYPE() procedure may also take a pattern or a Name or a structure of programmer-defined datatype as its argument. A description of the use of PROTOTYPE() with an argument of one of these datatypes may be found in Appendix A, section II.B.

<u>The TYPE() Procedure.</u> The TYPE() procedure is one which will accept any expression as its single argument. If the value of its argument is of a predefined datatype, the procedure returns as its value a string specifying that datatype; if the value is of a programmer-defined datatype, the string DATA is returned. For example, execution of the rule

```
         OUTPUT  =  TYPE('SASSAFRAS')
```

will print STRING while execution of the rule

```
         OUTPUT  =  TYPE(ARB)
```

(if ARB still has its predefined value) will produce PATTERN; the rule

```
         OUTPUT  =  TYPE(LIST) '□□□□'  TYPE(LIST[1])
```

will print ARRAY followed by INTEGER.

TYPE() is often used to test whether or not some variable has a value of the expected datatype before some process is allowed to continue. It is particularly useful for testing whether values passed to the formal variables of a procedure are of the correct datatype, and for insuring that all values assigned to OUTPUT are of datatype String or datatype Integer.

The short loop presented earlier to print the values of all items belonging to a specified array may be amended with the use of the TYPE() procedure to first test the datatype of each value and then to print only those of datatype String or Integer. This amended program segment uses indirect referencing within the go-to to transfer to a label representing the type of the value being processed. If the value is of datatype String or Integer then the value is printed; if it is of any other datatype, a message regarding its type is printed. In either case, the value of the selector is printed first so that the particular item whose value is being printed or described may be identified. The PROTOTYPE() procedure is used in the first statement to insure that a one-dimensional array is being processed, and to determine the lower bound of this array.

```
*  TEST WHETHER ARRAY IS 1-DIMENSIONAL AND FIND LOWER BOUND
         PROTOTYPE(LIST) BREAK(':') . N ':'
+                 SPAN('-0123456789') RPOS(0)   :  F(ERROR)
*
*  LOOP TO PRINT ALL VALUES WHICH ARE STRINGS
*  IF LIST[N] EXISTS, GO TO THE STATEMENT LABELLED BY THE
*    TYPE OF ITS VALUE
*
LOOP     LIST[N]                  :  F(DONE) S($TYPE(LIST[N]))
*
STRING
INTEGER  OUTPUT  =  N  '□□'    LIST[N]   :    (INC)
REAL
PATTERN
ARRAY
NAME
CODE
DATA     OUTPUT = N '□□THIS□ITEM□IS□OF□TYPE□' TYPE(LIST[N])
*
*  INCREMENT INDEX TO GET NEXT ITEM
INC      N  =  N + 1                       :    (LOOP)
```

The labels provided in the program text (with the exception of LOOP and INC) are exactly the strings returned by the TYPE() procedure. All have been mentioned except CODE, which is described briefly in Appendix A, section II.C. These labels provide an exhaustive list of the string values which TYPE() can return.

The program text may appear strange because of the number of null rules. Since the statements labelled STRING and INTEGER both need the same rule, it has been written only once in the second of these statements, the one labelled INTEGER. If control is sent to the statement labelled STRING, it is sent on immediately to the statement labelled INTEGER where the rule which calls for printing is executed, since the statement labelled STRING has no rule and no go-to to be processed. Similarly, since the statements labelled REAL, PATTERN, ARRAY, NAME, CODE, and DATA all need the same rule, it is written only once in the last of these statements, the one labelled DATA.

The evaluation rule LIST[N] is needed in order for failure of the item reference to be detected. If this evaluation rule were omitted and the statement consisted solely of the go-to

```
                                        :    ($TYPE(LIST[N]))
```

then there would be no way to terminate the loop gracefully, and an execution-time error would result when the item reference failed within the go-to because the value of N became too large.

Procedure to Return a Selector. There are a number of processes concerning arrays which it would be convenient to express as programmer-defined procedures since they are so frequently needed. For example, one often wants to know the selector associated with the first null-valued item of an array so that this item may be given another value. The following SELECT() procedure fails if there are no null-valued items, or succeeds and returns the selector of the first null item as its value. It works for any one-dimensional array, and uses PROTOTYPE() as before to test that the array is one-dimensional and to find its lower bound. The single argument of SELECT() may be any expression whose value is an array.

```
         DEFINE('SELECT(ARR1)N','PR.SEL')    : (END.SELECT)
*  TEST WHETHER FIRST ARGUMENT HAS AN ARRAY AS ITS VALUE
PR.SEL   IDENT(TYPE(ARR1),'ARRAY')      :  F(SEL.ER1)
*
*  TEST WHETHER ARRAY IS 1-DIMENSIONAL AND FIND LOWER BOUND
         PROTOTYPE(ARR1)  BREAK(':') . N  ':'
+                 SPAN('-0123456789') RPOS(0)   :   F(SEL.ER2)
*
*  TEST WHETHER THIS ITEM HAS A NULL VALUE
*    RETURN ITS SELECTOR IF IT DOES
OUT.SEL  SELECT  =  IDENT(ARR1[N])  N  :  S(RETURN)
*
*  ELSE INCREMENT INDEX TO LOOK AT THE NEXT ITEM
         N  =  N + 1
*
```

```
*  TEST WHETHER THIS SELECTOR IS OUTSIDE THE BOUNDS OF ARRAY
*    IF SO, THIS ARRAY CONTAINS NO NULL-VALUED ITEMS
          ARR1[N]                  :  F(FRETURN)  S(OUT.SEL)
*
*  PRINT ERROR MESSAGES AND STOP
SEL.ER1   OUTPUT  =  'ARGUMENT□OF□SELECT()□NOT□AN□ARRAY'
+                                       :    (END)
SEL.ER2   OUTPUT  =  'ARRAY□PASSED□IS□NOT□1-DIMENSIONAL'
+                                       :    (END)
END.SELECT
```

When this procedure is used, as in the statements

```
          Q  =  SELECT(LIST)            :  F(FULL)
          LIST[Q]  =  WORD
```

or, equivalently,

```
          LIST[SELECT(LIST)]  =  WORD   :  F(FULL)
```

the procedure reference SELECT(LIST) causes the value of the variable LIST to be assigned as the value of the formal variable ARR1 internal to the procedure call. If the value of LIST is an array, as is intended, this means that the two variables LIST and ARR1 have the same array as their values. The first statement of the procedure body tests the value of ARR1 to insure that it is indeed of datatype Array before proceeding; the second statement further tests that this array is one-dimensional. If either test fails, an appropriate error message is written and the procedure ends execution of the program. If ARR1 has as value a one-dimensional array, then the lower bound of this array is assigned to the internal variable N. Then the evaluation rule ARR1[N] is executed; this refers to the same array item as LIST[N] since ARR1 and LIST both have the same array as value. This rule fails only when the value of N exceeds the upper bound of the array, which occurs only when all items of the array have already been considered. Hence if the rule fails the array contains no null-valued items and an FRETURN is taken. If the rule ARR1[N] does not fail then the value of ARR1[N] is tested to see whether or not it is null; if it is null then the result variable SELECT is assigned the value of N so that this value is returned as the value of the procedure call.

Procedure to Interchange Two Arrays. There are some procedures which need to be passed the name of the variable whose value is an array, rather than the array which is the value of that variable. Consider two variables named X and Y; the value of X is a one-dimensional array of 10 items, while the value of Y is a one-dimensional array of 100 items. The programmer wishes to cause the value of X to be the 100-item array, and the value of Y to be the 10-item array. Before performing this swap he wants to be sure that X and Y are both one-dimensional arrays. This process may be performed with the side-effect procedure SWAP() which has three arguments: the names of the two variables whose values are arrays, and the number of dimensions these arrays are both to have. Each name is presented as a string which will be passed to the procedure body to be used as the operand of the indirect referencing operator to return a variable; the number of dimensions may be expressed as any numeric-valued expression. The SWAP() procedure uses the REPEAT() procedure, described at the beginning of Chapter 6, to build a pattern which can be used to determine whether or not the prototype of each array has the specified number of dimensions.

```
         DEFINE('SWAP(A,B,N)PAT1,PAT2,TEMP','PR.SWAP')
+                                              :    (END.SWAP)
*
*  TEST WHETHER THE FIRST TWO ARGUMENTS ARE ARRAY-VALUED
PR.SWAP  IDENT(TYPE($A),'ARRAY')               :  F(SWAP.ER1)
         IDENT(TYPE($B),'ARRAY')               :  F(SWAP.ER2)
*
*  TEST WHETHER BOTH ARRAYS ARE OF THE SPECIFIED DIMENSION
*    BUILD A PATTERN USING REPEAT() TO LOOK FOR THE RIGHT
*    NUMBER OF COLONS WITHIN THE PROTOTYPE
         PAT1  =  BREAK(':')   ':'
         PAT2  =  POS(0)   REPEAT(PAT1,N)
+                SPAN('-0123456789')   RPOS(0)
         PROTOTYPE($A)   PAT2                  :  F(SWAP.ER3)
         PROTOTYPE($B)   PAT2                  :  F(SWAP.ER4)
*
*  BOTH ARE ARRAYS OF THE SPECIFIED DIMENSION
*    SWAP THEM AND RETURN
         TEMP  =  $A
         $A  =  $B
         $B  =  TEMP                           :    (RETURN)
*
*  PRINT ERROR MESSAGES AND FAIL
SWAP.ER1 OUTPUT = 'FIRST□ARGUMENT□OF□SWAP()□NOT□AN□ARRAY'
+                                              :    (FRETURN)
SWAP.ER2 OUTPUT = 'SECOND□ARGUMENT□OF□SWAP()□NOT□AN□ARRAY'
+                                              :    (FRETURN)
SWAP.ER3 OUTPUT = 'FIRST□ARRAY□NOT□OF□DIMENSION□'  N
+                                              :    (FRETURN)
SWAP.ER4 OUTPUT = 'SECOND□ARRAY□NOT□OF□DIMENSION□'  N
+                                              :    (FRETURN)
END.SWAP
```

A call on this procedure to do the swapping of the values of X and Y as described above could have the form

```
        SWAP('X','Y',1)                    :  F(ERROR)
```

Since the formal variables A and B never appear within the procedure body except preceded by a $ operator, it would seem at first that the call SWAP(X,Y,1) could be used instead of the call SWAP('X','Y',1) and all the indirect referencing operators removed from the procedure body, since the expression $'X' is indeed equivalent to X in all cases. If this were done, however, the <u>value</u> of X would be used wherever the formal variable A occurred in the procedure body. While the expressions TYPE(A) and PROTOTYPE(A), where A has as its value the same array that is the value of X, will indeed work as desired, rules of the form A = B and B = TEMP, will not produce the desired effect. Execution of the rule A = B would cause the formal variable A to be assigned the array which is the value of Y, and the rule B = TEMP would cause the formal variable B to be assigned the array which is the value of X. Thus the values of A and B, which are internal to the procedure call only, would be swapped rather than the values of the external variables X and Y. In order to change the value of X, the string which is its name must be passed and a rule of the form $A = $B must be used, since the expression $A, in this case, will return the external variable X to which an assignment can then be made.

<u>The Name Operator.</u> Since array items do not have strings as names, problems arise when one tries to pass the name of an array item to a procedure. If the 100-item array described above had been assigned to the created variable LIST[1] instead of to the natural variable Y, and its value was to be swapped with that of the 10-item array which is the value of X, then a call of the form

```
        SWAP('X','LIST[1]',1)
```

would not produce the desired effect since the string LIST[1] is the name of a natural variable, and thus cannot be the name of a created variable.

The problem of passing the name of a created variable is solved with the use of the name operator, a unary operator whose symbol is a period. This operator takes any variable as its operand and returns as its value a special object of datatype Name which is a name for that variable. Thus the name of the created variable LIST[1] may be represented as .LIST[1], so a procedure call of the form

```
        SWAP('X',.LIST[1],1)
```

would produce the desired effect.

If the operand of the name operator is a natural variable, which thus has a string name like X for example, then the Name .X provides still a different name by which to refer to that variable. The two names always refer to the same variable, and can be used interchangeably. The application of the $ operator to an operand of datatype Name gives the same effect as its application to a string-valued operand: the variable named by the operand is returned. Thus the call

```
        SWAP(.X,.LIST[1],1)
```

could be used as well. The only necessity for the use of the name operator arises when names of created variables must be passed to and from procedures. Note that objects of datatype Name cannot be printed.

As an example of an application in which a Name is to be returned by a procedure, consider an amended version of the SELECT() procedure, presented earlier in this chapter, which would return the Name of the first null-valued item of an array rather than its selector. This amended procedure, called STEP(), is presented below; the entire procedure body is the same as that of SELECT() except for the statement labelled OUT.STEP in which the result variable is assigned a value of datatype Name.

```
*   PROCEDURE TO RETURN NAME OF FIRST NULL-VALUED ITEM
*
        DEFINE('STEP(ARR1)N','PR.STEP')  :    (END.STEP)
*
*   TEST WHETHER FIRST ARGUMENT HAS AN ARRAY AS ITS VALUE
PR.STEP  IDENT(TYPE(ARR1),'ARRAY')      :  F(STEP.ER1)
*
*   TEST WHETHER ARRAY IS 1-DIMENSIONAL AND FIND LOWER BOUND
        PROTOTYPE(ARR1)  BREAK(':') . N  ':'
+                SPAN('-0123456789') RPOS(0)  : F(STEP.ER2)
*
*   TEST WHETHER THIS ITEM HAS A NULL VALUE
*     RETURN THE NAME OF THIS ITEM IF IT DOES
OUT.STEP STEP = IDENT(ARR1[N],NULL) .ARR1[N]  :   S(RETURN)
*
*   ELSE INCREMENT INDEX TO LOOK AT NEXT ITEM
        N  =  N + 1
*
```

```
*   TEST WHETHER THIS SELECTOR IS OUTSIDE THE BOUNDS OF ARRAY
*     IF SO, THIS ARRAY CONTAINS NO NULL-VALUED ITEMS
         ARR1[N]                : F(FRETURN)  S(OUT.STEP)
*
*   PRINT ERROR MESSAGES AND STOP
STEP.ER1  OUTPUT = 'ARGUMENT□OF□FIND()□NOT□AN□ARRAY' : (END)
STEP.ER2  OUTPUT = 'ARRAY□PASSED□IS□NOT□1-DIMENSIONAL':(END)
END.STEP
```

The rule

```
        $STEP(LIST)  =  WORD            :  F(FULL)
```

may be used to assign the value of WORD to the first null-valued item of the array which is the value of LIST. Execution will cease if the value of LIST is not a one-dimensional array (in which case an error message is printed). The procedure call will fail if there are no null-valued items remaining within the array. If the procedure call succeeds it returns the Name of the first null-valued item; this Name is used as the operand of the $ operator which returns the needed variable.

Alternatively, an NRETURN could be used to cause the procedure to return a variable rather than an object of datatype Name, but the name operator would still be needed within the procedure body. If the statement labelled OUT.STEP were written as

```
OUT.STEP STEP = IDENT(ARR1[N],NULL) .ARR1[N] : S(NRETURN)
```

then the procedure call would have the form

```
        STEP(LIST)  =  WORD             :  F(FULL)
```

since the value returned by STEP() is the variable needed for assignment.

<u>Forming all Selectors of an Array.</u> Whenever the STEP() procedure is called, it always starts by investigating the "first" item of a one-dimensional array, that is, the one whose selector is formed by using the lower bound of the array as its single index. The procedure continues to form new selectors by adding one to the value of this index until a null value is found, or until an attempt is made to increase the index beyond the upper bound of the array; if this happens, then every selector of the array has been used. Since the STEP() procedure has been written to process one-dimensional arrays only, the method it uses for determining all selectors of an array is very simple. The

process of determining all selectors becomes more complicated when an array is multi-dimensional.

A general purpose method which would work for an array of any number of dimensions could be described as follows. Start with a selector formed by using the lower bound of each dimension as its index; this information may be obtained from the prototype of the array. (For example, the initial selector of an array whose prototype is 0:2,1:10,1:10 is 0,1,1.) Subsequent selectors are formed by adding one to the index of the last (rightmost) dimension until the upper bound for that dimension is reached (just as for a one-dimensional array), while keeping all other indices constant. When the upper bound of the last index is reached, reset that index to its lower bound and increment the index of the penultimate dimension by one. For this value of the next-to-the-last index, run through all values of the last index again, resetting when the upper bound is reached. Repeat this process for all values of the penultimate dimension, then reset the this index to its lower bound and begin incrementing the index of the antipenultimate dimension, repeating the previously described processes for each of its values, etc. Proceed until the index of the first dimension has reached its upper bound; then, all selectors of the array have been formed.

If the process just described is applied to a three-dimensional array whose prototype is 1:3,1:2,1:2, the following selectors will be formed in the indicated "numeric" order.

(1.)	1,1,1	(5.)	2,1,1	(9.)	3,1,1
(2.)	1,1,2	(6.)	2,1,2	(10.)	3,1,2
(3.)	1,2,1	(7.)	2,2,1	(11.)	3,2,1
(4.)	1,2,2	(8.)	2,2,2	(12.)	3,2,2

It is easily seen from this display that the rightmost index does indeed vary most often, while the leftmost index is never reset but goes through its range of values only once. The process could be described just as easily with the leftmost index varying most often, but the order in which the particular selectors are formed is immaterial since the same process may be used whenever all items of an array are to be considered. Thus if all items are assigned values by the method just described and later the same method is used to print the values, then the values will be printed in whatever order they were assigned. Since there are many applications in which all items of an array must be considered, it is convenient to express this process in terms of a procedure.

Procedure to Return the "Next" Selector. Presented below is a programmer-defined procedure, NEXT(), which requires two strings as arguments: the first represents a current selector and the second the prototype of the array whose "next" selector is to be formed; this selector is returned in the form of a string as the value of the NEXT() procedure. Here "next" is used to mean the selector which follows in the order described in the preceding section. The NEXT() procedure fails when there is no next selector, for example, when the current selector passed as its argument is the last in the order described above.

```
*   PROCEDURE TO RETURN THE "NEXT" SELECTOR
*
          DEFINE('NEXT(SEL,PROTO)INDEX,LB,UB','PR.NEXT')
*
*   PATTERN FOR TEARING SELECTOR APART INTO ITS INDICES
*   ASSIGN THIS PATTERN TO THE MAIN-PROGRAM VARIABLE SEL.PAT
          SEL.PAT = (',' | NULL) SPAN('-0123456789') . INDEX
+                   RPOS(0)
*
*   PATTERN FOR TEARING PROTOTYPE APART TO FIND LOWER AND
*     UPPER BOUNDS
*   ASSIGN THIS PATTERN TO THE MAIN-PROGRAM VARIABLE PROT.PAT
          PROT.PAT = (',' | NULL) SPAN('-0123456789') . LB
+           ':' SPAN('-0123456789') . UB RPOS(0) : (END.NEXT)
*
*   FIND RIGHTMOST INDEX OF THE SELECTOR STRING AND REMOVE
*     FAIL IF NO MORE INDICES TO BE FOUND
PR.NEXT  SEL  SEL.PAT  =  NULL          :  F(FRETURN)
*
*   FIND LOWER & UPPER BOUNDS FOR THIS DIMENSION
          PROTO  PROT.PAT  =  NULL
*
*   INCREMENT INDEX IF IT IS LESS THAN THE UPPER BOUND
          INDEX = LT(INDEX,UB) INDEX + 1  :  F(RESET.NEXT)
*
*   FORM NEXT SELECTOR STRING BY CONCATENATION
          NEXT = IDENT(SEL,NULL) INDEX ',' NEXT : S(RET.NEXT)
          NEXT  =  SEL ',' INDEX ',' NEXT
*
*   REMOVE SPURIOUS FINAL COMMA FROM SELECTOR STRING
RET.NEXT NEXT  ',' RPOS(0)  =  NULL     :  (RETURN)
*
*   RESET THIS INDEX TO ITS LOWER BOUND, CONCATENATE IT TO
*     THE SELECTOR STRING BEING FORMED AND PROCEED TO WORK
*     ON THE NEXT INDEX
RESET.NEXT
          NEXT  =  LB ',' NEXT          :   (PR.NEXT)
END.NEXT
```

Note that the NEXT() procedure returns a string as its value. Thus the selector represented by that string cannot be used within an item reference, where only a selector list is appropriate, but may be used as the second argument of the ITEM() procedure, as in the rule

```
        OUTPUT = ITEM(LIST,NEXT(SELECT,PROTOTYPE(LIST)))
```

where the value of SELECT is a string representing the last-used selector. If the ITEM() procedure were not defined to accept a string as its second argument, it would not be possible to write a useful, general purpose NEXT() procedure to work on an array with any number of dimensions.

NEXT() was devised for the purpose of returning all successive selectors of an array, each call to NEXT() returning the next selector until a failure transfer is executed. The loop shown below uses the NEXT() procedure in this way. The INIT() procedure which precedes the loop provides a string to be used as the initial value of SELECT; INIT() takes a prototype as its argument and returns the "first" selector of an array described by that prototype.

```
         DEFINE('INIT(PROTO)LBPAT,LB','PR.INIT')
*
*  SET UP PATTERN TO FIND LOWER BOUND FOR EACH DIMENSION
*  ASSIGN THIS PATTERN TO THE MAIN-PROGRAM VARIABLE LB.PAT
         LBPAT = BREAK(':') . LB ':' (BREAK(',') ',' | REM)
+                                               :   (END.INIT)
*
*  USE THIS PATTERN TO FIND NEXT LOWER BOUND
PR.INIT  PROTO   LB.PAT   =   NULL              :   F(RET.INIT)
*
*  FORM INITIAL SELECTOR STRING BY CONCATENATION
         INIT    =  INIT ',' LB                 :   (PR.INIT)
*  REMOVE SPURIOUS INITIAL COMMA AND RETURN
RET.INIT INIT    ','   =   NULL                 :   (RETURN)
END.INIT
*
*  LOOP TO PRINT ALL SELECTORS OF LIST
         SELECT   =   INIT(PROTOTYPE(LIST))
LOOP     OUTPUT   =   ITEM(LIST,SELECT)
         SELECT   =   NEXT(SELECT,PROTOTYPE(LIST))
+                                               :   S(LOOP)
```

Since NEXT() is meant to be used in this and similar ways, it has no special provision for dealing with selector strings passed as the first argument which fall outside the range of the array; such provisions could be added to make the procedure more generally useful.

Procedure to Return a Copy of any Array. It is often necessary to make a copy of an array, rather than merely assigning the same array as the value of more than one variable, so that changes in the values of the copy can be made without affecting the original. To make a copy of an array means to create a new array with the same prototype as that of the original, and to assign to each of its items the same value as that of the corresponding item in the original array. The following COPY() procedure returns as its value a copy of any array; it requires only one argument, which may be any expression whose value is the array to be copied -- this array may have any number of dimensions. The COPY() procedure invokes the INIT() procedure to form the initial selector string, and the NEXT() procedure to insure that all items are considered and hence copied; both of these procedures are described in the preceding section. A call to the COPY() procedure fails, causing an error message to be printed, only if its argument is not of datatype Array.

```
*  PROCEDURE TO RETURN A COPY OF ANY ARRAY
*
        DEFINE('COPY(ARR1)SELECT,P','PR.COPY') : (END.COPY)
*
*  TEST WHETHER ARGUMENT IS AN ARRAY
PR.COPY  IDENT(TYPE(ARR1),'ARRAY')       :  F(COPY.ER1)
*
*  CREATE A NEW ARRAY WITH PROTOTYPE OF ARGUMENT
*    AND ASSIGN IT AS THE VALUE OF THE RESULT VARIABLE
        P  =  PROTOTYPE(ARR1)
        COPY  =  ARRAY(P)
*
*  CALL INIT() TO RETURN THE FIRST SELECTOR OF THIS ARRAY
        SELECT  =  INIT(P)
*
*  COPY VALUE OF NEXT ITEM OF ARRAY, USING ITEM()
COPY.COPY
+       ITEM(COPY,SELECT)   =  ITEM(ARR1,SELECT)
*
*  CALL NEXT() TO RETURN THE NEXT SELECTOR OF THIS ARRAY
*    IF NO NEXT SELECTOR, RETURN
        SELECT  =  NEXT(SELECT,P)       :  S(COPY.COPY)
+                                          F(RETURN)
COPY.ER1 OUTPUT = 'ARGUMENT□OF□COPY□NOT□AN□ARRAY'
+                                       :  (FRETURN)
END.COPY
```

Appendix A. SUMMARY OF PREDEFINED PROCEDURES

I. PROGRAM PROCEDURES are used by the programmer as basic operations in constructing programs.

A. Test Procedures

1. General Comparison

IDENT()
DIFFER()

2. String Comparison

LGT()

3. Arithmetic Comparison

EQ()
NE()
GT()
GE()
LT()
LE()

B. Result Procedures

1. Pattern Construction

ANY()
NOTANY()
SPAN()
BREAK()
LEN()
TAB()
RTAB()
POS()
RPOS()
ARBNO()

2. String Operation

TRIM()

C. Data Procedures

1. Structure Creation

ARRAY()

2. Field Selection

PARAM()
FIRST()
REST()
LEFT()
RIGHT()
FAMILY()
SELECTOR()

II. SYSTEM PROCEDURES are used to communicate instructions and requests to the Snobol system.

A. Declarations

1. Programmer-defined Procedures

DEFINE()

2. Programmer-defined Datatypes

DATA()

B. Access to System Information

1. Attributes of Objects

SIZE()
DATATYPE()
TYPE()
PROTOTYPE()

2. Execution Information

ALPHABET()
DATE()
CLOCK()
TIME()
STCOUNT()
STLIMIT()

MAXLNGTH()
FNCLEVEL()
NEXTVAR()

C. Requests for System Actions

1. Special Execution

ITEM()
APPLY()
IF()

2. Set Mode of Pattern-Matching

ANCHOR()

3. Datatype Conversion

CONVERT()
CODE()

D. Input/Output Procedures

1. File Association

INPUT()
OUTPUT()
DETACH()

2. Requests for File Actions

ENDGROUP()
REWIND()
REMARK()
FREEZE()

3. Tests of File Position

EORLEVEL()
EOI()

The foregoing classification scheme is introduced as an aid to understanding the purpose and use of the various predefined procedures; the particular classes differentiated play no part in the definition of Snobol, and other classifications could be devised. Notice that most programmer-defined procedures declared by DEFINE() constitute extensions of the classes of test procedures and result procedures, and that those declared by DATA() constitute extensions of the classes of structure creation and field selection procedures.

In the descriptions which follow, each predefined procedure is shown along with the kind of value required for its argument(s) and the kind of value it returns. There are no syntactic restrictions on the form of arguments; since all arguments are passed "by value" in Snobol procedure calls, actual arguments may be written as arbitrarily-complicated expressions. There are, however, semantic restrictions on the values resulting from evaluation of actual arguments, defined in terms of "datatypes." Every data object known to a Snobol program is of datatype String, Integer, Pattern, Real, Array, Name, Code, or a programmer-defined datatype. Each procedure is shown here with the datatypes it will accept; a call of a procedure using an argument with a wrong datatype will result in an execution-time error. Some procedures are described as accepting the non-datatype "structure"; these procedures will accept an argument of any programmer-defined datatype. Some procedures are described as accepting the non-datatype "any"; these procedures impose no restrictions on their arguments. Some procedures are described with an empty argument list; these procedures are defined to have no arguments.

There are two generalizations not specifically mentioned in the descriptions: (1) a procedure which accepts a Pattern will accept a String or an Integer; (2) a procedure which accepts a String will accept an Integer.

Any predefined procedure may be called with more or fewer arguments than are shown in its definition. Missing arguments are assumed to be the null value; extra arguments are evaluated but otherwise ignored. The evaluation of extra arguments may have important consequences, however; if the evaluation involves the invocation of procedures which produce side effects, for example, it will cause those side-effects to occur before the outer procedure call occurs, and failure during any part of the evaluation of the arguments will result in failure of the rule before the procedure call occurs. The extra arguments are ignored only in the sense that they are not passed to the procedure being called.

I. PROGRAM PROCEDURES

I.A Test Procedures

IDENT(any,any)	Returns: null value, or fails
DIFFER(any,any)	Returns: null value, or fails

IDENT() and DIFFER() are used to compare two arguments of any datatype to see if they are indistinguishable to the Snobol system -- equivalent pattern structures, the same array, equal integers, identical character strings, or whatever. IDENT() succeeds if its arguments are identical; DIFFER() succeeds if its arguments are not identical.

```
IDENT(PRU.PAT,TEST.PAT)    ;    DIFFER(WORD,NULL)
```

LGT(String,String)	Returns: null value, or fails

LGT() -- a mnemonic for Lexicographically Greater Than -- compares two strings to see if they are "alphabetically" ordered, using as an alphabet the computer's character set in its standard collating sequence. (Notice that the arguments must be given in the reverse of the desired order; the test is whether the first argument <u>follows</u> the second argument.)

```
LGT(WORD,'LEMUEL')    ;    LGT(WORD,TEST)
```

EQ(Integer,Integer)	Returns: null value, or fails
EQ(Real,Real)	Returns: null value, or fails
NE(Integer,Integer)	Returns: null value, or fails
NE(Real,Real)	Returns: null value, or fails
GT(Integer,Integer)	Returns: null value, or fails
GT(Real,Real)	Returns: null value, or fails
GE(Integer,Integer)	Returns: null value, or fails
GE(Real,Real)	Returns: null value, or fails
LT(Integer,Integer)	Returns: null value, or fails
LT(Real,Real)	Returns: null value, or fails
LE(Integer,Integer)	Returns: null value, or fails
LE(Real,Real)	Returns: null value, or fails

These arithmetic test procedures are used to compare the first argument to the second argument to see if the relationship symbolized by the procedure name is true. The two arguments must be of the same datatype.

```
EQ(ACNT,BCNT)    ;  LT(LINE,5)
X  =  LE(X,8)   X + 1                :  F(OUT)
```

I.B Result Procedures

ANY(String) Returns: Pattern

ANY() returns a pattern which will match any single character from its argument string.

```
ANY('AEIOU')    ;    ANY(VOWELS)
```

NOTANY(String) Returns: Pattern

NOTANY() returns a pattern which will match any single character not appearing in its argument string.

```
NOTANY('AEIOU')    ;    NOTANY(VOWELS)
```

SPAN(String) Returns: Pattern

SPAN() returns a pattern which will match the longest continuous string of one or more characters appearing in its argument string.

```
SPAN('AEIOU')  ;  SPAN(VOWELS)  ;  SPAN('MISSISSIPPI')
```

BREAK(String) Returns: Pattern

BREAK() returns a pattern which will match the longest continuous string of none or more characters not appearing in its argument string; that is, everything up to but not including any character in its argument.

```
BREAK('AEIOU')  ;  BREAK(VOWELS)  ;  BREAK('MISSISSIPPI')
```

LEN(Integer) Returns: Pattern

LEN() returns a pattern which will match any string of characters of the length given by its argument.

LEN(5) ; LEN('22') ; LEN(SIZE(VOWELS))

TAB(Integer) Returns: Pattern

TAB() returns a pattern which will match all the characters up to the string position specified by its argument. (The convention for string numbering is that string position 0 precedes the first character, string position 1 is after the first character, and string position n is after the n-th character.)

TAB(5) ; TAB('22') ; TAB(COUNT)

RTAB(Integer) Returns: Pattern

RTAB() returns a pattern which will match all the characters up to the string position specified by its argument. Its action is identical to TAB(), matching strings of characters from left to right; the only difference between them is the numbering convention used by the argument. (RTAB()'s numbering convention is that string position 0 is after the last character, string position 1 is before the last character, and string position n is before the n-th character from the end of the string.)

RTAB(5) ; RTAB('22') ; RTAB(0)

POS(Integer) Returns: Pattern

POS() returns a pattern which will match only the string position specified by its argument; it matches no characters at all. (String positions follow the numbering convention of TAB().)

POS(0) ; POS(5) ; POS('22')

RPOS(Integer) Returns: Pattern

RPOS() returns a pattern which will match only the string position specified by its argument; it matches no characters at all. (String positions follow the numbering convention of RTAB().)

```
RPOS(5)    ;    RPOS('22')    ;    RPOS(COUNT)
```

ARBNO(Pattern) Returns: Pattern

ARBNO() returns a pattern which will match zero or more occurrences of the pattern which is its argument.

```
ARBNO(BREAK('□.,;') LEN(1))    ;    ARBNO(ANY('AEIOU'))
```

TRIM(String) Returns: String

TRIM() returns a string which is the same as its argument, but shorn of trailing blanks.

```
TRIM(WORD)    ;    TRIM(INPUT)    ;    TRIM(UNCLE.TOBY)
```

I.C Data Procedures

ARRAY(String) Returns: Array

ARRAY() accepts as its single argument a prototype string specifying the number of dimensions wanted and the upper and lower bounds for the index of each dimension. ARRAY('10,15') specifies a two-dimensional array with indices from one to ten and one to fifteen. ARRAY('0:60,-5:+5') specifies a two-dimensional array with indices from zero to sixty and from minus five to plus five (i.e., a sixty-one by eleven item array). All array items are initialized to the null value. There is no limit on the number of dimensions which may be specified for an array.

Since ARRAY() returns an object of datatype Array as its value, it is used by writing something like

```
LIST = ARRAY('0:60')
```

which has the effect of creating a family of sixty-one variables, which may then be referred to by the item references LIST[0], LIST[1],...,LIST[60].

PARAM(Pattern) Returns: Pattern, String, or Integer

PARAM() accepts as its argument only a pattern returned by one of the ten predefined pattern procedures; it returns the argument (parameter) with which one of those was called to construct the pattern. If the pattern is one constructed by LEN(), POS(), RPOS(), TAB(), or RTAB(), then PARAM() returns an integer; if the pattern was constructed by ANY(), NOTANY(), SPAN(), or BREAK(), then PARAM() returns a string of characters in their standard collating sequence (the sequence defined by ALPHABET()). If the pattern was constructed by ARBNO(), then PARAM() returns the pattern that was its argument, which may of course be of datatype String or Integer in simple cases.

FIRST(Pattern) Returns: Pattern

FIRST() accepts as an argument a pattern constructed by an alternation or concatenation operator. It returns the first element of the pattern. Thus if

```
PAT = X Y | Z
```

has been executed, then

```
FIRST(PAT)
```

returns the pattern which is the value of the expression X Y, a concatenation. On the other hand, if

```
PAT = X (Y | Z)
```

has been executed, then

```
FIRST(PAT)
```

returns the pattern which is the value of X.

REST(Pattern) Returns: Pattern

REST() is the complement to FIRST(); it also accepts alternated or concatenated patterns as arguments, and returns all but the first element. Thus, if

```
PAT = X Y | Z
```

has been executed, then

```
REST(PAT)
```

returns the pattern which is the value of Z. If, however,

```
PAT = X (Y | Z)
```

has been executed, then

```
REST(PAT)
```

returns the pattern which is the value of Y | Z, an alternation.

LEFT(Pattern) Returns: Pattern

LEFT() accepts as an argument a Pattern constructed by an immediate assignment or conditional assignment operator; it returns the pattern which is the left-hand operand of that operator. Thus if

```
PAT = ANY(VOWELS) . V
```

has been executed, then

```
LEFT(PAT)
```

returns the pattern which is the value of the expression ANY(VOWELS).

RIGHT(Pattern) Returns: Name
RIGHT(Name) Returns: String

RIGHT() may have a pattern constructed by an assignment operator, in which case it is the complement to LEFT(). For instance, if

```
PAT = ANY(VOWELS) $ V
```

has been executed, then

```
RIGHT(PAT)
```

returns the value of the expression .V, the Name of the variable V.

RIGHT() may also have as argument a deferred evaluation pattern, in which case it returns the Name of the operand of the deferred evaluation operator. If

```
PAT  =  *V
```

has been executed, then

```
RIGHT(PAT)
```

returns the value of the expression .V, the Name of the variable V.

Finally, RIGHT() may have as its argument the Name (datatype Name) of a natural variable, in which case it returns the String which is the other name of that variable. (RIGHT() will not accept the Name of a created variable, nor the String name of a natural variable.) Thus, the value of RIGHT(.V) is the String V; the statements

```
PAT  =  ANY(VOWELS) $ V
OUTPUT  =  RIGHT(RIGHT(PAT))
```

will print the character V. Since objects of datatype Name cannot be printed, it is the RIGHT() procedure which converts Names of natural variables into a form suitable for assignment to OUTPUT. (To print Names of created variables, see FAMILY() and SELECTOR() below.)

FAMILY(Name) Returns: Array or structure

FAMILY() accepts as argument the Name of a created variable (array item, or field of a programmer-defined data structure). It returns the object which is the family of variables to which the Named variable belongs. If LIST has been assigned an array as value as in

```
LIST  =  ARRAY('0:10')
```

and the rule

```
ELEMENT  =  .LIST[5]
```

has been executed (notice that the value of ELEMENT is of datatype Name), then

```
FAMILY(ELEMENT)
```

returns the Array which is the value of LIST. Similarly, after the statements

```
DATA('NODE(LLINK,RLINK,INFO)')
NEXT  =  NODE(,,15)
ELEMENT  =  .INFO(NEXT)
```

have been executed, then

```
FAMILY(ELEMENT)
```

returns the object of datatype Node which is the value of NEXT.

Since FAMILY() returns the Array or structure rather than the Name of the variable whose value is the Array or structure, the value of FAMILY() is suitable for use as the first argument of ITEM(), or a second argument of APPLY().

SELECTOR(Name) Returns: String

SELECTOR() is the other half of FAMILY(). It also accepts as its argument the Name of a created variable, and returns a String which may be used to select that variable in its family. For Arrays, SELECTOR() returns a string which is a list of indices; for structures, SELECTOR() returns a string naming a field selection procedure. The String returned by SELECTOR() is appropriate for use as the first argument of APPLY(), or a second argument of ITEM(). (Note that this last use takes advantage of the fact that ITEM() will accept such a String of indices; only in the case of one-dimensional Arrays may the value of a call to SELECTOR() be used within square brackets in an item reference.)

II. SYSTEM PROCEDURES

II.A Declarations

DEFINE(String,String) Returns: null value

The first argument of DEFINE() is a string consisting of the name of the procedure being defined, followed by a pair of parentheses containing the names of the formal variables (if any), which in turn are followed (without a comma) by the names of internal variables (if any). The second argument is a string naming the "entry label" for the procedure; if the second argument is null, the entry label is assumed to have the same form as the name of the procedure being defined.

```
DEFINE('PRINT(N,NAME)M,W,E')
DEFINE('RECORDS()','PR.RECORDS')
```

DATA(String) Returns: null value

The DATA() declaration has as its argument a prototype string consisting of the name of the datatype being defined, followed by a parenthesized list of the names of the fields which an object of that datatype is to comprise (if any). The effect of the DATA() declaration is to define (without any DEFINE()'s) a structure creation procedure for the datatype, along with a field selection procedure for each field. Thus, after the declaration

```
DATA('NODE(LLINK,RLINK,INFO)')
```

has been executed, Node's may be created with statements of the form

```
NEXT = NODE() ; CURRENT = NODE(NEXT,,TRIM(INPUT))
```

Fields of the created structure have values initialized according to the values of the corresponding arguments of the procedure call; null arguments produce null fields.

The variables which are fields of structures are referred to by field references, consisting of a reference to a field selection procedure with an argument of the proper datatype to specify the family; for the example above, by statements of the form

```
LEFT  =  LLINK(CURRENT)
NAME  =  INFO(NEXT)
RLINK(CURRENT)  =  NEXT
```

The same field name may be used in definitions of more than one datatype, since its interpretation is governed by the datatype of the argument in any field reference. Notice, however, that the names of structure creation procedures and field selection procedures are drawn from the same set as all other procedure names, so that (for instance) defining a structure

```
DATA('ENTRY(TYPE,SIZE,INFO)')
```

will re-define the predefined procedures TYPE() and SIZE() as field selection procedures for objects of datatype Entry.

II.B Access to System Information

SIZE(String) Returns: Integer

SIZE() returns the integer length (the number of characters) of the string which is its argument.

```
SIZE(VOWELS)   ;   SIZE(TRIM(INPUT))
```

DATATYPE(any) Returns: String

DATATYPE() returns the string of characters which is the name of the datatype of its argument (predefined or programmer-defined). It is used for controlling branching, and can be used with IDENT() to simulate other test procedures. To test whether COUNT is an integer, write IDENT(DATATYPE(COUNT),'INTEGER').

```
DATATYPE(COUNT)   ;   :($('L' DATATYPE(VAL)))
```

TYPE(any) Returns: String

TYPE() returns the same result as DATATYPE() for objects of predefined datatypes, and the string DATA for objects of programmer-defined datatypes. Thus, an exhaustive listing of the strings returned by TYPE() is:

STRING	INTEGER	REAL	PATTERN
ARRAY	NAME	CODE	DATA

```
PROTOTYPE(Array)         Returns: String
PROTOTYPE(structure)     Returns: String
PROTOTYPE(Pattern)       Returns: String
PROTOTYPE(Name)          Returns: String
```

PROTOTYPE() returns as its value a String representing the system definition of the object which is the value of its argument. Its operation is rather different according to the datatype of its argument. In each case, the string returned is intended to be convenient for investigation by Snobol pattern-matching.

When the argument of PROTOTYPE() is an object created by a call to the predefined structure creation procedure ARRAY(), the string returned is the list of upper and lower bounds of indices for the dimensions -- essentially the same as the argument given to the ARRAY() procedure, except that lower bounds are always explicitly present, and each integer is in canonical form (no signs for positive numbers, no leading zeroes). Thus, if the rule

```
LIST  =  ARRAY('00:5,-1:+3,05')
```

has been executed, then

```
PROTOTYPE(LIST)
```

will return the 12-character string 0:5,-1:3,1:5.

When the argument of PROTOTYPE() is an object of a programmer-defined datatype -- one created by a call to a programmer-defined structure creation procedure -- then the string returned is that defining the datatype of the object. This is the same as the string which was the argument of the call to the DATA() procedure which declared the datatype -- not the argument list of the structure creation procedure which created the object (unlike the case for Arrays). Thus, if the two statements

```
DATA('NODE(LLINK,RLINK,INFO)')
CURRENT  =  NODE(LAST,,'SCNNET□15')
```

have been executed, the value of CURRENT is an object of datatype Node, with its LLINK() and INFO() fields initialized as shown and its RLINK() field null. Then the rule

```
PROTOTYPE(CURRENT)
```

would return the 22-character string NODE(LLINK,RLINK,INFO).

For both arrays and data structures, the argument of PROTOTYPE() is an object which is a family of variables, and the result returned is a string which can be used to determine all the valid selectors for members of that family — items or fields, as the case may be. (The difference is that for arrays this information is provided in the argument to the predefined structure creation procedure, for data structures this information is given in the declaration of the datatype.) In the last example, for instance, one could obtain the values of the fields of the object named by CURRENT by obtaining its PROTOTYPE(), then searching with a pattern between the parentheses to find the strings delimited by commas, and using the strings located in this way as the first argument of APPLY() with CURRENT as the second argument.

This idea is extended to objects of datatype Pattern and datatype Name, by observing that although objects of these datatypes are not families of variables, nevertheless they may have an internal structure which a Snobol program may wish to investigate. A Pattern may be constructed of many parts, for instance, and a Name may indicate a family plus a selector. For this reason, the different kinds of Patterns and Names are provided with predefined system prototypes, strings which contain substrings corresponding to the names of the predefined field selection procedures (see section I.C of this appendix). Thus, the structure of Patterns and Names may be investigated in the same way as that of programmer-defined data structures. The twenty-one predefined prototypes for patterns are given in the right-hand column of the following table.

<u>predefined pattern variables</u>

```
P = ARB  ;                        PROTOTYPE(P)-> ARB()
P = REM  ;                        PROTOTYPE(P)-> REM()
P = BAL  ;                        PROTOTYPE(P)-> BAL()
P = FENCE  ;                      PROTOTYPE(P)-> FENCE()
P = FAIL  ;                       PROTOTYPE(P)-> FAIL()
P = ABORT  ;                      PROTOTYPE(P)-> ABORT()
```

predefined pattern procedures

```
P = LEN(6)   ;                 PROTOTYPE(P)-> LEN(PARAM)
P = POS(6)   ;                 PROTOTYPE(P)-> POS(PARAM)
P = RPOS(6)   ;                PROTOTYPE(P)-> RPOS(PARAM)
P = TAB(6)   ;                 PROTOTYPE(P)-> TAB(PARAM)
P = RTAB(6)   ;                PROTOTYPE(P)-> RTAB(PARAM)
P = ANY('AEIOU')   ;           PROTOTYPE(P)-> ANY(PARAM)
P = NOTANY('AEIOU')   ;        PROTOTYPE(P)-> NOTANY(PARAM)
P = SPAN('AEIOU')   ;          PROTOTYPE(P)-> SPAN(PARAM)
P = BREAK('AEIOU')   ;         PROTOTYPE(P)-> BREAK(PARAM)
P = ARBNO(ANY('AEIOU'))   ;    PROTOTYPE(P)-> ARBNO(PARAM)
```

alternation and concatenation

```
P = 'A' | 'B' | 'C'   ;        PROTOTYPE(P)-> ALT(FIRST,REST)

P = 'A' ANY('AEIOU') 'C'   ;   PROTOTYPE(P)-> CAT(FIRST,REST)
```

assignment operators

```
P = SPAN('AEIOU') . VOWELS  ; PROTOTYPE(P)-> PRD(LEFT,RIGHT)
P = BREAK('AEIOU') $ VOWELS ; PROTOTYPE(P)-> DOL(LEFT,RIGHT)
```

deferred evaluation

```
P = *VOWEL  ;                  PROTOTYPE(P)-> STAR(RIGHT)
```

Similarly, a Name may be the name of a natural variable (one that is also named by a String), or one of the two types of created variables — an Array item, or a field of a data structure. There is a predefined prototype for each of these:

```
VAR = .VOWELS  ;      PROTOTYPE(VAR)-> INDIRECT(RIGHT)
VAR = .LIST[I,J]  ;   PROTOTYPE(VAR)-> ITEM(FAMILY,SELECTOR)
VAR = .RLINK(NODE) ; PROTOTYPE(VAR)-> APPLY(SELECTOR,FAMILY)
```

Notice that the Name of a natural variable, returned by the name operator, is a suitable argument for PROTOTYPE(); the String which names the same variable (in the example above, VOWELS) would cause an execution-time error as an argument of PROTOTYPE().

ALPHABET() Returns: String

ALPHABET() returns the 63-character string which is the Snobol character set in standard collating sequence (see Appendix I).

```
ALPHABET()
```

DATE() Returns: String

DATE() returns a nine-character string representing the current date, in the form 02□JUL□72. The abbreviations used for the months are the first three letters of their names.

```
DATE()
```

CLOCK() Returns: String

CLOCK() returns an eight-character string representing the time of day at which the job is being run, in the form 19:03:57. Hours are counted from zero through twenty-three, minutes and seconds from zero through fifty-nine.

```
CLOCK()
```

TIME() Returns: Integer

TIME() returns the elapsed central processor time for the job, expressed as an integer number of milliseconds. By subtracting the value of one call to TIME() from the value of a later call, a programmer is able to determine the amount of central processor time used by a particular part of his program.

```
TIME()
```

STCOUNT() Returns: Integer

STCOUNT() returns the count kept by the Snobol system of the number of statements on which execution is begun. Its initial value is, of course, zero when a program starts executing.

```
STCOUNT()
```

STLIMIT(Integer) Returns: Integer

STLIMIT() is used to set the limit on the number of statements executed (the value of STCOUNT()). Its initial value is 1,000,000; lower limits may be set by the programmer by calling STLIMIT() with a non-null integer argument. An execution-time error results if STLIMIT() is exceeded. If called with a null argument, STLIMIT() returns its current value and remains unchanged.

```
STLIMIT('200')   ;   STLIMIT(5000)   ;   STLIMIT()
```

MAXLNGTH(Integer) Returns: Integer

MAXLNGTH() is used to set the limit on the length of strings which may be formed, in characters. Its initial value is 131,070; lower limits may be set by a programmer by calling MAXLNGTH() with a non-null integer argument. An execution-time error will result if an attempt is made to exceed this maximum length for strings. If called with a null argument, MAXLNGTH() returns its current value and is unchanged.

```
MAXLNGTH('200')   ;   MAXLNGTH(5000)   ;   MAXLNGTH()
```

FNCLEVEL() Returns: Integer

FNCLEVEL() returns an integer value to indicate the level of evaluation of nested or recursive procedure calls. Its use is to provide a trace of the evaluation for debugging of program logic, or to preserve a record of the level of evaluation causing a failure during execution. (At an execution-time error, this information is displayed by the system's error message.)

```
REMARK(TIME()  '--' FNCLEVEL() '□DEEP')
```

NEXTVAR(Name) Returns: Name
NEXTVAR(String) Returns: Name

NEXTVAR() accepts as its argument the Name of a created variable, or either the Name or String naming a natural variable.

For created variables -- array items or fields of data structures -- NEXTVAR() returns the name of the "next" member of the same family. For Arrays, names of items are returned in the order obtained by varying the rightmost index most rapidly. For data structures, names of fields are returned in left to right order of their appearance in the DATA() declaration which defined the datatype. In both cases, the order is cyclical, the name of the "first" member of a family (under this definition) being the value of NEXTVAR() applied to the name of the "last" member. Thus, if the rule

```
        LIST   =   ARRAY('0:2,0:2')
```

has been executed, the value of NEXTVAR(.LIST[0,0]) is the name of the array item referred to as LIST[0,1], and the value of NEXTVAR(.LIST[2,2]) is the name of the array item referred to as LIST[0,0]. Similarly, if the rules

```
        DATA('NODE(LLINK,RLINK,INFO)')
        CURRENT = NODE()
```

have been executed, the value of NEXTVAR(.LLINK(CURRENT)) is the name of the field referred to as RLINK(CURRENT), and the value of NEXTVAR(.INFO(CURRENT)) is the name of the field referred to as LLINK(CURRENT).

If a statement such as

```
        NEXT   =   NEXTVAR(NEXT)
```

is written in a loop, then the names of all the members of the family to which the value of NEXT belongs will be returned in order; but unless the programmer checks to see when he is back to where he started, the loop will be infinite. A suitable loop for going once through the fields of a Node, then would be

```
        SAVE   =   .LLINK(CURRENT)
        NEXT   =   SAVE
LOOP  [statements to process a field]
        NEXT   =   NEXTVAR(NEXT)
        IDENT(NEXT,SAVE)                 :  F(LOOP)
```

NEXTVAR() is convenient for referring in turn to all the variables of an array or a data structure, but its effect can be programmed in Snobol using PROTOTYPE(), ITEM(), and APPLY(). (See an example of this in Chapter 7.)

The more important use of NEXTVAR() arises from the fact that it also treats the set of all natural variables as a "family," and thus when given a String or a Name which names a natural variable, NEXTVAR() returns the name of another natural variable. Two important differences of NEXTVAR() in this use should be noted. First, since there is no defined order for the natural variables, their names are returned in an order which is convenient for NEXTVAR(). Second, NEXTVAR() cannot cycle through the names of all the natural variables, since there are an infinite number of them. Hence, it returns the names of a subset of the family of natural variables which is certain to include at least the names of all variables with non-null values, and may also include the names of some variables with null values. What is important is that by the time a full cycle has been completed and the starting place reached again, the name of every variable with a non-null value will have come up. (When used with families of created variables, by contrast, NEXTVAR() is guaranteed to cycle through the names of every variable in the family in turn, regardless of their values.) Observe that the names returned by NEXTVAR() are subject to the usual interpretation of names. In particular, if NEXTVAR() is called repeatedly in a loop within the body of a programmer-defined procedure, and some process is carried out on the variables referenced by the names returned, then the names of variables internal to procedure calls will refer to those internal variables. The customary interpretation of what variable a name refers to at any point in the execution of a program is not affected by NEXTVAR().

II.C Requests for System Actions

ITEM(Array,String,...,String) Returns: variable, or fails

ITEM() provides a convenient way to write item references for arrays chosen at execution-time, for arrays which are the values of array items, or which involve variable numbers of dimensions. The first argument of ITEM() is an array, and the following arguments are either integers or else lists of integers separated by commas. ITEM() constructs an item reference using the array which is its first argument for the family and the proper number of indices gathered from the remaining arguments to form the selector, ignoring extra indices and supplying null (zero) for missing ones. ITEM() NRETURNs the array item so referenced, or FRETURNs if any index of the selector exceeds the bounds specified by the prototype for the array. If TIC3 has been assigned the value

```
TIC3 = ARRAY('1:5,1:5,1:3')
```

then equivalent ways of referring to its central item are

```
TIC3[3,3,2]
ITEM(TIC3,3,2,2)
ITEM(TIC3,'3,3,2')
ITEM(TIC3,3,'3,2')
```

APPLY(String,any,...,any) Returns: any or variable, or fails

APPLY() provides the only way to write procedure references for procedures chosen at execution-time. The first argument of APPLY() must be a string which names a procedure; the Snobol system calls that procedure, using as its arguments the remaining arguments of APPLY() and observing the usual conventions for extra or missing arguments. APPLY() returns the value returned by the procedure it calls, using the same return (RETURN, NRETURN, or FRETURN).

If APPLY() is used to call a field selection procedure, then its use is analogous to the use of ITEM() for item references; the Snobol system forms a field reference using the first argument as the selector and the second argument for the family, and NRETURNs the field so selected.

```
FLD = 'RLINK'
APPLY( FLD,CURRENT) = TRIM(INPUT)
RLINK(CURRENT) = APPLY('TRIM',INPUT)
```

IF() Returns: null value

IF() always succeeds. Since it is defined to have no arguments, any arguments in a reference to IF() are evaluated but otherwise ignored. Thus if any part of that evaluation fails, that failure causes failure of the rule. If a reference to a procedure returning a non-null value is written as an argument of an IF() procedure, the combination will work like a test procedure. The same principle applies to other expressions returning values which can similarly be converted into test procedures.

```
N = IF(ARR1[N+1]) N + 1 : F(OUT)
```

ANCHOR(any) Returns: null value

ANCHOR() works like a switch, distinguishing between null and non-null arguments. Calling ANCHOR() with a non-null argument turns on the anchored mode of pattern-matching; calling it again with a null argument restores the usual, unanchored mode.

```
ANCHOR('ON')   ;   ANCHOR(OFF)   ;   ANCHOR()
```

CONVERT(Integer) Returns: Real
CONVERT(String) Returns: Real
CONVERT(Real) Returns: String

CONVERT() is useful for creating and printing real numbers. If its argument is of datatype Integer, the value returned is the corresponding real number. The only permissible String-valued argument is a string of digits, possibly including an initial sign and possibly including a decimal point; the returned value is the corresponding real number. If the argument is of datatype Real, the value returned by CONVERT() is the numeral string representing the real number to twelve digits. CONVERT() is defined for integers and real numbers from about 10^{-300} to about 10^{300}.

```
CONVERT(45)  ;  CONVERT('-57.69')   ;   CONVERT('.75')
CONVERT(REALNUMB)   ;   CONVERT(TRIM(INPUT))
```

CODE(String) Returns: Code

CODE() accepts as its argument a string which is a Snobol program text; that is, a sequence of syntactically-correct Snobol statements (see the definition of the construct <program text> in the syntax, Appendix J), and returns as its value the corresponding compiled Code; its use, then, is to permit a program to extend itself while it is executing. All characters in the Snobol character set, including space, have their customary significance in the argument to CODE(). Statement separators are semicolons, but no final semicolon is required in the string.

```
        NULP =  CODE('LOOP  BLWORD  "A"  =       ;'
+ '   N  =  LT(N,X)   N + 1        :  S(LOOP)   F($("L" X))')
```

IID. Input/Output Procedures

```
INPUT(String,String,String) Returns: null value
INPUT(Name,String,String)   Returns: null value
```

INPUT() is used to associate a variable in a Snobol program with an input file. The first argument is the name of a variable to be used in the program; the second argument specifies a SCOPE fileset; the third argument specifies the number of characters to be read from each record on the file. (Excess characters are lost; missing characters are filled out with spaces.) If the variable is already associated with a file, it loses its previous association. It is through INPUT() -- and OUTPUT() -- procedures that the Snobol program establishes contact with the files set up for it by SCOPE.

```
INPUT('READ','INPUT','50')
INPUT('LNGREADER','DISKSRT',600)
INPUT(.LIST[12],'TAPE1',TRIM(INPUT))
INPUT(.LLINK(NEXT),'INFILE',80)
```

```
OUTPUT(String,String,String) Returns: null value
OUTPUT(Name,String,String)   Returns: null value
```

OUTPUT() is used analogously to INPUT(), to associate variables in Snobol programs with SCOPE filesets which are to be used for output. The first argument is the name of a variable to be used in the Snobol program; the second argument specifies a SCOPE fileset; the third argument is the carriage control character which will be concatenated at the head of every record written. (If omitted, none will be concatenated.) If the variable is already associated with a file, it loses its previous association.

```
OUTPUT('WRITE','OUTPUT','-')
OUTPUT('PAGE','DISKFIL',1)
OUTPUT(.LIST[13],'TAPE1',□')
OUTPUT('PUNCH','PUNCH')
OUTPUT(.RLINK(NEXT),'OUTFILE')
```

```
DETACH(String)           Returns: null value
DETACH(Name)             Returns: null value
```

DETACH() is used to break the association between the variable named by its argument and any fileset. There is no need to DETACH() an associated variable before giving it a new association. (A variable may be associated with only one fileset at a time, but a fileset may have many variables associated with it simultaneously.)

```
DETACH('OUTPUT')
DETACH('WRITE')
DETACH(.LIST[12])
DETACH(.RLINK(NEXT))
```

```
ENDGROUP(String,Integer) Returns: null value
```

ENDGROUP() writes a SCOPE end-of-group mark on the SCOPE fileset which is specified by its first argument. The "level" associated with the mark is specified by the second argument, which must be an integer between 0 and 15 inclusive. Such a mark of any level will cause failure on input if later read by a Snobol program.

```
ENDGROUP('TAPE20',9)    ;    ENDGROUP('DISKFIL')
```

```
REWIND(String)           Returns: null value
```

REWIND() performs a standard SCOPE rewind on the SCOPE fileset specified by its argument. The fileset is positioned at its beginning; if the last operation on this file was a write, an end-of-group mark of level zero is written before the file is rewound.

```
REWIND('TAPE20')    ;    REWIND('DISKFIL')
```

```
REMARK(String)           Returns: null value
```

REMARK() is used to write the string which is its argument onto the special file which is the job log. Obvious uses are to preserve messages about the course of execution associated with timing information, and to decorate the dayfiles.

```
REMARK('ENTERING FREEZE TO TAPE20.')
REMARK('MOTHER IS DEAD.')
```

FREEZE(String) Returns: String

FREEZE() is a procedure which permits a programmer to suspend execution of a compiled Snobol program, and then to re-load it and re-commence execution. The argument to FREEZE() is a string which is the name of a SCOPE fileset. When FREEZE() is encountered during execution, the Snobol system writes out a copy of the entire field length of the job onto the fileset specified by the argument, and execution is terminated. SCOPE then reads and carries out the next control card. When SCOPE finally hits a control card asking that the Snobol program be reloaded, it does so and execution continues from the point where it was frozen.

On a call in a program such as FREEZE('TAPE20'), the program is "frozen" onto SCOPE fileset TAPE20. Execution begins again when a SCOPE control card is encountered of the form LGO,TAPE20. There is no requirement, naturally, that a frozen program be loaded and executed in the same job in which it was written out; it can perfectly well be saved on a COMMON file, or on tape, or even punched out on cards.

It is a peculiarity of FREEZE() that it returns for its value the string which is its argument. This could be used to preserve a record of which of several FREEZE()'s had been executed, but FREEZE() is customarily written where its returned value is not preserved.

```
        FREEZE('DISKFIL')
```

EOI(String) Returns: null value, or fails

EOI() tests whether the SCOPE fileset specified by its argument is positioned at the end-of-information on the file. If so, the procedure succeeds and returns the null value. If there is more information on the file, the procedure fails.

```
      EOI('TAPE20')       :  S(OUT)
```

EORLEVEL(String) Returns: Integer, or fails

EORLEVEL() tests to see whether the SCOPE fileset named by its argument is positioned at an end-of-group mark; if so, the level associated with the mark is returned as the value of the procedure call. (Such a mark is written by the ENDGROUP() procedure; the value returned by EORLEVEL() is the second parameter of the ENDGROUP() which wrote the mark, 0 to 15 inclusive.) If the fileset is positioned at end-of-information -- if the EOI() procedure would succeed -- the value returned by EORLEVEL() is -1.

As a practical matter, a fileset will only be positioned at an end-of-group mark if the last reference to a variable associated with that fileset failed; customarily, then, a call to EORLEVEL() would only be made after a failure on input had occurred, to check the level of the end-of-group mark which caused the failure. If a call to EORLEVEL() is executed at any other time -- at any time when the fileset is not at an end-of-group mark -- the call to EORLEVEL() will itself fail.

```
EQ(EORLEVEL('TAPE20'),9)          :  S(NINE)
LVL  =  EORLEVEL('DISKFIL')
```

Appendix B. SUMMARY OF PREDEFINED PATTERN VARIABLES

There are precisely six variables initialized to a value other than the null value when execution of a Snobol program begins: the six natural variables named ARB, REM, BAL, FAIL, ABORT and FENCE. Each of these has a pattern as its initial value, but except for this initialization receives no special treatment. Each may be assigned any value by a program, upon which its initial value is lost. This makes no great difference for ARB, REM, BAL, or FAIL, but the value of ABORT is a pattern which cannot be constructed in any other way by a Snobol program, and FENCE can be constructed only with the use of ABORT.

<u>ARB and REM.</u> The patterns which are the initial values of ARB and REM are equivalent in effect to two commonly used patterns which may be constructed by pattern procedures. ARB is equivalent to the value of the expression ARBNO(LEN(1)); REM is equivalent to the value of the expression RTAB(0). The Snobol system can and does distinguish between ARB and ARBNO(LEN(1)), or between REM and RTAB(0); an IDENT() comparison of such a pair will fail, and PROTOTYPE() will return different prototype strings for them. But the performance of either member of a pair in a pattern-matching statement is exactly the same.

<u>BAL.</u> BAL has as its initial value a pattern which matches any non-null string of characters which is "balanced" with respect to parentheses — that is, which has the same number of left and right parentheses, including none, where each left parenthesis occurs before its matching right parenthesis. A pattern equivalent to the initial value of BAL can be constructed in Snobol, thus providing a precise definition of its action:

```
BALEXP = NOTANY('()') | '(' ARBNO(*BALEXP) ')'
BAL  =  BALEXP  ARBNO(BALEXP)
```

Again, the system distinguishes between the predefined BAL and the pattern constructed by the rules above, but the two would perform in the same way in a pattern match.

<u>FAIL.</u> FAIL has as its initial value a pattern which matches <u>no</u> strings (not even the null value), and which thus always fails. This makes it the "empty" pattern alternative — one which may be present in any pattern without altering the set of strings matched. The expressions FAIL | LPAT and LPAT will match the same set of strings, no matter what pattern is the value of LPAT. A pattern which would have the

same effect could be constructed by the rule

```
        FAIL  =  ANY(NULL)
```

One use for the empty pattern alternative is to construct an alternated pattern from data. For instance, with the statements

```
         IN.PAT  =  FAIL
PATLOOP  IN.PAT  =  IN.PAT | TRIM(INPUT)    :  S(PATLOOP)
```

Here the loop statement extends the alternatives of IN.PAT by one more each time it is successfully executed. If the data read were the first three letters of the Greek alphabet spelled out on cards, followed by failure of INPUT, then the resulting pattern would be equivalent to

```
         IN.PAT =  FAIL | 'ALPHA' | 'BETA' | 'GAMMA'
```

which matches the same set of strings as does

```
         IN.PAT =  'ALPHA' | 'BETA' | 'GAMMA'
```

Note that if IN.PAT had not been first assigned the value FAIL, the resulting pattern would have been equivalent to

```
         IN.PAT =  NULL | 'ALPHA' | 'BETA' | 'GAMMA'
```

which is rather different -- since it will match the null value (as its first alternative, in fact), it will always succeed.

<u>ABORT.</u> ABORT has as its initial value a pattern which causes immediate failure of an entire pattern match when it is encountered. The usefulness of ABORT is that it permits a pattern match to fail if something <u>is</u> found. For instance,

```
         SH.PAT  =  LEN(10) ABORT | ':'
```

is a pattern which will fail by ABORT if it is set to search a string of ten or more characters; shorter strings it will search for a colon. It will succeed, then, only on a string of nine or fewer characters containing a colon. More generally, patterns which have characteristics <u>p</u> but not <u>q</u> can often be written in the form <u>q</u> ABORT | <u>p</u> .

<u>FENCE.</u> The initial value of FENCE is a pattern which has the following interesting property: when encountered in a pattern match it matches the null value, and then if the remainder of the pattern cannot be succesfully matched from

that point, the match will fail. A pattern which would have the same effect could be constructed by the rule

```
FENCE = NULL | ABORT
```

When FENCE is used as the first element of a pattern, its effect is like writing POS(0); it "anchors" the pattern so that it must match beginning with the first character. When FENCE is used after other pattern elements, then its effect is that of a conditional "anchor" applying only to the remainder of the pattern, and only if the elements to the left of FENCE within its alternative have been successfully matched.

Appendix C. SUMMARY OF OPERATORS

Operator	Operation	Precedence
unary *	deferred evaluation	7 (highest)
unary .	name	7
unary $	indirect reference	7
binary .	conditional assignment	6
binary $	immediate assignment	6
binary *	multiplication	5
binary /	division	5
unary +	plus	4
unary -	minus	4
binary +	addition	3
binary -	subtraction	3
binary ◻	concatenation	2
binary |	alternation	1 (lowest)

Appendix D. SUMMARY OF PROCEDURE EXECUTION

When a call is made to a programmer-defined procedure: (1) the arguments are evaluated; (2) the variable name which is the same as the procedure name is made to refer to an internal "result variable"; (3) the formal variable names are made to refer to internal "formal variables"; (4) any additional names in the first argument of the DEFINE() procedure are made to refer to additional internal variables; (5) the formal variables are assigned the values of their corresponding arguments; (6) the result variable and all additional internal variables are assigned the null value; (7) control passes to the statement of the procedure body whose label is specified by the second argument of the DEFINE() procedure (this may be expressed by default); (8) execution of the statements of the procedure body continues until a return transfer is executed.

When return is made from a procedure using RETURN: (1) the last value assigned to the result variable is returned as the value of the procedure call; (2) the variables previously referred to by the formal variable names, the result variable name, and any additional internal variable names, are restored; (3) execution of the calling statement continues from the point of the procedure call.

When return is made from a procedure using NRETURN: the variable named by the last value assigned to the result variable (which must be a string or a Name) is returned as the value of the procedure call; the remaining actions are the same as for RETURN.

When return is made from a procedure using FRETURN: (1) the variables previously referred to by the formal variable names, the result variable name, and any additional internal variable names are restored; (2) the call fails, the rule from which the call was made fails, and control is returned to the go-to of the calling statement where the failure transfer will be taken.

Appendix H. PROGRAM TEXT REPRESENTATION

Each statement of a Snobol program is usually punched on a separate 80 column card. Only the first 72 columns, however, may be used for the statement; the remaining columns may be used for purposes of identification. (For example, sequence numbers may be punched there which would allow you to put the deck back in order, either by hand or with a mechanical sorter, if the cards should be disarranged.) All columns of the card appear in the printed listing of the program when it is executed, but 10 spaces are provided between columns 72 and 73 to separate any identification from the statement.

Statement Format. If the label of a statement is present it must be punched starting in column 1. If the label is absent and the rule is present, then column 1 must be left empty and the rule may be punched beginning in column 2 or beyond. If the statement consists only of a go-to, the colon introducing it may be punched in column 1.

Wherever a single blank occurs in a statement, any number of blanks would serve as well; wherever many blanks occur, a single blank would serve as well. Since all parts of a statement may be absent, a totally blank card is treated as a null statement.

The semicolon may be used as a delimiter between statements, making it possible to punch more than one statement per card. The semicolon signals the end of a statement, so the column directly after the semicolon is treated as "column 1" of the following statement. For example, four assignment statements may be punched on a single card as follows:

```
ONE = 1;   TWO = 2;   THREE = 3;LAST  FOUR = 4
```

Note that the final statement of the sequence has a label, while the others do not. A semicolon is assumed at the end of a card which is not followed by a continuation card.

Continuation Cards. More commonly, a method is needed for dealing with statements which are too long rather than too short. Statements which are too long to fit on a single card may be continued onto as many cards as necessary. This is done by means of continuation cards, each of which has either a plus sign or a period punched in column 1, indicating that its information is a continuation of whatever appeared on the foregoing card. Statements may be broken anywhere; a blank is never assumed at the break.

<u>Comment Cards.</u> Comments may be introduced into the program with the use of comment cards, which are distinguished by having an asterisk in column 1, and any other information in the remaining columns. Comment cards may appear anywhere within the program deck except directly before a continuation card. Comments themselves may <u>not</u> be continued by placing a plus sign or a period in column 1.

<u>Listing Control Cards.</u> A card with a minus sign in column 1 is a listing control card, used to specify the format of the listing which is produced by the compiler. The word appearing after the minus sign specifies what is to be done to the listing, as follows:

-SPACE Leave a blank line in the listing.

-EJECT Print the next statement of the compiler listing at the top of a new page.

-UNLIST Stop printing the statements of the program text until a listing control card specifying LIST is encountered.

-LIST Resume printing the program text.

Listing control cards, like comment cards, may appear anywhere within the program deck except directly before a continuation card.

<u>Extended Syntax of Snobol Statements.</u> In addition to the forms used for them in example program texts, certain language elements have alternative representations.

Array Prototypes. Instead of colons in the argument of the ARRAY() procedure, slashes may be used. The rules

```
LIST = ARRAY('0:2,0:3')
```

and

```
LIST = ARRAY('0/2,0/3')
```

would assign identically-dimensioned arrays as the value of LIST. The PROTOTYPE() procedure returns colons in its canonical version of the prototype string, regardless of which character was used in the argument of ARRAY().

Item References. Instead of left and right brackets around the selector of an item reference, a combination of parentheses and adjacent slashes may be used. For example, LIST[2,3] and LIST(/2,3/) are alternative ways of writing the same item reference.

Go-to Parts. Rather than a colon to introduce a go-to part, a slash may be used; but a slash used for this purpose must not be followed by a blank. Thus,

```
        VOWELS  =  TRIM(INPUT)          :  F(ERROR)
```
and
```
        VOWELS  =  TRIM(INPUT)          /F(ERROR)
```

are equivalent statements.

Instead of left and right brackets in direct go-to's (used only in connection with objects of datatype Code), the parentheses and adjacent slashes notation may be used, in the same way as for item references. Thus, the two statements

```
        RESULT  =  CODE(TRIM(INPUT))  :  [RESULT]
```
and
```
        RESULT  =  CODE(TRIM(INPUT))  :  (/RESULT/)
```

are equivalent, as is

```
        RESULT  =  CODE(TRIM(INPUT))    /(/RESULT/)
```

Pattern Alternations. The alternation operator may be written as two adjacent slashes, bounded by blanks, instead of the usual single character. Thus, X | Y and X // Y may be written with the same effect.

String Literals. Within string literals, all characters other than the quotation mark (single or double) being used as the delimiter of that literal may be used freely. The delimiter character may occur within the string only in pairs, and each such pair will be taken to represent a single instance of the character. For example, the rules containing a single string literal each

```
        AWW  =  """ALL'S□WELL"""
```
and
```
        AWW  =  '"ALL''S□WELL"'
```

are equivalent to the rule containing a concatenation of three string literals

```
        AWW  =  '"ALL'   "'"   'S□WELL"'
```

Any one of them would assign to AWW the 12-character string "ALL'S WELL".

Appendix I. CHARACTER SET REPRESENTATIONS

The Snobol character set consists of sixty-three characters: the capital letters A-Z, followed by the digits 0-9, followed by the remaining characters in the order

+ - * / ( ) $ = ¤ , . ≡ [ ] : ' → | ∧ " ↓ < > ≤ ≥ ¬ ;

This ordering of the sixty-three characters is called their standard collating sequence. Fifty-four of these play a part in the syntax of the language (see Appendix J), and have equivalents in the reference symbol set used to construct program texts; the remaining nine characters may occur only in string literals or in data read from input files.

Program texts in examples are shown in symbols from the reference set. For input each of these must be represented by a punched card code produced on a keypunch (either model 026 or model 029); for output each will be represented by a character on a line printer. Each symbol of the reference set has a single card code, and a single printer representation. Each card code and printer representation corresponds to a single reference symbol, except for one special case: the blank used to separate language elements and the space character (¤) used in literal data have the same card code and printer representation, although they are differentiated in the reference symbol set for clarity.

The reference symbol set consists of the twenty-six capital letters, the ten digits, and nineteen special characters. Codes for the letters and digits are produced by the keys marked with them on both an 026 or an 029 keypunch, and all have the expected representation on a line printer.

The special characters in the reference symbol set are shown in the accompanying chart. On an 026 keypunch, codes for the reference symbols are produced by keys marked with the same symbols where they exist, but six symbols (:;"|[ ]) have no keys and so they must be multiple-punched. (In Snobol expressions—not, obviously, in literal data—these six symbols may be avoided by using the extended syntax described in Appendix H.) On an 029 keypunch, codes for all but one of the reference symbols (|) are produced by some key, but most of the keys are marked with different symbols. On a line printer, all but three of the reference symbols ('"|) look like their counterparts in the reference set. The final nine characters in the chart are those without equivalent reference symbols.

Snobol symbol	026 key	card code	line printer character	Snobol usage	029 key
=	=	8-3	= (equal)	assignment	#
.	.	12-8-3	. (period)	condit. assign., name, real lit.	.
,	,	0-8-3	, (comma)	list separator	,
:	none	8-2	: (colon)	go-to's, array prototypes	:
;	none	12-8-7	; (semicolon)	statement terminator	|
'	'	8-4	≠ (not equal)	string literal delimiter	@
"	none	11-8-5	↑ (up arrow)	string literal delimiter	)
$	$	11-8-3	$ (dollar)	indirect ref., immed. assign.	$
|	none	11-0	v (logical or)	alternation	none
(	(	0-8-4	((left paren)	arg. lists, expr. grouping	%
)	)	12-8-4	) (right paren)	arg. lists, expr. grouping	<
[	none	8-7	[(left bracket)	item ref., direct go-to's	"
]	none	0-8-2	] (right bracket)	item ref., direct go-to's	0-8-2
-	-	11	- (minus)	negative, subtraction	-
+	+	12	+ (plus)	positive, addition	&

Snobol symbol	026 key	card code	line printer character	Snobol usage	029 key
*	*	11-8-4	* (asterisk)	deferred eval., multiplication	*
/	/	0-1	/ (slash)	division	/
blank	space bar	blank	(space)	concatenation, separator	space bar
□	space bar	blank	(space)	data only	space bar
	none	0-8-6	≡ (identity)	data only	>
	none	0-8-5	→ (right arrow)	data only	_
	none	0-8-7	∧ (logical and)	data only	?
	none	11-8-6	↓ (down arrow)	data only	;
	none	12-0	< (less than)	data only	none
	none	11-8-7	> (greater than)	data only	¬
	none	8-5	≤ (less or equal)	data only	'
	none	12-8-5	≥ (greater or equal)	data only	(
	none	12-8-6	¬ (logical not)	data only	+

Appendix J. SYNTAX OF PROGRAM TEXTS

```
 1.  <string literal> ::=
             ' <string format 1> ' |
             " <string format 2> "

 2.  <digit string> ::=
             <digit> |
             <digit string> <digit>

 3.  <integer literal> ::=
             <digit string>

 4.  <real literal> ::=
             <digit string> . |
             . <digit string> |
             <digit string> . <digit string>

 5.  <literal> ::=
             <string literal> |
             <integer literal> |
             <real literal>

 6.  <identifier> ::=
             <letter> |
             <identifier> <letter> |
             <identifier> <digit> |
             <identifier> .

 7.  <simple variable> ::=
             <identifier>

 8.  <subscript list> ::=
             <expression> |
             <subscript list> <,> <expression>

 9.  <array item reference> ::=
             <simple variable> <[> <subscript list> <]>

10.  <procedure identifier> ::=
             <identifier>

11.  <argument list> ::=
             <optional expresion> |
             <argument list> <,> <optional expression>
```

```
12.  <procedure reference> ::=
             <procedure identifier> <(> <argument list> <)>

13.  <variable> ::=
             <simple variable> |
             $ <primary> |
             <array item reference> |
             <procedure reference>

14.  <primary> ::=
             <literal> |
             <variable> |
             . <variable> |
             <(> <expression> <)>

15.  <factor> ::=
             <primary> |
             <factor> <blank> ** <blank> <primary>

16.  <multiplying operator> ::=
             <blank> * <blank> |
             <blank> / <blank>

17.  <term> ::=
             <factor> |
             <term> <multiplying operator> <factor>

18.  <adding operator> ::=
             <blank> + <blank> |
             <blank> - <blank>

19.  <sum> ::=
             <term> |
             + <term> |
             - <term> |
             <sum> <adding operator> <term>

20.  <concatenation> ::=
             <sum> |
             <concatenation> <blank> <sum>

21.  <expression> ::=
             <concatenation>

22.  <deferred pattern> ::=
             * <variable>
```

```
23.  <pattern assignment operator> ::=
             <blank> $ <blank> |
             <blank> . <blank>

24.  <pattern assignment> ::=
             <pattern primary> <pattern assignment operator>
                 <variable>

25.  <pattern primary> ::=
             <literal> |
             <variable> |
             . <variable> |
             <deferred pattern> |
             <pattern assignment> |
             <(> <pattern expression> <)>

26.  <pattern factor> ::=
             <pattern primary> |
             <pattern factor> <blank> ** <blank> <pattern primary>

27.  <pattern term> ::=
             <pattern factor> |
             <pattern term> <multiplying operator> <pattern factor>

28.  <pattern sum> ::=
             <pattern term> |
             + <pattern term> |
             - <pattern term> |
             <pattern sum> <adding operator> <pattern term>

29.  <pattern concatenation> ::=
             <pattern sum> |
             <pattern concatenation> <blank> <pattern sum>

30.  <pattern alternation> ::=
             <pattern concatenation> |
             <pattern alternation> <blank> <|> <blank>
                 <pattern concatenation>

31.  <pattern expression> ::=
             <pattern alternation>

32.  <optional expression> ::=
             <null> |
             <pattern expression>

33.  <label> ::=
             <identifier>
```

```
34.  <label part> ::=
             <null> |
             <label>

35.  <right side> ::=
             <=> <optional expression>

36.  <rule part> ::=
             <null> |
             <blank> <primary> |
             <blank> <primary> <blank> <pattern expression> |
             <blank> <variable> <right side> |
             <blank> <variable> <blank> <pattern expression>
                 <right side>

37.  <loc> ::= <location expression> ::=
             <(> <label> <)> |
             <(> $ <primary> <)> |
             <[> <expression> <]>

38.  <go-to part> ::=
             <null> |
             <:> <loc> |
             <:> S <loc> |
             <:> F <loc> |
             <:> S <loc> <optional blank> F <loc> |
             <:> F <loc> <optional blank> S <loc>

39.  <statement> ::=
             <label part> <rule part> <go-to part>

40.  <program text> ::=
             <statement> |
             <program text> <;> <statement>

41.  <letter> ::=
             A | B | C | D | E | F | G | H | I | J | K | L | M |
             N | O | P | Q | R | S | T | U | V | W | X | Y | Z

42.  <digit> ::=
             0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

43.  <blank> ::=
             □ | <blank> □

44.  <optional blank> ::=
             <null> |
             <blank>
```

```
45.  <string format 1> ::=
             <null> |
             <string format 1> <class 1 character>

46.  <class 1 character> ::=
             <any character except '> | ''

47.  <string format 2> ::=
             <null> |
             <string format 2> <class 2 character>

48.  <class 2 character> ::=
             <any character except "> | ""

49.  <(> ::= ( <optional blank>

50.  <)> ::= <optional blank> )

51.  <[> ::= [ <optional blank> |
             (/ <optional blank>

52.  <]> ::= <optional blank> ] |
             <optional blank> /)

53.  <|> ::= <the character |> | //

54.  <:> ::= <optional blank> : <optional blank> |
             <optional blank> /

55.  <,> ::= <optional blank> , <optional blank>

56.  <=> ::= <optional blank> = <optional blank>

57.  <;> ::= <optional blank> ;

58.  <null> ::=
```

Appendix K. SUMMARY OF COMPILE-TIME ERROR MESSAGES

Each statement which is syntactically incorrect is marked in the program listing by an up arrow which is printed beneath its statement number along with the message ERROR. It is planned that in the future a specific message for each particular type of syntactic error will be provided.

Appendix L. SUMMARY OF EXECUTION-TIME ERROR MESSAGES

When an error is detected during the execution of a Snobol program, the Snobol interpreter writes a message on the output file and then ceases execution. The message consists of three parts: (1) the identifying number of the statement being executed when the error was detected (each statement of the program text is given a number by the compiler, and these numbers appear at the left of the statements in the compiler listing of the program text); (2) the level of procedure execution at the time the error was detected (the same information which would be returned by the predefined procedure FNCLEVEL()); (3) one of the error messages from the list below, specifying which of the fifty-two possible errors was detected.

Some of the messages in the following list are self-explanatory. Notes have been added to many messages amplifying them, or explaining terminology which differs from that used in this description of Snobol, or recommending page numbers and sections where further information relevant to the interpretation of the message can be found.

THE LEFT OPERAND FOR A PATTERN MATCH MUST BE A STRING.

THE RIGHT OPERAND FOR A PATTERN MATCH MUST BE A PATTERN.

PATTERN MATCH WITH REPLACEMENT REQUIRES STRING-VALUED RIGHT HAND SIDE.

TRANSFER TO AN UNDEFINED LABEL. A go-to specifies a transfer to a label which is not present in the program text, and which is not RETURN, FRETURN, NRETURN, or END.

A FAILURE OCCURRED IN THE EVALUATION OF THE GO-TO PART. Conditions which would cause failure in the rule part of a statement cause an error in the go-to part (see page 68).

TYPE ERROR IN GO-TO PART. Either the operand of an indirect referencing operator in the go-to is not a string or a Name (see page 67), or else the value of the expression in a direct go-to is not an object of datatype Code.

FORBIDDEN OPERAND TYPE FOR ALTERNATION. Operands of the alternation operator must be of datatype String, Integer, or Pattern (see page 35).

THE DATA TYPE USED MAY ONLY BE CONCATENATED WITH THE NULL STRING. Strings, Integers, and Patterns may be concatenated freely. An object of any other datatype may be concatenated only with the null value.

THE VALUE OF A VARIABLE IN A DEFERRED-EVALUATION PATTERN (UNARY *) MUST BE A PATTERN OR STRING. See the description of the deferred evaluation operator, page 50.

LEFT OPERAND FOR BINARY $ AND . MUST BE A PATTERN. See the descriptions of the immediate and conditional assignment operators, pages 38 and 40.

INDIRECT REFERENCE TO THE NULL STRING. The operand of the indirect referencing operator may not be the null value (see page 57).

OPERAND FOR INDIRECTION MUST BE NAME OR STRING. The operand of the indirect referencing operator must be a string or a Name (see page 57).

NON-INTEGER STRING USED IN NUMERIC CONTEXT. Only strings of datatype Integer — those consisting of an optional sign followed by an optional string of digits — may be used where Integers are expected.

TYPE ERROR IN NUMERIC CONTEXT. An object of either datatype Integer or Real was expected, but an object of some other datatype occurred.

DIVISION BY ZERO WAS ATTEMPTED.

STRING ARITHMETIC NOT YET IMPLEMENTED. Integers may have values of magnitudes as large as 10^{130000}, but the arithmetic operations are defined only for integers of magnitudes less than 10^{10}. It is intended that the arithmetic operations should be extended to integers as large as can be represented, by performing "string arithmetic" on the digit strings of which they are composed.

REAL ARITHMETIC OVERFLOW. A real number larger than can be represented has been produced (about 10^{300}).

MIXED MODES (INTEGER, REAL) FOR ARITHMETIC OPERATION. The operands of arithmetic operators (and the arguments of predefined arithmetic test procedures) must be of the same datatype. If operands of different datatypes are to be operated upon, one must first be converted (see the description of CONVERT() in Appendix A, section II.C).

WRONG PARAMETER TYPE FOR STANDARD PROCEDURE. An argument of a predefined procedure is of an incorrect datatype. Permissible datatypes of arguments for all predefined procedures are given in Appendix A.

ARGUMENT FOR LEN, POS, RPOS, TAB, OR RTAB MUST BE IN THE INTERVAL [0,2**17-1]. The integer arguments to these five predefined pattern procedures must be non-negative, and must be less than 131,072.

SYNTAX ERROR IN STRING TO BE COMPILED. An argument string for the CODE() procedure is incorrect; see the description of CODE() in Appendix A, section II.C, and the Syntax of Program Texts in Appendix J.

INCORRECT SYNTAX FOR STRING TO BE CONVERTED TO REAL. See the description of CONVERT() in Appendix A, section II.C.

IMPROPER ARGUMENT FOR PSEUDO-FIELD FUNCTION (FIRST, REST, LEFT, RIGHT, PARAM, FAMILY, OR SELECTOR). The arguments of the predefined field selection procedures PARAM(), FIRST(), REST(), LEFT(), RIGHT(), FAMILY(), and SELECTOR() are quite specialized; see the descriptions of these procedures in Appendix A, section I.C.

CALL OF AN UNDEFINED PROCEDURE. The DEFINE() declaration for a programmer-defined procedure must be executed before it can be invoked (see page 72).

SYNTAX ERROR IN PROCEDURE PROTOTYPE. There is an error in the form of the string which is the first argument of the DEFINE() procedure (see page 72).

RETURN FROM LEVEL ZERO. A transfer to RETURN, FRETURN, or NRETURN has been executed in a main program (see page 87).

AN -NRETURN- WAS EXPECTED FROM THE PROCEDURE CALLED. A procedure call occurs where a variable is required, but the procedure does not return by NRETURN; see the description of NRETURN, page 90.

A PROCEDURE RETURNING BY -NRETURN- MUST SUPPLY A NAME AS ITS VALUE. When a procedure returns by NRETURN, the value of the result variable must be a string or an object of datatype Name; see the description of NRETURN, page 90.

VARIABLE TO THE LEFT OF A [DOES NOT CONTAIN AN ARRAY. The value of the family part of an item reference is not of datatype Array. See the description of item references, page 101.

TOO MANY SUBSCRIPTS IN AN ARRAY REFERENCE. There are more index expressions in the selector of an item reference than there are dimensions defined for the family being indexed. See pages 106 and 109.

TOO FEW SUBSCRIPTS IN AN ARRAY REFERENCE. There are fewer index expressions in the selector of an item reference than there are dimensions defined for the family being indexed. See pages 106 and 109.

ILLEGAL CHARACTER IN ARRAY PROTOTYPE. See the description of the argument for the ARRAY() procedure, page 104.

SYNTAX ERROR IN ARRAY PROTOTYPE. See page 104.

LOWER BOUND GREATER THAN UPPER BOUND IN ARRAY PROTOTYPE. See page 104.

AN ARRAY BOUND WAS TOO LARGE. An expression for an upper or lower bound in an Array prototype was greater in magnitude than 131,071.

AN ARRAY DIMENSION WAS TOO LARGE. The difference between any pair of upper and lower bounds was greater in magnitude than 131,071.

AN ARRAY MUST CONTAIN FEWER THAN 2**17 ELEMENTS. A prototype string for the ARRAY() procedure specifies an array containing more than 131,071 items.

SYNTAX ERROR IN SELECTOR FOR ITEM(). See the description of the ITEM() procedure, page 108.

SYNTAX ERROR IN DATA PROTOTYPE. See the description of the argument of the DATA() procedure in Appendix A, section II.A.

DUPLICATE NAMES IN DATA PROTOTYPE. Two fields defined for objects of a single datatype may not have the same name, nor may a field name be the same as the datatype — otherwise all the necessary procedures could not exist simultaneously. See the description of DATA() in Appendix A, section II.A.

DATA CONSTRUCTOR CANNOT SUPPLY A NAME. Structure creation procedures, predefined or programmer-defined, do not return Names, but rather objects of datatype Array or of a programmer-defined datatype, respectively.

THE PARAMETER FOR A FIELD FUNCTION WAS NOT A DATA REFERENCE. The argument of a programmer-defined field selection procedure was not an object of a programmer-defined datatype.

NO SUCH FIELD IN THE REFERENCED DATA STRUCTURE. The structure which is the argument of a programmer-defined field selection procedure does not contain a field identified by that procedure name.

FILE SPECIFIED TO I/O PROCEDURE MUST BE CURRENTLY ATTACHED. The filesets named by the arguments of ENDGROUP(), REWIND(), EORLEVEL(), and EOI() must be currently associated with some variable (see Appendix A, section II.D).

ILLEGAL FILENAME GIVEN TO I/O ASSOCIATION PROCEDURE. A legal SCOPE fileset name is a string of one to seven letters and digits, beginning with a letter (see Appendix A, section II.D).

ATTEMPT TO READ PAST END-OF-INFORMATION. See the descriptions of EORLEVEL() and EOI() in Appendix A, section II.D.

STRING TO BE DISPLAYED WAS LONGER THAN 80 CHARACTERS. The string which is the argument to the REMARK() procedure must contain 80 or fewer characters.

ONLY STRINGS MAY BE OUTPUT. A value of a datatype other than String or Integer was assigned to a variable which currently has an output association.

THE MAXIMUM FIELD LENGTH HAS BEEN EXCEEDED. The program requires more storage to execute than was requested.

THE MAXIMUM STRING LENGTH HAS BEEN EXCEEDED. See the description of MAXLNGTH() in Appendix A, section II.B.

THE STATEMENT LIMIT HAS BEEN EXCEEDED. See the description of STLIMIT() in Appendix A, section II.B.

COMPILER STACK OVERFLOW, SIMPLIFY THE CONSTRUCTION. A storage area for intermediate results in the Snobol compiler has been exhausted. The statement should be rewritten as two or more statements, since it contains too many levels of nested parentheses.

Appendix M. Non-standard Features of Berkeley Snobol

The initial design and implementation of Snobol4 was done at Bell Telephone Laboratories for IBM System 360 machines. The latest version of this implementation is described in The SNOBOL4 Programming Language by R. E. Griswold, J. F. Poage, and I. P. Polonsky (second edition, Prentice-Hall, 1971). This book contains many interesting examples and should be of use to all serious Snobol programmers, even those who are working with non-standard implementations for different machines.

The implementation described here was produced at the Computer Center of the University of California at Berkeley by Paul McJones and Charles Simonyi for CDC 6000 series machines. The language they implemented, which we shall call the Berkeley version, is non-standard since it differs from the Bell version in three basic ways: some features of the language are handled differently, some features are absent, and some new features not present in the Bell version are provided. This appendix describes the differences between the Bell version and the Berkeley version, presenting the information in terms of these three types of differences. It is provided to make this more comprehensible description of the Snobol language useful to those writing programs in the Bell version, and to specify which parts of the Bell documentation are useful for those writing programs in the Berkeley version of the language.

Quite apart from differences between the two versions of the Snobol language, there are some differences in terminology between the documentation of Griswold, Poage, and Polonsky, and the present description. The pairs of terms in the following table are equivalent, and represent differences in the descriptions only, not in the language versions described.

Bell description	this description
primitive	predefined
defined	programmer-defined
function	procedure
predicate	test procedure
value of function name	value of result variable
formal argument	formal variable
local variable	internal variable
function procedure	procedure body
entry point	entry label

Bell description	this description
explicit name	string name
created name	Name
implicit name	Name
generated variable	indirect reference
aggregate	family
referencing argument	selector
array element	array item
array reference	item reference
field function	field selection procedure
source program	program text
statement component	statement part
subject (assignment)	left side
subject (pattern match)	string reference
object	right side
compilation error	compile-time error
program error	execution-time error

I. Features which are Handled Differently

Procedures. In the Bell version, it is an execution-time error to call a predefined procedure with more arguments than its definition prescribes; in the Berkeley version, extra arguments to all procedures are evaluated but otherwise ignored.

Since the character sets of IBM System 360 machines and CDC 6000 series machines are different, the ALPHABET() procedure, which returns a string specifying the character set in standard collating sequence, necessarily returns a different string in the two versions. (This procedure exists as a keyword in the Bell version.)

Since the Bell system uses FORTRAN IV I/O, and the Berkeley system does its own I/O, the INPUT() and OUTPUT() procedures require quite different sorts of arguments.

The ARRAY() procedure has two arguments in the Bell version, the second specifying an initial value to be assigned to all items of an array. In the Berkeley version, the ARRAY() procedure has one argument only; all items are initialized to the null value.

Since numeric strings are of datatype Integer in the Berkeley version, IDENT('1',1) succeeds while in the Bell version it fails. In the Bell version, patterns are considered identical only if they are indeed the same pattern. Thus

```
X = A | B
Y = A | B
IDENT(X,Y)
```

fails since two different copies of the pattern are being compared. In the Berkeley version this comparison would succeed, since patterns with the same structure are considered identical. IDENT(.VAR,'VAR') fails in the Berkeley version while it succeeds in Bell owing to the different implementations of the Name operator (described in the section on operators below).

The CODE() procedure in the Berkeley version does not allow labels to be redefined; consequently the labels of the statements which are to be added to the program during execution must be different from any existing labels of the program.

The Bell version provides more datatypes than does the Berkeley version and much more flexibility about converting from one datatype to another. In the Bell version, the CONVERT() procedure which is used for this purpose has two arguments; the second argument specifies the datatype to which the first argument is to be converted. In the Berkeley version the CONVERT() procedure has only one argument since only a limited kind of conversion is available. If the single argument of CONVERT() is a numeral string or an integer, it is converted into a real number; if the single argument is a real number, it is converted into a string.

<u>Operators.</u> The interrogation operator (?) has been implemented as the IF() procedure (see Appendix A, section II.C).

The unary operator * is called in the Bell version the unevaluated expression operator, and expressions introduced by it are of datatype Expression. This operator is defined more narrowly in the Berkeley version. It is called the deferred evaluation operator, and may be applied to simple variables only; thus *EQ(X,Y) causes an execution-time error. The datatype Expression is not defined in the Berkeley version; expressions introduced by the deferred evaluation operator are of datatype Pattern. Hence LEN(*V) causes an execution-time error since the argument of LEN() cannot be a Pattern.

In the Bell version when the name operator is applied to a natural variable it returns an object of datatype

String, but when applied to a created variable it returns an object of datatype Name. In the Berkeley version, the name operator always returns an object of datatype Name.

In the Bell version the multiplication operator has higher precedence than the division operator; in the Berkeley version the precedence is the same.

Keywords. There are no keywords in the Berkeley version (and hence no keyword operator). Some of the Bell keywords assume the form of procedures; these are listed in the table below.

Bell version	Berkeley version
&ALPHABET	ALPHABET()
&ANCHOR	ANCHOR()
&FNCLEVEL	FNCLEVEL()
&MAXLNGTH	MAXLNGTH()
&STCOUNT	STCOUNT()
&STLIMIT	STLIMIT()

These procedures are described in Appendix A, section II.

Datatypes. In the Berkeley version, numeric strings are of datatype Integer. Numeric strings may have an initial sign and hence the single characters '+' and '-' in isolation have the datatype Integer and have the value zero when used in arithmetic contexts. Correspondingly, the null value is of datatype Integer. In the Bell version, the null value is called the null string and is of datatype String.

System Transfers. In the Berkeley version, RETURN, FRETURN, NRETURN, and END are treated as system transfers, having the same predefined meanings as in Bell. They may be used as any other labels in the program text, however, in which case the special system meaning is lost.

Output. Objects of datatype other than String or Integer cannot be printed in the Berkeley version, and an attempt to print such a value results in an execution-time error. In the Bell version an attempt to print such a value results in the printing of a string designating the datatype of the value.

Assigning the variable OUTPUT a value of more than 132 characters in the Berkeley version results in only the first 132 being printed (a single line); in the Bell version, as many lines as necessary are printed.

Program Representation. There are a number of small differences in the way that programs may be represented; most consist of extra optional features which have been added to the Berkeley version.

In the Berkeley version, the assignment sign (=) need not be bounded by blanks; similarly, the colon introducing a go-to need not be preceded by a blank.

In the Berkeley version, the quote sign used as a literal delimiter may appear within that literal in pairs; each pair is then treated as representing a single quote. Thus 'DON''T' may be used to represent the string DON'T.

In the Berkeley version, statements continued over line boundaries may be broken anywhere; a blank is never assumed at the point of the break. In the Bell version, statements may be broken only where a blank is required.

In the Berkeley version, real literals need not begin with digits (that is, they may begin with an initial decimal point).

In the Berkeley version it is not necessary to terminate a program text with a statement labelled END as it is in the Bell version. The program may terminate by taking a transfer to END, if no END label is present. END may be used as a label in a program text in which case it then loses its system significance, and a program containing an END label can terminate only by running out of program text; this is not an error as it is in Bell (see Chapter 3). In the Berkeley version it is not possible to specify by use of an END statement which statement of the program is to be executed first; execution always begins with the first statement of the program text.

Alternative characters may be used in the Berkeley version to represent some of those which must otherwise be multiple punched on an 026 keypunch. Thus the go-to may be introduced by either a colon (:) or a slash (/). (If the slash is used it must not be followed by any blanks as it might then be indistinguishable from the binary division operator.) The colon used as a delimiter between the upper and lower bounds of an index in forming the prototype of an array may also be represented by a slash. The alternation operator (|) may be represented by two slashes (//) and the square brackets of an item reference may be represented by (/ for an open bracket and /) for a close bracket. The Bell version does not provide any of these particular options, but has a different extended syntax to take advantage of special characters available on the IBM 360; lower case letters are also available.

The representation of labels is freer in the Bell version than in the Berkeley version. In the Bell version a label may consist of a letter or a digit followed by any number of other characters from the entire character set except blank. In the Berkeley version a label must be an identifier; that is, it must begin with a letter and consist of nothing but letters, numbers, and periods.

The Program Listing. In the Berkeley version, columns 72 and 73 of the program text are separated by ten spaces in the output listing. The statement numbers always appear to the left of the statements. In the Bell version the statement numbers normally appear to the right of the statements, but it is possible to specify that they appear to either the left or the right. This is done by writing the terms LEFT or RIGHT following the listing directive LIST; the default option is RIGHT. There is no way to specify that the statements should be numbered to the right in the Berkeley version.

In the Berkeley version the listing directive SPACE has been added to cause one blank line to appear in the listing.

II. Features Absent from the Berkeley Version

Procedures. The following procedures are available in the Bell version but not in the Berkeley version. Unless otherwise indicated, their actions cannot be simulated.

ARG() returns the name of the n-th argument in the declaration of a programmer-defined procedure.

BACKSPACE() backspaces a file one logical record.

CLEAR() causes all natural variables to be assigned the null value. This procedure can be written in Berkeley Snobol using NEXTVAR().

COLLECT() forces a storage regeneration. (Not needed since storage regeneration occurs automatically.)

COPY() produces a copy of an array or a data structure. It can be written in Berkeley Snobol using ITEM() for arrays (see Chapter 7), and APPLY() for data structures.

DUMP() produces an unalphabetized list of all non-null natural variables and their values. It can be written in Berkeley Snobol using NEXTVAR().

DUPL() returns a string consisting of n duplications of one of its arguments. It is virtually the same as the programmer-defined procedure REPEAT() given in Chapter 6.

EVAL() returns the result of evaluating a string which is a Snobol expression or an object of datatype Expression.

FIELD() returns the name of the n-th field in the declaration of a programmer-defined datatype. It can be written in Berkeley Snobol, because the Berkeley PROTOTYPE() procedure may be applied to structures (see Appendix A, Section II.B).

INTEGER() succeeds if its argument is an integer. It can be easily written as

```
IDENT(DATATYPE(ARG),'INTEGER')
```

(In the same way, any other test procedure for testing datatypes may be written.)

LOAD() causes an external function to be loaded from the library during execution.

LOCAL() returns the name of the n-th local (internal) variable of a programmer-defined procedure.

OPSYN() allows the programmer to specify synonyms for procedures or operators. Thus the same procedure may be referred to by more than one name and the same operator by more than one symbol. In addition, operators and procedures may be made synonymous; thus this procedure makes possible the definition of new operators.

REMDR() returns the integer remainder of dividing its first argument by its second. This can be written in Snobol as a programmer-defined procedure employing nothing but arithmetic operators.

REPLACE() returns a string in which every character of one argument has been replaced by a corresponding character of another argument. It can be written as a programmer-defined procedure in Snobol.

STOPTR() cancels the tracing of the variable named by its argument.

TABLE() creates a family of variables, similar to a one-dimensional array except that individual variables may be selected in terms of any data object, not just integers. This datatype is not defined in the Berkeley version, but table-like structures can be formed using indirect referencing if the selector is a string.

TRACE() initiates tracing of the variable named by its argument.

UNLOAD() causes the unloading of an external library function which is no longer needed.

VALUE() has the same effect as the indirect referencing operator when applied to a String or a Name, but if VALUE has been defined to be a field of a structure, then it may have an argument of that datatype as well.

<u>Operators.</u> The following operators are not available:

negation (¬)
cursor position (@)
exponentiation (**)

The negation operator fails if its operand succeeds, and succeeds if its operand fails. (Its counterpart, the interrogation operator (?), which always succeeds, has been implemented as the IF() procedure.)

The cursor position operator has a variable as its operand and is used within the pattern part of a rule. The variable is assigned, by immediate assignment, an integer representing the position of the cursor when pattern matching occurs. Thus

```
'ABC'   'B'   @POINTER
```

causes POINTER to be assigned successively the values 0 and 1.

<u>Keywords.</u> The Berkeley version of Snobol contains no keywords. Some keywords have been implemented as predefined procedures, as indicated in Section I of this appendix; the remaining keywords, listed below, cannot be simulated, although sometimes a similar effect may be achieved through other means. Those whose values are protected (i.e., cannot be changed directly by the programmer) are marked with an asterisk.

&ABEND is used to specify whether or not a system core dump is to be printed at program termination.

&ABORT has the same value as that of the predefined pattern ABORT. (*)

&ARB has the same value as that of the predefined pattern ARB. (*)

&BAL has the same value as that of the predefined pattern BAL. (*)

&CODE can be assigned an integer which will be returned to the operating system as the user completion code at program termination.

&DUMP is used to specify whether or not a dump of the natural variables is to be printed at program termination.

&ERRLIMIT has a value which controls the handling of certain program errors.

&ERRTYPE acquires an integer code identifying the type of any program error which may occur. (*)

&FAIL has the same value as that of the predefined pattern FAIL. (*)

&FENCE has the same value as that of the predefined pattern FENCE. (*)

&FTRACE is used to specify whether or not diagnostic tracing information is to be provided on calls to and returns from all programmer-defined procedures.

&FULLSCAN is used to specify whether or not the fullscan mode of pattern matching (in which no heuristics are employed) is to be used.

&INPUT is used to specify whether or not any input is to cccur.

&LASTNO acquires as its value an integer specifying the statement number of the previous statement executed. (*)

&OUTPUT is used to specify whether or not any output is to occur.

&REM has the same value as that of the predefined pattern REM. (*)

&RTNTYPE acquires as value the string RETURN, FRETURN, or NRETURN, depending on the type of return made by the last programmer-defined procedure which returned. (*)

&STFCOUNT acquires as value an integer specifying how many statements have failed. (*)

&STNO acquires as value an integer specifying the statement number of the statement currently being executed.

&SUCCEED has the same value as that of the predefined pattern SUCCEED. (*)

&TRACE is used to specify whether or not tracing is to occur.

&TRIM is used to specify whether or not all trailing blanks are to be trimmed on input.

Pattern Variables. The predefined pattern variable SUCCEED, which always matches the null value (and which has very limited practical application) is not available.

Datatypes. The following datatypes do not exist in the Berkeley version:

Table (see the description of the TABLE() procedure above)

Expression (see the description of deferred evaluation in section I of this appendix)

External, which refers to external library functions (see the description of the LOAD() and UNLOAD() procedures above).

Pattern matching. There is no quickscan mode of pattern-matching (a mode which makes use of heuristics). This is the normal mode in the Bell version, while fullscan is the normal mode in the Berkeley version.

Arithmetic. Mixed mode arithmetic or comparisons (involving integers and real numbers) are not permitted.

Output. The variable PUNCH has a predefined association with the punch file in the Bell version; this is not true of the Berkeley version, but the association can be made by simply executing the rule

```
OUTPUT('PUNCH','PUNCH')
```

The Berkeley version currently provides no compile-time error messages and no program statistics. As is indicated by the foregoing, it also provides no tracing facilities and no dump.

III. Features not Present in the Bell Version.

Procedures. The following predefined procedures have been added to the Berkeley version; all are described more fully in Appendix A.

CLOCK() returns the 24-hour time of day (e.g. 17:00:59). (See Appendix A, section II.B.)

TYPE() returns the same result as DATATYPE() for objects of predefined datatypes, and the string DATA for all objects of programmer-defined datatypes. (See Appendix A, section II.B.)

ITEM() has been made more flexible and more useful in the Berkeley version than it is in the Bell version. It is described in detail in Chapter 7.

PROTOTYPE() has been significantly extended so that it may be applied to structures, Patterns, and Names, as well as to Arrays. (See Appendix A, section II.B.)

A number of field selection procedures have been added for use in conjunction with the systems-defined "prototypes" of Patterns and Names which are returned by the PROTOTYPE() procedure. The procedures PARAM(), FIRST(), REST(), LEFT(), and RIGHT() may be used to decompose Patterns into the objects from which they were constructed. A similar service for Names is provided by the procedures RIGHT(), FAMILY(), and SELECTOR(). (See Appendix A, section I.C.)

NEXTVAR() returns the names of all members of any family cyclically, treating the set of all non-null natural variables as a "family." (See Appendix A, section II.B.)

INDEX