Language
g Language
ng Language
ing Language
ming Language
mming Language
amming Language
ramming Language
gramming Language
ogramming Language
rogramming Language
Programming Language
Programming Language
4 Programming Language
L4 Programming Language
OL4 Programming Language
BOL4 Programming Language
OBOL4 Programming Language
NOBOL4 Programming Language
SNOBOL4 Programming Language
SNOBOL4 Programming Language
e SNOBOL4 Programming Language
he SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language
The SNOBOL4 Programming Language

THE SNOBOL4 PROGRAMMING LANGUAGE

R. E. Griswold
J. F. Poage
I. P. Polonsky

THE SNOBOL4 PROGRAMMING LANGUAGE

R. E. Griswold
J. F. Poage
I. P. Polonsky
Bell Telephone Laboratories, Incorporated

Prentice-Hall, Inc., Englewood Cliffs, New Jersey

Current printing (last digit): 10 9 8 7 6 5 4

13-815357-4

Library of Congress Catalog Card Number: 79-77614

Printed in the United States of America

Preface

SNOBOL4 is a computer programming language containing many features not commonly found in other programming languages. It evolved from SNOBOL [1,2,3][1], a language for string manipulation, developed at Bell Telephone Laboratories, Incorporated, in 1962. Extensions to SNOBOL through various versions have made it a useful tool in such areas as compilation techniques, machine simulation, symbolic mathematics, text preparation, natural language translation, linguistics, and music analysis.

The basic data element of SNOBOL4 is a string of characters, such as this line of printing. The language has operations for joining and separating strings, for testing their contents, and for making replacements in them. If a string is a sentence, it can be broken into phrases or words. If it is a formula, it can be taken apart into components and reassembled in another format. A string can appear either as a literal or as the value of a variable. The literal form is indicated by enclosing the string in quotation marks:

```
        'THIS IS A STRING'
```

The string value may be assigned to a variable:

```
        LINE   =   'THIS IS A STRING'
```

A common operation on a string is examination of its contents for a desired structure of characters. This structure, known as a pattern, can be as simple as a string or a given number of characters. A pattern also can be an extremely complicated expression consisting, for example, of a number of alternatives followed by another set of alternatives, all of which must begin a given number of characters from the end of the string. The pattern, as a data type, may also appear either in literal or variable form. The data type of a variable - string, pattern, or any other in the language - depends on the last value assigned to it. There are no type declaration statements for variables as in other programming languages.

SNOBOL4 provides numerical capabilities with both integers and real numbers. Because the language is essentially character oriented, the facilities are not extensive. Since most numerical operations with strings involve character counting, integers are much more commonly used, with conversion to and from strings performed automatically as required.

Often it is desirable to associate a group of items with one variable name through numeric indexing. The SNOBOL4 array provides this capability with more flexibility than most programming languages. An array is a data element consisting of a set of pointers to other data elements, so that each array element may be any data type, even an array. Several other system-defined data types are also included.

Execution of SNOBOL4 programs is interpretive. Instead of compiling a program into actual computer instructions, the compiler translates the program into a notation the interpreter can easily execute. This makes it fairly simple to provide capabilities such as tracing of new values for variables, an operation that is quite difficult in noninterpretive systems. Another important product of interpretation is flexibility. Functions can be defined and redefined during program execution. Function calls can be made recursively with no special program notation. The language is extendable to new data types needed for a program through data type definition operations. Linked-list nodes and complex numbers are possible programmer-defined data types. Operations on these new data types can be defined as functions.

----

[1]Numbers in brackets refer to references listed at the end of this manual.

This manual is an instructional and reference guide, and provides many examples of usage of the language. The description of the language is complete and does not require familiarity with earlier versions of the language. Some familiarity with elementary concepts of programming is presumed, however.

Foreword

The SNOBOL4 programming language has been developed over a period of years and new language features have been added from time to time during the course of this development. Consequently there are several somewhat different versions of the language in use. The description in this manual corresponds to Version 2.0 (October 7, 1968).

SNOBOL4 has been implemented on several different computers, including the IBM System/360, the UNIVAC 1108, the GE 635, the CDC 6000 series, and the RCA Spectra 70 series. Implementations for other machines are in various stages of completion. These machines have different operating environments and character sets. As a result, implementations of SNOBOL4 vary from machine to machine in details of syntax, operating system interface, and so forth. This manual corresponds to the implementation of SNOBOL4 for the IBM System/360 operating under OS. Sections of the manual containing language features particularly dependent upon this implementation make specific reference to this dependency.

Programs contained in this manual were run on an IBM 360 Model 65.

Acknowledgments

The authors' most pleasant responsibility is the acknowledgement of the assistance provided in the course of the design, implementation, and documentation of the SNOBOL4 language.

The ideas of many individuals have helped shape the form of SNOBOL4. Particularly valuable contributions have been made by Messrs. B. N. Dickman, D. J. Farber, P. D. Jensen, M. D. McIlroy, R. F. Rosin, M. A. Seelye, and M. D. Shapiro.

The authors have been fortunate in having the assistance of a number of people during various stages of the implementation of SNOBOL4. Mr. R. A. Yates designed and implemented the storage allocation and regeneration techniques used in SNOBOL4. Mr. Yates also contributed many useful ideas to the overall design of the system. Messrs. B. N. Dickman and P. D. Jensen designed and implemented the tracing facilities and provided many valuable suggestions for improving the system. Special thanks are due Mr. L. C. Varian for his assistance in preparing the initial implementation for the IBM System/360.

The authors would like to express their appreciation to Mrs. R. E. Griswold who has given freely of her time to prepare much of the machine-readable material used in the development of the SNOBOL4 langauage and its documentation.

Mr. J. F. Gimpel has made an important contribution to the documentation of the language. The present document reflects much of his thinking on the organization and presentation of descriptive material. Several of the programs used in the examples are his.

Mr. M. A. Seelye provided an an unusually thorough and perceptive criticism of a preliminary draft of this manual, enabling the authors to correct many errors and clarify a number of obscure points.

The authors' special thanks go to Mrs. P. M. Hammer and Mr. M. D. Shapiro for their help in preparing this manual. Their editorial competence and unfailing good humor have made the laborious process of preparing this manuscript a bearable, if not rewarding, experience.

Contents

Chapter 1. Introduction to the SNOBOL4 Programming Language

This chapter is an introductory overview of the SNOBOL4 programming language. It describes the format of statements, some of the operations, and some of the types of data handled by the language. Later chapters describe in more detail much of the material in this introductory chapter.

A SNOBOL4 program consists of a sequence of statements. There are four basic types of statements:

1) the assignment statement,
2) the pattern matching statement,
3) the replacement statement, and
4) the end statement.

The end statement terminates the program.

A. Assignment Statements and Basic Data Types

The simplest type of statement is the assignment statement. It has the form

```
     variable   =   value
```

The assignment statement may be said to have the following meaning: "Let variable have the given value." For example, let V have the value 5, or

```
     V    =    5
```

The value may be given by an expression, consisting, for example, of arithmetic operations as in the statement

```
     W    =    14 + (16 - 10)
```

which assigns the value 20 to the variable W. Blanks are required around arithmetic operators such as + and - . The value need not be an integer, which is just one type of data handled by SNOBOL4. For example, the value may be a string of characters, indicated by enclosing quotes. An example is the assignment statement

```
     V    =    'DOG'
```

which assigns the string DOG to the variable V. Various types of data and operations that may be performed on them are described later.

Typically a variable is a name such as V, X, or ANS. Variables appearing explicitly in a program must begin with a letter which may be followed by any number of letters, digits, periods, and underscores.

The value of a variable may be used in an assignment statement. Thus

```
     RESULT   =   ANS.1
```

assigns to the variable RESULT the value of ANS.1 . (Quotation marks distinguish literal strings from variables.)

Blanks are required to separate the parts of a statement. In an assignment statement, the equal sign must be separated from the variable on the left and the value on the right by at least one blank.

A statement which is longer than one line can be continued onto successive lines by starting the continuation lines with a period or plus sign. An example is

```
     N   =   (3 + M)   (2 + SUM)  -
.   (F - 2)
```

When continuing a statement over a line boundary, the statement may be broken wherever a blank is required.

Several statements may be placed on one line by using semicolons which indicate the ends of statements. An example is

```
     X   =   2;   Y   =   3;   Z   =   10
```

A line beginning with an asterisk is treated as a comment and does not affect the operation of the program.

1. <u>Integers</u>

The arithmetic operations of addition, subtraction, multiplication, division, and exponentiation of integers may be used in expressions. The statements

```
     N   =   5;   M   =   4
     P   =   N * M / (N - 1)
```

assign the value 5 to P. While blanks are required between the binary operators and their operands, unary operators such as the minus sign must be adjacent to their operands. An example is the statement

```
     Q2   =   -P / -N
```

which assigns the value 1 to Q2 .

Arithmetic expressions can be arbitrarily complex. When evaluating arithmetic expressions, the natural order of operator precedence applies. The unary operations are performed first, then exponentiation (**), then multiplication, followed by division, and finally addition and subtraction. All operations associate to the left except exponentiation. Hence,

```
        X   =   2 ** 3 ** 2
```

is equivalent to

```
        X   =   2 ** (3 ** 2)
```

Parentheses may be used to emphasize or alter the order of evaluation of an expression.

In the above examples all the operands are integers and the results are integers. The quotient of two integers is also an integer. The remainder is discarded. Thus

```
        Q1   =   5 / 2
        Q2   =   5 / -2
```

give Q1 and Q2 the values 2 and -2, respectively. Similarly,

```
        MOD   =   N - (N / M) * M
```

gives MOD the value N modulo M if N and M are positive integers.

2. Real Numbers

Arithmetic expressions involving real operands are also permitted in assignment statements. The statements

```
        PI   =   3.14159
        CIRCUM   =   2. * PI * 5.
```

assign real values to PI and CIRCUM.

There are several limitations on real arithmetic in SNOBOL4. Exponentiation involving reals is undefined and causes execution of the program to terminate with an error message. Operations involving mixed types of numbers are not permitted, and also cause execution of the program to terminate.

3. Strings

Expressions involving operands that are character strings are also permitted in assignment statements. For example, the assignment statement

```
        SCREAM   =   'HELP'
```

assigns the string HELP as the value of SCREAM .

The string is specified by enclosing it within a pair of quotation marks. Any character may appear in a string. A pair of double quotation marks can be used instead of single quotation marks. This permits the use of quotation marks within a string as in the statements

```
        PLEA    =   'HE SHOUTED, "HELP."'
        QUOTE    =    '"'
        APOSTROPHE    =    "'"
```

The Null String

The null string, which is a string of length zero, is frequently used in SNOBOL4. With a few exceptions, explained later, all variables have the null string as their initial value. A variable can also be assigned the null string by a statement like

```
        NULL    =    ''
```

or, more briefly,

```
        NULL    =
```

The variable NULL is used in many examples that follow to represent the null string.

The null string is different from the following strings, each of which has length one:

```
        '0'
        " "
```

Strings in Arithmetic Expressions

Numeral strings can be used in arithmetic expressions with integers. For example, as a result of the statements

```
        Z    =    "10"
```

Reals cannot be used.

```
        X   =   5 * -Z + '10'
```

X has the value -40. Numeral strings contain only digits and perhaps a preceding sign. Thus, the following strings cannot be used in arithmetic expressions:

```
        '3.257'
        '1,253,465'
        '.364 E-03'
```

They cause execution of the program to terminate with the comment "ILLEGAL DATA TYPE."

Strings cannot be used in expressions involving real numbers.

The null string is equivalent to the integer zero in arithmetic expressions.

String-Valued Expressions

Concatenation is the basic operation for combining two strings to form a third. The following statements illustrate the format of an expression involving concatenation.

```
        TYPE    =    'SEMI'
        OBJECT    =    TYPE 'GROUP'
```

The resulting value of OBJECT is the string SEMIGROUP . Notice there is no explicit operator for concatenation. Concatenation is indicated by specifying two string-valued operands separated by at least one blank.

```
        FIRST    =    'WINTER'
        SECOND    =    'SPRING'
        TWO.SEASONS    =    FIRST ',' SECOND
```

are equivalent to

```
        TWO.SEASONS    =    'WINTER,SPRING'
```

Strings can also be concatenated with integers as in

```
        ROW    =    'K'
        NO.    =    22
        SEAT    =    ROW NO.
```

which gives SEAT the value K22 .

In an expression involving concatenation and integer arithmetic, concatenation has the lowest precedence. Thus

```
        SEAT    =    ROW NO. + 4 / 2
```

is equivalent to

```
        SEAT    =    ROW (NO. + (4 / 2))
```

or

```
        SEAT    =    'K24'
```

Input and Output of Strings

Three variables provide means for reading and writing data. The variables OUTPUT and PUNCH are for printing and punching. Whenever either of them is assigned a string or integer value, a copy of the value is put out.

```
        OUTPUT    =    'THE RESULTS ARE:'
```

assigns THE RESULTS ARE: to OUTPUT and also prints it.

```
        PUNCH    =    OUTPUT
```

causes the same line to be punched on a card. The statements

```
        OUTPUT    =
        PUNCH    =
```

cause a blank line to be printed and a blank card to be punched.

The variable INPUT is used for reading in strings. Each time the value of INPUT is required in a statement, another card is read in and the 80-character string on it is assigned as the value of INPUT. Thus

```
        PUNCH    =    INPUT
```

punches a copy of the input card.

B. Pattern Matching Statements

The operation of examining substrings for the occurrence of specified substrings (i. e. pattern matching) is fundamental to the SNOBOL4 language. Pattern matching can be specified in two types of statements:

1) the pattern matching statment, and
2) the replacement statement.

The pattern matching statement has the form

```
subject   pattern
```

where the two fields are separated by at least one blank. The subject specifies a string that is to be examined, and the pattern can be thought of as specifying a set of strings. The statement causes the subject string to be scanned from the left for the occurrence of a string specified by the pattern.

If

```
TRADE   =   'PROGRAMMER'
```

the statement

```
TRADE   'GRAM'
```

examines the value of TRADE for an occurrence of GRAM . If

```
PART   =   'GRAM'
```

then an equivalent statement is

```
TRADE   PART
```

The following example illustrates a pattern matching statement in which the pattern is a string-valued expression.

```
ROW    =   'K'
NO.    =   20
'K24'    ROW   NO. + 4
```

The subject is a literal and the value of the expression is the string K24 .

Notice that there is no explicit pattern matching operator between the subject and the pattern. The two fields are separated by blanks.

If it is necessary to have concatenation in the subject, the expression must be enclosed within parentheses to avoid ambiguity. An example is

```
TENS    =   2
UNITS   =   5
(TENS UNITS) 30
```

On the other hand, a pattern formed by concatenation does not need parentheses. The following statements are equivalent:

equivalent statements

```
TENS UNITS 30
```

```
TENS (UNITS 30)
```

C. <u>Replacement Statements</u>

A replacement statement has the form

<u>subject</u>   <u>pattern</u>   =   <u>object</u>

where the fields are separated by at least one blank. If the pattern matching operation succeeds, the subject string is modified by replacing the matched substring by the object. For example, if

```
WORD   =   'GIRD'
```

then the replacement statement

```
WORD  'I'   =   'OU'
```

causes the subject string GIRD to be scanned for the string I and then, since the pattern matches, I is replaced by OU . Hence WORD has as value the string GOURD . If the statement is

```
WORD 'AB'   =   'OU'
```

the value of WORD does not change because the pattern fails to match.

Another example of the use of replacement statements is given in the following sequence of statements

```
HAND   =   'AC4DAHKDKS'
RANK   =   4
SUIT   =   'D'
HAND  RANK SUIT   =   'AS'
```

which replaces the substring 4D with the string AS .

A matched substring is deleted from the subject string if the object in the replacement statement is the null string. Thus

```
HAND  RANK SUIT   =
```

deletes 4D from HAND leaving it with the string ACAHKDKS as value.

D. Patterns

The patterns in the preceding examples specify single strings. It is also possible to specify more complex patterns. There are two operations available for constructing such patterns:

1) alternation, and
2) concatenation.

Alternation is indicated by an expression of the form

```
    P1 | P2
```

where the two patterns P1 and P2 are separated from the | by blanks. The value of the expression is a pattern structure that matches any string specified by either P1 or P2. For example, the statement

```
    KEYWORD   =   'COMPUTER' | 'PROGRAM'
```

assigns to KEYWORD a pattern structure that matches either of these two strings.

Subsequently, KEYWORD may be used wherever patterns are permitted. For example,

```
    KEYWORD   =   KEYWORD | 'ALGORITHM'
```

gives KEYWORD a new pattern value equivalent to the value assigned by executing the statement

```
    KEYWORD   =   'COMPUTER' | 'PROGRAM' | 'ALGORITHM'
```

Similarly,

```
    TEXT  KEYWORD   =
```

examines the value of TEXT from the left and deletes the first occurrence of one of the alternative strings. If

```
    TEXT   =   'PROGRAMMING ALGORITHMS FOR COMPUTERS'
```

the result of the replacement statement is as if the following statement were executed:

```
    TEXT   =   'MING ALGORITHMS FOR COMPUTERS'
```

Concatenation of two patterns, P1 and P2, is specified in the same way as the concatenation of two strings:

```
        P1  P2
```

That is, the two patterns are separated by blanks. The value of the expression is a pattern that matches a string consisting of two substrings, the first matched by P1, the second matched by P2. For example, if

```
        BASE    =    'BINARY' | 'DECIMAL' | 'HEX'
        SCALE   =     'FIXED' | 'FLOAT'
        ATTRIBUTE   =    SCALE BASE
```

and

```
        DCL    =    'AREAFIXEDDECIMAL'
```

then the pattern match succeeds in the statement

```
        DCL   ATTRIBUTE
```

Concatenation has higher precedence than alternation. Thus

```
        ATTRIBUTE   =   'FIXED' | 'FLOAT' 'DECIMAL'
```

matches FIXED or FLOATDECIMAL . The order of evaluation may be altered by using parentheses.

```
        ATTRIBUTE   =   ('FIXED' | 'FLOAT') 'DECIMAL'
```

matches either FIXEDDECIMAL or FLOATDECIMAL .

E. Conditional Value Assignment

It is possible to associate a variable with a component of a pattern such that if the pattern matches, the variable is assigned the substring matched by the component. The operator . is the conditional value-assignment operator and it is used in an expression of the form

<u>pattern</u> . <u>variable</u>

where the operator is separated from its operands by blanks. For example

```
        BASE    =    ('HEX' | 'DEC') . B1
```

assigns to BASE a pattern that matches either HEX or DEC . If BASE is used successfully in a pattern match, the value of B1 is set to the substring matched by BASE .

The operator . has the highest precedence of all the operators and associates to the left. Thus

```
        A.OR.B   =   A  |  B  .  OUTPUT
```

is equivalent to

```
        A.OR.B   =   A  |  (B . OUTPUT)
```

which assigns to A.OR.B a pattern that matches the value of A or B . If B matches, the substring matched is printed.

There is also an operator $ for immediate value assignment which assigns value to a variable if the associated component of the pattern matches regardless of whether the entire pattern matches. Immediate value assignment is discussed in more detail later.

F. Flow of Control

A SNOBOL4 program is a sequence of statements terminated by an end statement. Statements are executed sequentially unless otherwise specified in the program. Labels and gotos are provided to control the flow of the program.

A statement may begin with a label, permitting transfer to the statement. For example, the assignment statement

```
START   TEXT   =   INPUT
```

has the label START . A label consists of a letter or a digit followed by any number of other characters up to a blank. Blanks separate the label from the subject. A statement with no label must begin with at least one blank. The end statement is distinguished by the label END, indicating the end of the program.

Transfer to a labelled statement is specified in the goto field which may appear at the end of a statement and is separated from the rest of the statement by a colon. Two types of transfers can be specified in the goto field: conditional and unconditional.

A conditional transfer consists of a label enclosed within parentheses and preceded by an F or S corresponding to failure or success goto. An example is the statement

```
        TEXT   =   INPUT      :F(DONE)
```

This statement causes a record to be read in and assigned as the value of TEXT. If, however, there is no data in the input file, i.e. an end of file is encountered, no new value is assigned to TEXT. Then, because of the failure to read, transfer is made to the statement labelled DONE.

A use of the success goto is illustrated in the following program which punches a copy of the input file.

```
LOOP    PUNCH   =    INPUT      :S(LOOP)
END
```

The first statement is repeatedly executed until the end of file is encountered and then the program flows into the end statement which causes the program to terminate.

The success or failure of a pattern match can also be used to control the flow of a program by conditional gotos. For example

```
        COLOR   =    'RED' | 'GREEN' | 'BLUE'
BRIGHT  TEXT    COLOR    =      :S(BRIGHT)F(BLAND)
BLAND
```

All occurrences of the strings RED, GREEN, and BLUE are deleted from the value of TEXT before the pattern fails to match. Control then passes to the statement labelled BLAND. Both success and failure gotos can be specified in one goto field, and may appear in either order.

For an example of an unconditional transfer, consider the following program that punches and lists a deck of cards.

```
LOOP    PUNCH   =    INPUT      :F(END)
        OUTPUT   =    PUNCH     :(LOOP)
END
```

The goto field in the second statement specifies an unconditional transfer.

G. Indirect Reference

Indirect referencing is indicated by the unary operator $. For example, if

```
        MONTH   =    'APRIL'
```

then $MONTH is equivalent to APRIL . That is, the statement

```
        $MONTH   =    'CRUEL'
```

is equivalent to

```
        APRIL   =    'CRUEL'
```

The indirect reference operator can also be applied to a parenthesized expression as in the statements

```
        WORD    =    "RUN"
        $(WORD ':')    =    $(WORD ':') + 1
```

which increment the value of RUN: .

In general, the unary operator $ generates a variable that is the value of its operand. The expression

```
        $("A" | "B")
```

causes the program to terminate with the message "ILLEGAL DATA TYPE" because the value of the operand of $ is a pattern, not a string. Indirect reference in a goto is demonstrated by

Concatenation

```
        N    =    N + 1        :($("PHASE" N))
```

If, for example, the assignment statement sets N equal to 5, then the transfer is to the statement labelled PHASE5 .

H. Functions

Many SNOBOL4 procedures are invoked by functions built into the system, called primitive functions. Operations that occur frequently are implemented as primitive functions for efficiency. Other primitive functions are used to invoke more complex operations that are fundamental to the language, affect parameters and tables internal to the system, and perform operations that could not be programmed in source language by other means. In addition, facilities are available for a programmer to define his own source-language functions.

1. Primitive Functions

Consider the function SIZE, which has a single string argument and returns as value an integer which is the length (number of characters) of the string. The statements

```
        APE     =    'SIMIAN'
        OUTPUT    =    SIZE(APE)
```

print the number 6 .

Arguments to all functions are passed by value, and an arbitrarily complex expression may be used in the argument. Thus the statements

```
        N    =    100
        OUTPUT    =    SIZE('PART' N + 4)
```

print the number 7 , because the value of the argument is the string PART104 .

The argument of SIZE is supposed to be a string. Therefore, a call of the form

```
     SIZE("APE" | "MONKEY")
```

causes the program to terminate with the diagnostic message "ILLEGAL DATA TYPE," because the value of the argument is a pattern.

TRIM is another function that performs an operation frequently required. TRIM(string) returns as value a string which is equal to the argument with trailing blanks removed. It is often used in a statement of the form

```
READ   TEXT   =   TRIM(INPUT)     :F(END)
```

which assigns as value to TEXT the string on the next input card, trimmed of trailing blanks. Notice that the use of the variable INPUT in the argument causes a card to be read.

REPLACE is a function called with three string-valued arguments.

```
     REPLACE(TEXT,CH1,CH2)
```

returns as value a string which is equal to TEXT with each occurrence of a character appearing in CH1 replaced by the corresponding character in CH2. For example, the statements

```
     STATEMENT   =   'A(I,J)    =   A(I,J)  + 3'
     OUTPUT   =   REPLACE(STATEMENT,'()','<>')
```

print the line

```
A<I,J>   =   A<I,J> + 3
```

If the last two arguments of the function call do not have the same length, the function fails. Function failure, like input failure, can be used in a conditional transfer.

Another example of the use of REPLACE is the following program that produces a simple cryptographic encoding of an input deck.

```
       INALPH   =   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
       OUTALPH   =   'KLMNOPQRSTUVWXYZABCDEFGHIJ'
LOOP   PUNCH   =   REPLACE(INPUT,INALPH,OUTALPH)     :S(LOOP)
END
```

The iteration is terminated by input failure.

There are also several functions that return patterns as their values. LEN is such a function. LEN(integer) returns a pattern that matches any string of the length specified by the integer.

The following example punches the value of STR centered on a card.

```
        BLANKS   =   '                                             '
        BLANKS   LEN((80 - SIZE(STR)) / 2) . PAD
        PUNCH    =   PAD STR
```

If the size of STR is greater than 80, the argument of LEN is negative, causing error termination with the message "NEGATIVE NUMBER IN ILLEGAL CONTEXT."

2. Predicates

A predicate is a function or operation that returns the null string as value if a given condition is satisfied. Otherwise it fails.

LE is an example of a predicate used for comparing integers.

```
        LE(N1,N2)
```

returns the null string as value if N1 is an integer less than or equal to N2. Thus

```
        PUNCH    =    LE(SIZE(TEXT),80) TEXT
```

punches the string TEXT if its length is not greater than 80. The null string value of the predicate does not affect the string that is punched. If the predicate fails, no assignment is made to PUNCH, and no card is punched.

The success or failure of a predicate can be used with a conditional goto to control the flow of a program. For example,

```
        N    =    0;    SUM    =    0
ADD     N    =    LT(N,50) N + 1       :F(DONE)
        SUM    =    SUM + N            :(ADD)
DONE    OUTPUT   =    SUM
```

sums the first 50 integers. Iteration continues as long as N is less than 50. When the predicate fails, the conditional transfer to DONE is performed and the string 1275 is printed.

There are several predicates for comparing strings. For example,

```
        DIFFER(ST1,ST2)
```

returns the null string as value if the values of two arguments are not identical. Thus

```
        OUTPUT   =    DIFFER(FIRST,SECOND) FIRST SECOND
```

concatenates the values of FIRST and SECOND if they are not the same, and then prints them.

For all functions, an omitted argument is assumed to be the null string. Thus

```
          PUNCH    =    DIFFER(TEXT)  TEXT
```

punches the value of TEXT if it is not the null string.

LGT is a predicate that lexically compares two strings.

```
          LGT(ST1,ST2)
```

succeeds if ST1 follows (is lexically greater than) ST2 in alphabetical order. The statements

```
          OUTPUT    =    LGT(TEXT1,TEXT2) TEXT2      :S(SKIP)
          OUTPUT    =    TEXT1
          OUTPUT    =    TEXT2                       :(JUMP)
SKIP      OUTPUT    =    TEXT1
JUMP
```

print the values of TEXT1 and TEXT2 in alphabetical order.

3. Defined Functions

The SNOBOL4 language provides the programmer with the capability to define functions in the source language. This feature facilitates the organization of a program and may improve its efficiency.

A programmer may define a function by executing the primitive function DEFINE to specify the function name, formal arguments, local variables, and the entry point of the function. The entry point is the label of the first of a set of SNOBOL4 statements constituting the procedure for the function.

The first argument of DEFINE is a prototype describing the form of the function call. The second argument is the entry point. For example, execution of the statement

```
          DEFINE('DELETE(STRING,CHAR)','D1')
```

defines a function DELETE having two formal arguments, STRING and CHAR, and entry point D1. The statements

```
D1        STRING CHAR    =       :S(D1)
          DELETE    =    STRING      :(RETURN)
```

form a procedure that deletes all occurrences of CHAR from the value of STRING. The statement assigning the resulting value to the variable DELETE illustrates the SNOBOL4 convention for returning a function value: The function name may be used as a variable in the function procedure. Its value on return from the procedure is the value of the function call. Return from a procedure is accomplished by transfer to the system label RETURN .

If the second argument is omitted from the call of DEFINE, the entry point to the procedure is taken to be the same as the function name. For example

```
       DEFINE('DELETE(STRING,CHAR)')
```

could have the procedure

```
DELETE STRING CHAR    =   :S(DELETE)
       DELETE   =   STRING      :(RETURN)
```

A call of the function is illustrated in the following statements

```
       MAGIC    =   'ABRACADABRA'
       OUTPUT   =   DELETE(MAGIC,'A')
```

which print BRCDBR .

Arguments are passed by value and may be arbitrarily complex expressions. Thus the statement

```
       TEXT   =   DELETE(TRIM(INPUT),' ')
```

deletes all blanks from the input string.

Functions can also fail under specified conditions. As an example, consider the following version of DELETE, which fails if STRING does not contain an occurrence of CHAR.

```
DELETE STRING   CHAR   =     :F(FRETURN)
D2     STRING   CHAR   =     :S(D2)
       DELETE   =   STRING     :(RETURN)
```

The transfer to the system label FRETURN indicates failure of the function call. Consequently,

```
       PUNCH   =   DELETE(TRIM(INPUT),'*')
```

punches a card only if the input string contains an * .

Arguments to a function and the value returned can be any type of data object. Consider, for example, the function MAXNO where MAXNO(P,N) returns a pattern that matches up to N adjacent strings matched by the pattern P. That is, if

```
       PAT   =   MAXNO('A' | 'B' | 'C' ,2)
```

then in the statement

```
       'EBCDIC'   PAT   'D'
```

the pattern match succeeds with PAT matching the string BC .

MAXNO has the defining statement

```
        DEFINE('MAXNO(P,N)')
```

and the procedure

```
MAXNO   N    =    GT(N,0) N - 1      :F(RETURN)
        MAXNO    =    NULL | P MAXNO         :(MAXNO)
```

Consider the function REVERSE that reverses a string. It has the defining statement

```
        DEFINE('REVERSE(STRING)','R1')
```

and the procedure

```
R1      ONECH    =    LEN(1) . CH
R2      STRING  ONECH   =              :F(RETURN)
        REVERSE    =    CH  REVERSE    :(R2)
```

There are two variables, ONECH and CH, used in the function definition in addition to the function name and formal argument. It is prudent to protect these variables so their use outside the function is not affected when the function is called. This is accomplished by declaring them to be local variables in the defining statement:

```
        DEFINE('REVERSE(STRING)ONECH,CH','R1')
```

When the function is called, the current values of the local variables, the formal arguments, and the function name are saved before the procedure is entered. These values are restored upon return from the procedure. This permits the programmer considerable freedom in defining functions. For example, a function can be <u>recursive</u>, i.e. include a call of the function itself. Consider the binomial coefficient c(n,m) which can be defined by equations

```
        c(n,0)   =  1
        c(n,m)   =  n*c(n-1,m-1)/m      for m > 0
```

Computational efficiency can be improved by employing the relation

```
        c(n,m)   =  c(n,n-m)
```

for m > n/2.

The corresponding programmer-defined function consists of the defining statement

```
        DEFINE('C(N,M)')
```

and the procedure

```
C       M   =   LT(N - M,M) N - M
        C   =   EQ(M,0) 1                  :S(RETURN)
        C   =   N * C(N - 1,M - 1) / M     :(RETURN)
```

COMB is an example of another recursively defined function. COMB(STR,N) lists all combinations of N characters from the string STR. The defining statement and procedure are

```
        DEFINE('COMB(STR,N,HEAD)CH')
```

and

[illegible]

```
COMB    OUTPUT    =    EQ(N,0) HEAD      :S(RETURN)
C2      STR LE(N,SIZE(STR)) LEN(1) . CH    =       :F(RETURN)
        COMB(STR,N - 1,HEAD CH)      :(C2)
```

Then

```
        COMB('ABCD',3)
```

prints

```
ABC
ABD
ACD
BCD
```

Notice that COMB is defined with three formal arguments but only two values are supplied in the initial call. The missing value is taken to be null.

I. Keywords

Several parameters and switches internal to the SNOBOL4 system can be accessed by means of keywords. Keywords are specified by prefixing an ampersand to certain identifiers. For example, if the value of the keyword &DUMP is a nonzero integer when a program terminates, a dump of natural variables is printed. Thus the execution of the statement

```
        &DUMP   =   1
```

indicates that a dump is to be produced. Other keywords are described elsewhere in this manual.

J. Arrays

Arrays of variables can be created by using the primitive function ARRAY. The arguments of ARRAY describe the number of dimensions, the bounds of each dimension, and the initial value of each variable in the array. Thus

```
        V  =  ARRAY(10,1.0)
```

creates and assigns to V a one-dimensional array of ten variables, each initialized to the real value 1.0. The created variables can be referenced by expressions of the form V<I> where I = 1,...,10. The statement

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

creates a 2-dimensional array of variables

```
        N<1,1>   N<1,2>   N<1,3>   N<1,4>   N<1,5>

        N<2,1>     .        .        .        .

        N<3,1>     .        .        .      N<3,5>
```

The omission of the second argument causes each of the variables to have the null string as initial value. The arguments in the call of ARRAY can be expressions. Thus

```
        A  =  ARRAY(TRIM(INPUT))
```

creates an array with dimensionality that is data dependent. An array reference, A<I>, that is outside the bounds of the array causes failure that can be used to control program flow. The statements

```
        I  =  1
        ST  =  ARRAY(TRIM(INPUT))
MORE    ST<I>  =  INPUT                                 :F(GO)
        I  =  I  +  1                                   :(MORE)
GO
```

generate an array, ST, and assign values to each of the variables. When all the variables in the array are assigned values, or an end of file is encountered, the transfer to GO is executed.

K. Programmer-Defined Data Types

Integers, reals, strings, patterns, and arrays are types of data objects that are built into the SNOBOL4 language. Facilities are provided in the language to permit a programmer to define additional data types. This facilitates representation of structural relationships inherent in data.

For example, a simple linear linked list is made up of nodes, each containing a value field and a link field.

```
┌─────┬────┐      ┌─────┬────┐      ┌─────┬────┐
|value|link|----->|value|link|----->|value|link|
└─────┴────┘      └─────┴────┘      └─────┴────┘
```

The primitive function DATA can be used to define the data type NODE and the two field functions, VALUE and LINK.

```
        DATA('NODE(VALUE,LINK)')
```

The statement

```
        P  =  NODE('S',)
```

creates a node with value field S and the null string in the link field. The value of P is a data object with two fields that can be referenced by means of the function calls VALUE(P) and LINK(P). The insertion of a node with value T at the head of the list is accomplished by the statement

```
        P  =  NODE('T',P)
```

The following statement deletes a node from the head of the list

```
        P  =  LINK(P)
```

L. Program_Example

This is an example of a complete SNOBOL4 program illustrating the use of comment lines, continuation lines, and the end statement. The program reads in data cards that follow the end statement.

```
*************************************************************************
*         EXAMPLE OF A FUNCTION THAT PRINTS ALL
*         PERMUTATIONS OF SIZE N FROM A GIVEN STRING.
*************************************************************************
*
          &DUMP     = 1
          DEFINE('PERM(STRING,N,HEAD)CH,USED')
*
          STRING  =  TRIM(INPUT)                           :F(ERROR)
          N  =  TRIM(INPUT)                                :F(ERROR)
          PERM(STRING,N)                                   :(END)
PERM      OUTPUT  =  EQ(N,0)   HEAD                        :S(RETURN)
PERMA     STRING  LEN(1)  .  CH  =                         :F(RETURN)
          USED
.         =  PERM(STRING USED,N - 1,HEAD CH)  USED  CH     :(PERMA)
END
ABCD
3
```

Chapter 2. Pattern Matching

A. Introduction

Strings of characters can be synthesized from smaller strings by concatenation. The converse of synthesis, decomposition of strings into substrings, is performed using pattern matching. Fundamentally, pattern matching is the process of examining a subject string for a substring which is one of a set specified by a pattern. The substring and parts thereof, formed by pattern matching, may be assigned as the values of variables, thereby naming pieces of the decomposition.

There are two types of statements in which pattern matching can occur: the pattern matching statement and the replacement statement. These statements have the respective forms

```
label   subject   pattern    goto
label   subject   pattern  =   object    goto
```

The pattern and object are expressions, as illustrated by

```
LAB1       TEXT          A | B                  :S(LAB2)F(LAB3)

LAB4       STR           C D          =    X '3'          :S(LAB5)F(LAB6)
```

Before matching actually occurs, the expression in the pattern field is evaluated. Its value may be a string, or it can be a pattern structure which may be thought of as a set of strings. The string or pattern structure is used to drive a pattern matching procedure (the scanner) which performs the actual matching. Should any string specified by the pattern field appear as a substring of the subject, pattern matching succeeds.

Two distinct tasks are performed as parts of pattern matching:

1) evaluation of expressions in the pattern field, and

2) scanning of the subject string for a substring under control of the pattern structure.

The primary purpose of this chapter is to consider in detail those SNOBOL4 language features that programmers may use to write expressions that, when evaluated, yield pattern structures. These features include the pattern building operations of concatenation and alternation, primitive pattern structures built into the system, primitive functions whose values are pattern structures, value assignment operations, and the unary operator * that produces an unevaluated expression. Pattern structures representing sets of fixed strings such as those built by

```
     BASE  =  'BINARY'  |  'DECIMAL'  |  'HEX'
     SCALE  =  'FIXED'  |  'FLOAT'
     ATTRIBUTE  =  SCALE  BASE
```

are basic to pattern matching. Additional language features provide natural ways to talk about more complicated sets of strings, such as:

All strings of ~~length~~ 5.
~~All~~ characters up to the first comma.
The longest string of blanks.
Any number of repetitions of a string.
Any string balanced with respect to parentheses.
Any string at all.

For many users of SNOBOL4, a knowledge of how patterns are actually matched is of little importance. The success or failure of matching is all that matters. However, by understanding the scanning procedure, a programmer can write more efficient patterns and make use of features such as immediate value assignment and unevaluated expressions that can actually change a pattern during matching. Thus, the secondary purpose of this chapter is to indicate how the scanner works.

B. Alternation and Concatenation

A brief introduction to the pattern building operations of alternation and concatenation appears in Chapter 1. There, alternation and concatenation are used to build pattern structures which match sets of strings.

Alternation, indicated by the binary operator | , builds a single pattern structure from its two arguments. If P1 and P2 are strings or pattern structures, the statement

```
     P3  =  P1  |  P2
```

builds a new structure and assigns it as the value of P3. P3 matches any string matched by P1 or P2.

No explicit operator is used to indicate concatenation. Concatenation is implied when two elements of an expression are separated by one or more blanks. If P4 and P5 are strings, the statement

```
     P6  =  P4  P5
```

assigns to P6 a string which is the value of P4 followed by the value of P5. If either P4 or P5 is a pattern structure, the statement above builds a pattern structure and assigns it as the value of P6. P6 matches any string which may be formed from a string matched by P4 followed by a string matched by P5.

Alternation and concatenation can be used to build pattern structures which match large numbers of strings. For instance, the following statements build a pattern structure PAT .

```
       P   =  'BE'  |  'BEA'  |  'BEAR'
       Q   =  'RO'  |  'ROO'  |  'ROOS'
       R   =  'DS'  |  'D'
       S   =  'TS'  |  'T'
       PAT  =  P  R  |  Q  S
```

Concatenation has higher precedence than alternation, so the structure for PAT is built as if

```
       PAT  =  (P  R)  |  (Q  S)
```

had been written. PAT matches any of the twelve strings:

```
       BEDS             ROTS
       BED              ROT
       BEADS            ROOTS
       BEAD             ROOT
       BEARDS           ROOSTS
       BEARD            ROOST
```

Execution of pattern matching or replacement statements involves evaluation of the pattern field (which may build a pattern structure) and the actual scanning of the subject string. Building pattern structures is a complicated process frequently requiring more time than the scanning itself. If a pattern matching or replacement statement appears in a program loop, the pattern field is evaluated for each iteration of the loop. If evaluation causes a pattern structure to be built, time and storage are often consumed needlessly. For example, the following program examines each card of an input deck for P IS TRUE or P IS FALSE , printing those cards in which either appears.

```
LOOP   CARD  =  TRIM(INPUT)                                :F(END)
       CARD  'P IS '  ('TRUE' | 'FALSE')                   :F(LOOP)
       OUTPUT  =  CARD                                     :(LOOP)
END
```

A pattern structure for 'P IS ' ('TRUE' | 'FALSE') is built for each iteration of the loop. A more efficient program is the following which builds the pattern structure in an assignment statement outside of the loop.

```
       TORF  =  'P IS '  ('TRUE'  |  'FALSE')
LOOP   CARD  =  TRIM(INPUT)                                :F(END)
       CARD  TORF                                          :F(LOOP)
       OUTPUT  =  CARD                                     :(LOOP)
END
```

C. Scanning

Matching a pattern structure against a subject string is done by a procedure called the scanner. The pattern structure behaves like a program that indicates to the scanner how to examine the subject string.

At any instant during scanning, the scanner needs two pieces of information:

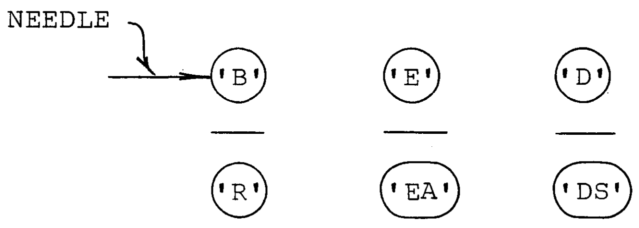
1) where in the subject string it should be looking, and
2) what component of the pattern structure it should match.

The scanner has a pointer called the cursor which is positioned to the left of the character that the scanner must match. A second pointer called the needle points at the component of the pattern structure.

Consider the following example, in which the string of characters READS is matched against a pattern structure which is the value of BR.

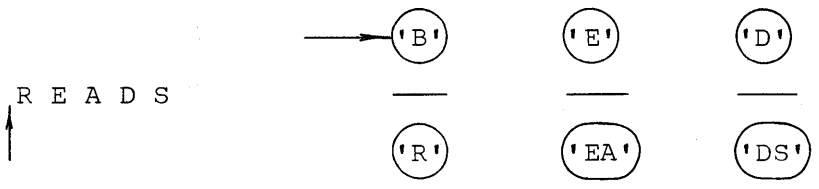
```
        BR  =  ('B' | 'R')  ('E' | 'EA')  ('D' | 'DS')
        'READS'  BR
```

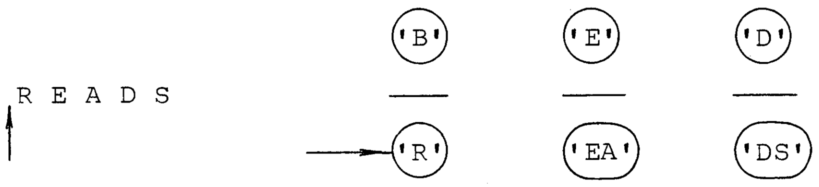
For illustrative purposes, it is convenient to think of components of a pattern structure as a set of beads which the scanner is trying to thread using the needle. A bead diagram representing BR is shown below.

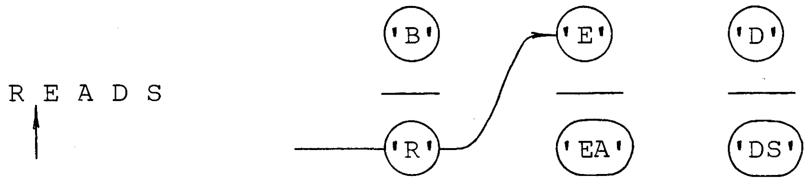


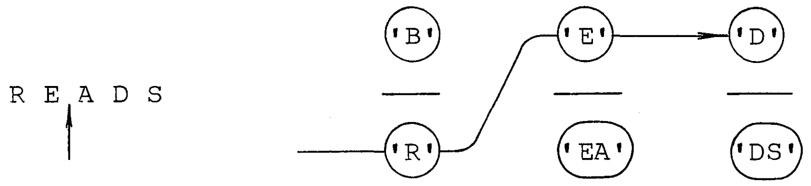
In bead diagrams, left to right order of concatenation is preserved. Alternation is represented top to bottom in the vertical direction. The needle points at the bead which the scanner is currently trying to match. If a bead matches, the needle passes through and moves upward as far as it can go without crossing a horizontal line. If a bead does not match, the needle moves down to an alternate bead provided one exists. Downward movement may not cross a horizontal line. If no alternate exists, the needle is pulled back through the last successfully matched bead and an alternative is sought there.

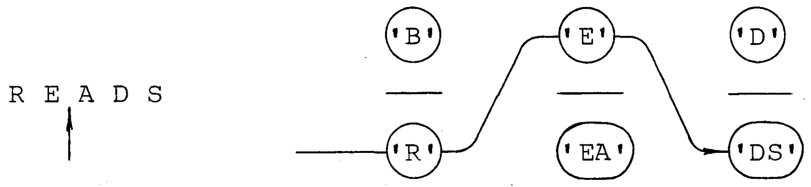
The following chart illustrates the steps in matching READS against BR. The arrow pointing at READS represents the cursor while the arrow pointing at the beads represents the needle. Failure in the fifth step causes the needle to be pulled back. The cursor is moved back at the same time.

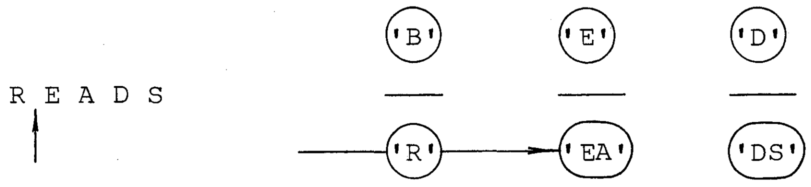

R E A D S
'B'
'E'
'D'
'R'
'EA'
'DS'

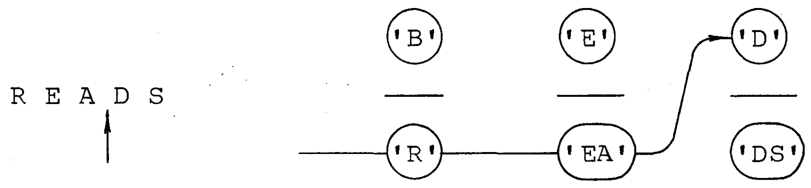

R E A D S
'B'
'E'
'D'
'R'
'EA'
'DS'

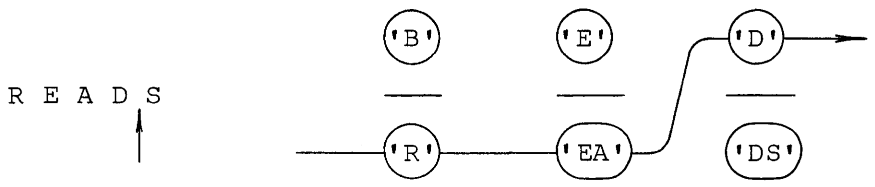

R E A D S
'B'
'E'
'D'
'R'
'EA'
'DS'


R E A D S
'B'
'E'
'D'
'R'
'EA'
'DS'


R E A D S
'B'
'E'
'D'
'R'
'EA'
'DS'


R E A D S
'B'
'E'
'D'
'R'
'EA'
'DS'


R E A D S
'B'
'E'
'D'
'R'
'EA'
'DS'

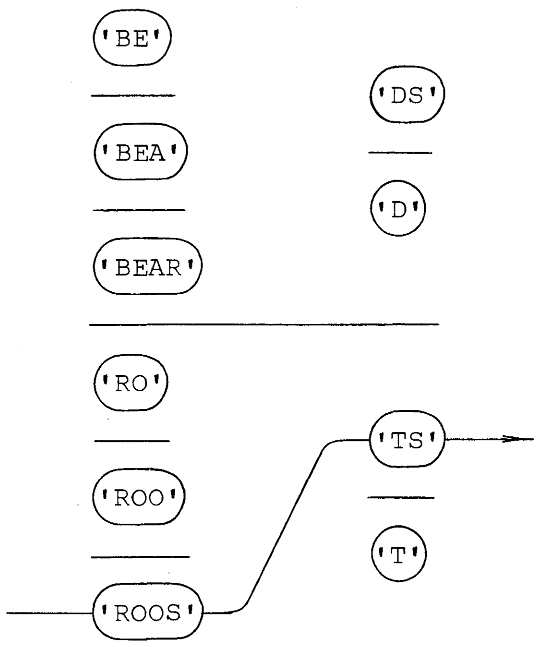


Bead diagrams graphically illustrate one important control which the programmer has over the scanner. In a pattern-valued expression such as

```
    BR  =  ('B' | 'R')  ('E' | 'EA')  ('D' | 'DS')
```

alternatives are matched by the scanner in left to right order (top to bottom in the bead chart). Thus, the scanner attempts to match 'B' before 'R', 'E' before 'EA', and 'D' before 'DS'. By positioning alternatives correctly a programmer can control the order in which the scanner looks at them.

The bead diagram for the pattern structure PAT developed in the previous section follows.



A successful match in the statement

```
    'ROOSTS'   PAT
```

requires eleven steps.

D. Modes of Scanning

Two keywords, &ANCHOR and &FULLSCAN, give the programmer additional control over the scanner. The scanner operates in an unanchored or anchored mode, depending on the value of &ANCHOR. When unanchored, a pattern can match anywhere in the subject string. When anchored, a pattern can match only beginning at the first character.

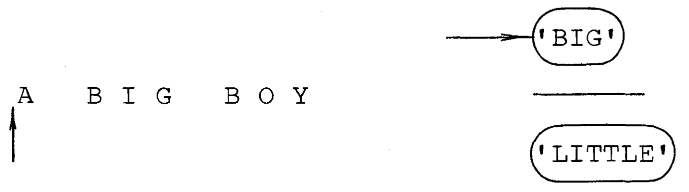
For efficiency, tests are made during scanning which prevent the scanner from looking at alternatives which cannot possibly succeed. &FULLSCAN can be used to turn these tests off, leading to complete but possibly inefficient pattern matching. Discussion of &FULLSCAN is deferred until the end of this chapter, since it is useful only with more sophisticated patterns.

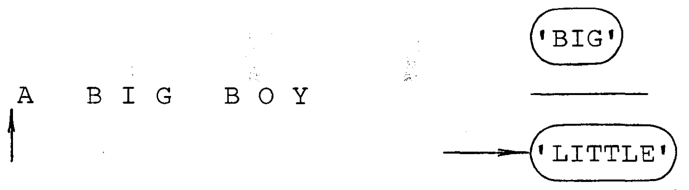
1. Unanchored Mode

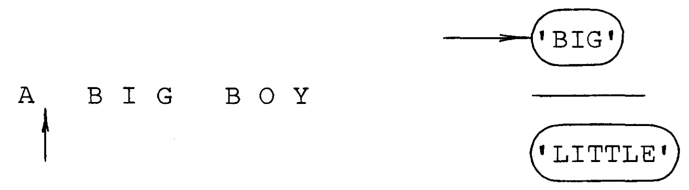
The keyword &ANCHOR initially has the value zero, signifying the unanchored mode of scanning. The scanner may look anywhere in the subject string for an appropriate substring. Consider the following example.

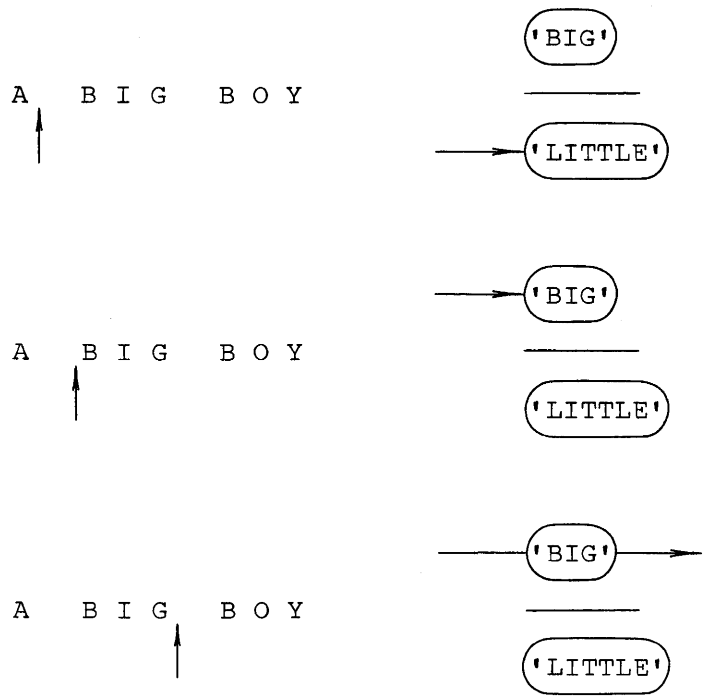
```
     'A BIG BOY'   'BIG'  |  'LITTLE'
```

Pattern matching succeeds. The steps involved are shown below using a bead diagram.









The cursor is initially at the left of the subject string. When all possible alternatives fail, the cursor is moved one character to the right. All possible alternatives are tried with the cursor beginning in the new position. Again, all alternatives fail. The cursor is moved again and this time the first alternative succeeds.

In the unanchored mode, the origin of pattern matching is moved by changing the initial position of the cursor. Thus, the scanner matches, if possible, a substring anywhere in the subject string. If more than one valid substring exists, the scanner finds the leftmost one.

2. Anchored Mode

Frequently it is necessary to know if a pattern matches with its origin at the first character of the subject string. As an example, suppose a program is desired which reads any other SNOBOL4 program and prints only those lines that are not comments (i.e. do not have * in column 1). At first glance, the following statements might seem to suffice.

```
BEGIN  LINE  =  INPUT                                      :F(END)
       LINE  '*'                                           :S(BEGIN)
       OUTPUT  =  LINE                                     :(BEGIN)
END
```

Unfortunately, the program does not work because a card with * appearing anywhere at all in it is rejected.

If &ANCHOR has a nonzero value obtained by executing an assignment statement such as

```
    &ANCHOR  =  1
```

the pattern match is anchored at the left of the subject string. Anchoring is achieved by not moving the initial position of the cursor when all alternatives in the pattern structure fail. Thus, the scanner, when anchored, only matches * against the first character of LINE.

The anchored mode of scanning is generally more efficient than the unanchored mode, since the scanner examines fewer possibilities. Anchored scanning should be used where possible. It is, of course, permissible to switch modes during execution of a program by simply changing the value of &ANCHOR.

E. Value Assignment through Pattern Matching

Pattern matching may be viewed as a means of decomposing a string into substrings. To be useful, a substring found by the scanner often must be assigned as the value of a variable. Consider the pattern BR used in an earlier section.

```
        BR  =  ('B' | 'R')  ('E' | 'EA')  ('D' | 'DS')
```

Used in a pattern matching statement such as

```
        STR  BR                                          :S(L1)F(L2)
```

where the subject string may be anything, success of matching indicates only that one of the valid strings appears somewhere in STR. It does not indicate which string matches or how it matches. On failure, no indication is given of how nearly successful the scanner was. There are two ways of assigning a substring found by the scanner to a variable: conditional value assignment and immediate value assignment.

1. Conditional Value Assignment

The binary operator . is used to indicate conditional value assignment. The expression

```
                 P  .  V
```

associates a variable V with a pattern P so that upon successful completion of pattern matching, the substring matched by P is assigned as the value of the variable V. Thus, by associating several variables with portions of a pattern, it is possible to ascertain what the overall pattern matches, and also which components of the pattern are used in the match. For example, rewriting BR as

```
        BR  =  (('B' | 'R') ('E' | 'EA') ('D' | 'DS'))  .  BRVAL
```

associates the variable BRVAL with the entire pattern. On successful completion of matching, the entire substring matched is assigned as value of BRVAL. Rewriting still further, variables can be associated with pieces of the pattern.

```
        BR   =   (('B' | 'R') . FIRST  ('E' | 'EA') . SECOND
                  ('D' | 'DS') . THIRD) . BRVAL
```

A successful match causes the entire substring to be assigned as the value of BRVAL. B or R becomes the value of FIRST, E or EA becomes the value of SECOND, and D or DS becomes the value of THIRD. Failure to match leaves the values of all variables unchanged.

2. Immediate Value Assignment

The binary operator $ signifies immediate value assignment. The expression

```
                  P  $  V
```

associates a variable V with a pattern P so that whenever P matches a substring, the substring immediately becomes the new value of V. It is possible, by using $, to associate variables with parts of a large pattern, to see how far scanning progressed in the event of failure. Value assignment is done for those parts of the pattern which match even though the overall match fails. Suppose BR is rewritten using $ instead of . where shown.

```
        BR   =   (('B' | 'R') $ FIRST  ('E' | 'EA') $ SECOND
+                 ('D' | 'DS') $ THIRD) . BRVAL
```

In the following statement, pattern matching fails.

```
        'BEATS'  BR                                     :S(L1)F(L2)
```

However, since immediate assignment is performed whenever the associated part of the pattern matches, the following assignments are made.

```
        FIRST   =  'B'
        SECOND  =  'E'
        SECOND  =  'EA'
```

Values of THIRD and BRVAL are unchanged. If conditional assignment is used, values of all four variables are unchanged. In the following example, the pattern matches.

```
        'BREADS'  BR                                    :S(L1)F(L2)
```

Values assigned both during and after scanning are:

```
        FIRST  =  'B'
        FIRST  =  'R'
        SECOND  =  'E'
        SECOND  =  'EA'
        THIRD  =  'D'
        BRVAL  =  'READ'
```

The outcome is the same as if conditional value assignment had been used. Immediate value assignment is less efficient in this case because two redundant assignments are made. As a general rule, conditional value assignment should be used whenever possible. Immediate value assignment should be used only in those cases where intermediate results are important.

Examples using both immediate and conditional value assignment appear throughout the remainder of this manual.

3. Special Considerations

Precedence

The operators . and $ have the highest precedence of all operators and associate to the left. Thus, in the statement

```
       BR   =  (('B' | 'R') $ FIRST  ('E' | 'EA') $ SECOND
+                ('D' | 'DS') $ THIRD) . BRVAL
```

the outer parentheses are required to associate BRVAL with the entire pattern, while additional parentheses are not required to associate FIRST, SECOND, and THIRD.

Association with the Variable OUTPUT

Since OUTPUT is a variable, it may be associated with any portion of a pattern. A successful match involving the pattern

```
       ('BED' | 'BUG' | 'BOMB')  .  OUTPUT
```

causes the successful alternative to be printed. Using $ to associate OUTPUT with several parts of a pattern achieves the effect of tracing the progress of the scanner. By constructing BR as

```
       BR   =  ('B' | 'R') $ OUTPUT  ('E' | 'EA') $ OUTPUT
+                ('D' | 'DS') $ OUTPUT
```

the output resulting from execution of the statement

```
       'READS'  BR                                          :S(L1)F(L2)
```

is

```
R
E
EA
D
```

Value Assignment in Replacement Statements

Value assignment is a necessity in some kinds of replacement statements. In the following replacement statement E or EA is replaced with I only if the overall pattern BR matches. In effect, the replacement statement changes BED and BEAD into BID, BEDS and BEADS into BIDS, etc., if these strings appear in STR.

```
        BR  =  ('B' | 'R') . FIRST  ('E' | 'EA')  ('D' | 'DS') . LAST
        STR  BR  =  FIRST  'I'  LAST
```

The replacement statement works properly because conditional assignment is done after pattern matching, but before the object expression is evaluated.

Association of Several Variables with One Pattern

Earlier examples illustrated how variable association may be nested. An example is

```
        PAT  =  (P1 . V1  P2 . V2) . V3
```

It is also possible to associate more than one variable with a single pattern structure. The statement

```
        PAT  =  P1 $ V1 . V2
```

builds a pattern structure where variables V1 and V2 are both associated with the pattern P1, V1 as immediate assignment and V2 as conditional assignment. Changing the order of association to

```
        PAT  =  P1 . V2 $ V1
```

has no effect on the value assignment. If PAT is involved in a successful pattern match, V1 and V2 are assigned the same value. If the pattern match fails, the value of V1 might be changed but the value of V2 is not.

F. The Null String in Pattern Matching

The null string is a string of zero length. Attempts by the scanner to match the null string always succeed. The variable NULL has the null string as its initial value and, by convention, is used as the null pattern which matches a string of zero length. Pattern matching in the statement

```
        STR  NULL                                     :S(ON)F(ERROR)
```

always succeeds even if STR itself has the null string as value.

The variable NULL is frequently used in more complex patterns. For example, a pattern which matches the eight strings

```
        C          BC
        D          BD
        AC         ABC
        AD         ABD
```

can be written as

```
        (NULL | 'A') (NULL | 'B') ('C' | 'D')
```

Matching a pattern of the form

```
        NULL $ X $ Y  PAT
```

sets the values of X and Y to the null string before matching of PAT begins.

A number of patterns described in this chapter match the null string. Where bead diagram representations of the patterns are given, NULL is used to indicate the null string.

G. LEN

LEN(integer) is a primitive function whose value is a pattern structure that matches any string of the specified length. The argument of LEN must have nonnegative integer value when pattern matching is performed. In the following example, pattern matching succeeds only if the subject STR has in it somewhere an open parenthesis separated from a closed parenthesis by exactly five characters.

```
      STR  '('  LEN(5)  ')'                          :S(L1)F(L2)
```

LEN can be used to break out fixed-length fields from strings. In the following example dates from data cards such as

```
1290 SEP. 27 CHINA, CHIHLI           100,000
1293 MAY  20 JAPAN, KAMARKURA         30,000
1531 JAN. 26 PORTUGAL, LISBON         30,000
```

are reformatted as

```
SEP. 27, 1290   CHINA, CHIHLI              100,000
MAY  20, 1293   JAPAN, KAMARKURA            30,000
JAN. 26, 1531   PORTUGAL, LISBON            30,000
```

```
       &ANCHOR  =  1
       DATE  =  LEN(4) . YR  ' '  LEN(4) . MO  ' '  LEN(2) . DAY
LOOP   CARD  =  INPUT                                 :F(END)
       CARD  DATE  =  MO  ' '  DAY  ', '  YR  ' '     :F(NOGOOD)
       OUTPUT  =  CARD                                :(LOOP)
NOGOOD OUTPUT  =  CARD  '  IMPROPERLY FORMATTED.'
END
```

LEN is used to match the various pieces of the data assigning the strings found to the variables YR, MO, and DAY. YR, MO, and DAY are assigned values after pattern matching but before the entire substring matched by DATE is replaced. Only the date portion of CARD is reformatted.

H. SPAN and BREAK

SPAN and BREAK are primitive functions whose values are pattern structures that match runs of characters. Patterns described by

a run of blanks,
a string of digits, and
a word (run of alphabetic characters)

can be formed using SPAN as

```
SPAN(' ')
SPAN('0123456789')
SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Patterns described by

everything up to the next blank,
everything up to the next punctuation mark, and
everything up to the next number,

can be formed using BREAK as

```
BREAK(' ')
BREAK(',.;:!?')
BREAK('+-0123456789')
```

Arguments of BREAK and SPAN must be nonnull strings when pattern matching is performed.

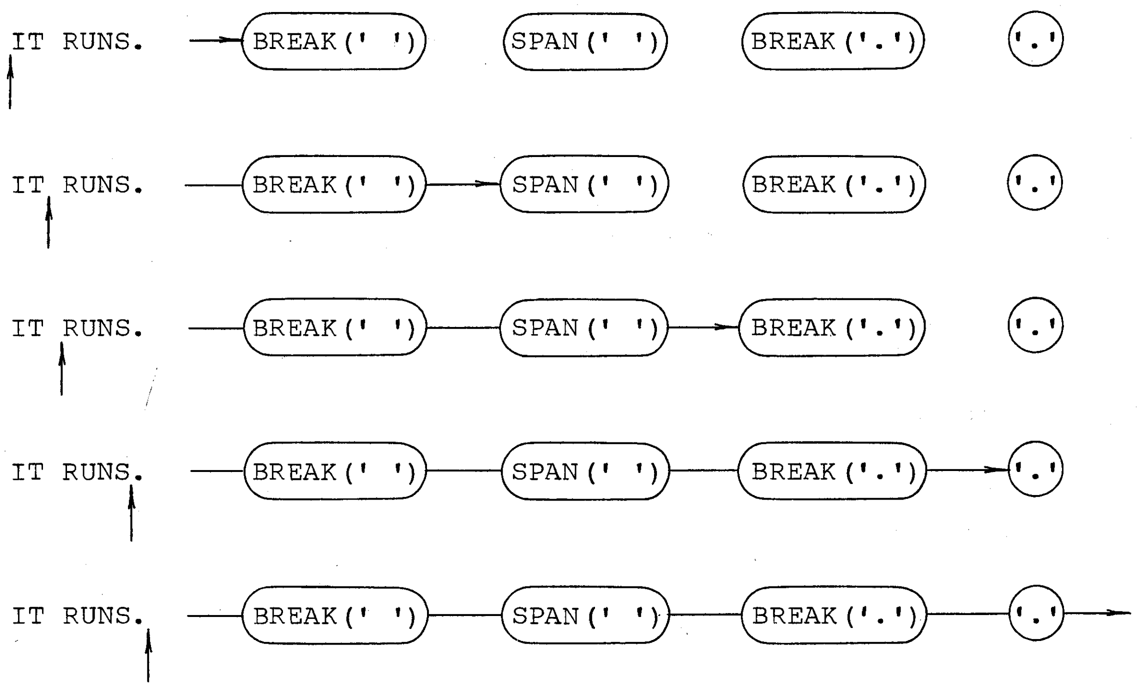
The pattern structure for SPAN matches the longest string beginning at the cursor which consists solely of characters which appear in the argument. SPAN may be thought of as streaming from the cursor until a character not included in the argument is found. SPAN must match at least one character.

BREAK generates a pattern structure that matches the longest string beginning at the cursor which does not contain a character of the argument. Thus, regarding its argument as a list of "break" characters, BREAK streams from the cursor up to but not including the first break character. <u>BREAK must find a break character.</u> If the cursor is positioned immediately to the left of a break character, BREAK matches the null string. BREAK fails if no break character is found.

A bead diagram for the statement

```
    'IT RUNS.'      BREAK(' ')  SPAN(' ')  BREAK('.')  '.'
```

illustrates how the cursor is moved by SPAN and BREAK.



The next program illustrates the use of both BREAK and SPAN. It compresses tabulated data, leaving fields separated by single colons rather than an arbitrary number of blanks. For example, if the input is

```
ACTINIUM     AC    89    227*       1899    DEBIERNE
ALUMINUM     AL    13     26.9815   1825    OERSTED
AMERICIUM    AM    95    243*       1944    SEABORG
ANTIMONY     SB    51    121.75     1450    VALENTINE
```

the output is

```
ACTINIUM:AC:89:227*:1899:DEBIERNE
ALUMINUM:AL:13:26.9815:1825:OERSTED
AMERICIUM:AM:95:243*:1944:SEABORG
ANTIMONY:SB:51:121.75:1450:VALENTINE
```

```
        &ANCHOR  =  1
        FIELD  =  BREAK(' ')  . CHARS   SPAN(' ')
LOOP    CARD  =  TRIM(INPUT)                                :F(END)
INLOOP CARD  FIELD  =  CHARS  ':'                           :S(INLOOP)
        PUNCH  =  CARD                                      :(LOOP)
END
```

Each input card is repeatedly examined for a run of blanks, and the blanks are replaced by a colon. When blanks no longer exist the compression is complete and a new card is punched.

Some care must be exercised in using BREAK, since it does not match the break character which stops the streaming. Suppose a program is wanted which restores, to some degree, the compressed data generated above. Each field of the compressed data can be broken out using a statement such as

```
        CARD  BREAK(':') . FLD  ':'  =
```

Since BREAK(':') does not "consume" the colon, the literal is included to remove the break character.

SPAN never matches a string shorter than the maximum span. For example,

```
        '9824761.'  SPAN('0123456789')  '6'
```

cannot succeed since SPAN always matches up to the decimal point.

In the event that components of the pattern beyond BREAK fail, BREAK does not skip over the break character and continue streaming. In the anchored mode the following statement never succeeds.

```
        '123,427,642.00'  BREAK('.,')  LEN(1)  '0'
```

BREAK('.,') matches 123 and that is all.

I. ANY and NOTANY

ANY(string) and NOTANY(string) are primitive functions whose values are pattern structures that match single characters. ANY matches any character appearing in its argument. NOTANY matches any character not appearing in its argument. Thus, the pattern structure for ANY('AEIOU') matches any vowel. The pattern for NOTANY('AEIOU') matches any character that is not a vowel. Arguments of ANY and NOTANY must be nonnull strings when pattern matching is performed.

ANY and NOTANY are fast ways of looking for one of a set of single characters. For example,

```
        ANY('AEIOU')
```

is preferable to

```
        'A' | 'E' | 'I' | 'O' | 'U'
```

The call

```
        NOTANY('STRUCTURE')
```

is valid even though the characters T and U appear twice.

Two examples utilizing ANY and NOTANY follow. The first counts the number of occurrences of vowels and consonants in an input deck of English text. The second counts and publishes the number of times individual letters appear in input text. In both cases, nonalphabetic characters are ignored.

```
        &ANCHOR  =  0
        VOWEL  =  'AEIOU'
        CONS  =  'BCDFGHJKLMNPQRSTVWXYZ'
        CHAR  =  ANY(VOWEL) . V  NULL . C  |
.                ANY(CONS) . C  NULL . V  |
.                LEN(1)  NULL . V . C
INPUT   OUTPUT  =  TRIM(INPUT)                              :F(LOOP)
        TEXT  =  TEXT  OUTPUT                               :(INPUT)
LOOP    TEXT  CHAR  =                                       :F(PUB)
        VCOUNT  =  VCOUNT  +  SIZE(V)
        CCOUNT  =  CCOUNT  +  SIZE(C)                       :(LOOP)
PUB     OUTPUT  =
        OUTPUT  =  'VOWELS OCCUR '  VCOUNT  '  TIMES.'
        OUTPUT  =  'CONSONANTS OCCUR '  CCOUNT  '  TIMES.'
END
```

The pattern CHAR matches one character. If that character is a vowel, it is assigned as the value of V, and the value of C becomes the null string. If CHAR matches a consonant, it becomes the value of C, and V becomes null. If the character is nonalphabetic, both C and V become null.

Inside the main loop, characters are removed from TEXT one at a time by CHAR. The two statements incrementing VCOUNT and CCOUNT are executed for every character. Because the conditional value assignment sets the values of V and C appropriately, only VCOUNT or CCOUNT or possibly neither is actually incremented by one.

Output from a typical run is:

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR PARTY.

         VOWELS OCCUR  32  TIMES.
         CONSONANTS OCCUR  54  TIMES.
```

The program to count occurrences of individual letters is

```
        &ANCHOR  =  1
        ALPH  =  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
        LETTER  =  LEN(1) . LET
        CHAR  =  NOTANY(ALPH) . SW  |  LETTER  NULL . SW
INPUT   OUTPUT  =  TRIM(INPUT)                          :F(LOOP)
        TEXT  =  TEXT  OUTPUT                           :(INPUT)
LOOP    TEXT  CHAR  =                                   :F(PUB)
        $LET  =  IDENT(SW)  $LET + 1                    :(LOOP)
PUB     OUTPUT  =
PUBL    ALPH  LETTER  =                                 :F(END)
        OUTPUT  =  '        '  LET  '  APPEARS  '
.                  $LET  '  TIMES.'                     :(PUBL)
END
```

The pattern CHAR matches exactly one character. If the character is nonalphabetic, the character becomes the value of SW. If the character is alphabetic, it becomes the value of LET and SW becomes null.

In the main loop, characters are removed from TEXT one at a time by CHAR and the values of SW and LET are assigned. The count for each character is kept in a variable having the name of the letter. (That is, the variable A contains the count for A.) The statement

```
        $LET  =  IDENT(SW)  $LET + 1
```

increments the count for the character found provided the value of SW is null, which is true only for the alphabetic characters.

Output from a typical run is:

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR PARTY.

         A  APPEARS  4  TIMES.
         B  APPEARS  1  TIMES.
         C  APPEARS  2  TIMES.
         D  APPEARS  3  TIMES.
         E  APPEARS  9  TIMES.
         F  APPEARS  3  TIMES.
         G  APPEARS  2  TIMES.
         H  APPEARS  5  TIMES.
         I  APPEARS  5  TIMES.
         J  APPEARS  1  TIMES.
         K  APPEARS  1  TIMES.
         L  APPEARS  3  TIMES.
         M  APPEARS  4  TIMES.
         N  APPEARS  3  TIMES.
         O  APPEARS  12  TIMES.
         P  APPEARS  2  TIMES.
         Q  APPEARS  1  TIMES.
         R  APPEARS  5  TIMES.
         S  APPEARS  2  TIMES.
         T  APPEARS  9  TIMES.
         U  APPEARS  2  TIMES.
         V  APPEARS  1  TIMES.
         W  APPEARS  2  TIMES.
         X  APPEARS  1  TIMES.
         Y  APPEARS  2  TIMES.
         Z  APPEARS  1  TIMES.
```

J. TAB, RTAB, and REM

TAB(integer) and RTAB(integer) are primitive functions whose values are pattern structures that match all characters from the ~~current cursor~~ position ~~up~~ to a specific point in the subject string. TAB(N) matches up through the Nth character of the subject string. RTAB(N) matches up to but not including the Nth character from the right end of the subject string. Stated another way, TAB(N) insures that N characters are matched by positioning the cursor to the right of the Nth character. RTAB(N) insures that all but N characters are matched by positioning the cursor to the left of the Nth character from the end. For example, in the statement

```
        'SNOBOL4'  LEN(2)  TAB(6)
```

the pattern matches the substring SNOBOL with TAB(6) matching OBOL . In a similar statement,

```
        'SNOBOL4'  LEN(2)  RTAB(1)
```

the substring SNOBOL is once again matched with RTAB(1) matching OBOL .

RTAB(0) is particularly useful for matching everything to the end of the subject string. For convenience, the variable REM has as its initial value the pattern structure for RTAB(0). Thus, the pattern

```
        LAST8  =  RTAB(8)  REM . L8
```

matches the entire subject and assigns the last eight characters as the value of L8.

TAB and RTAB require integer arguments when pattern matching is performed. If the argument of TAB or RTAB is negative, error termination occurs. An argument that would require moving the cursor left causes failure. The statement

```
        STR  LEN(5)  TAB(4)
```

fails because the cursor cannot be moved back by TAB(4).

TAB and RTAB are particularly valuable in breaking fields out of structured data. The following data is part of the 1964 list of congressmen from New Jersey.

```
   Column 4            Column 30     Column 36
   ↓                           ↓     ↓
 1 WILLIAM T. CAHILL           REP   COLLINGSWOOD
 2 THOMAS C. MCGRATH, JR.      DEM   MARGATE CITY
 3 JAMES J. HOWARD             DEM   WALL
        .
        .
        .
14 DOMINICK V. DANIELS         DEM   JERSEY CITY
15 EDWARD J. PATTEN            DEM   PERTH AMBOY
```

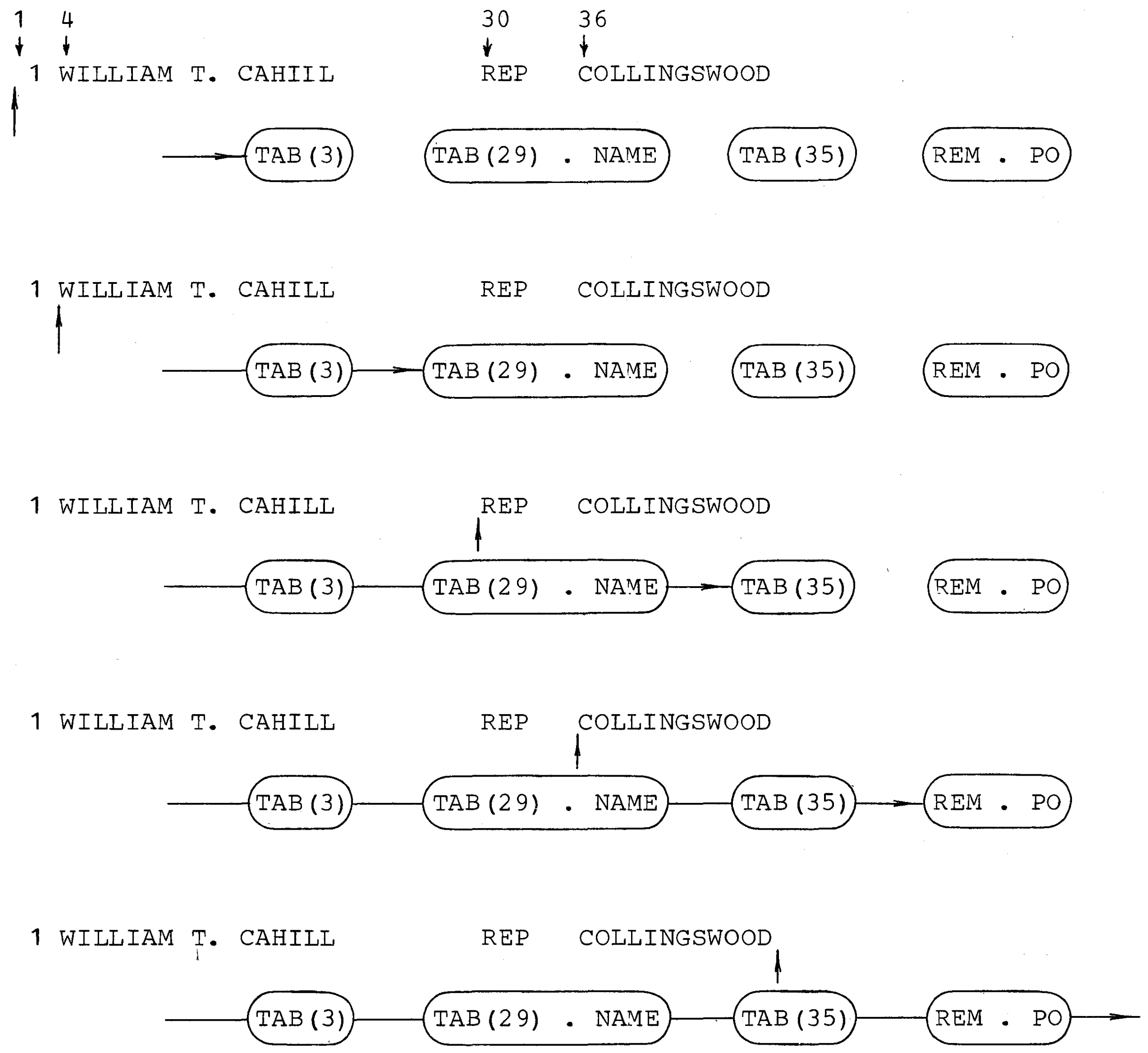
Suppose a new deck of cards is desired, listing only the names left justified at column 1, and the post office address right justified at column 44. The following program reads the cards, breaks out the NAME and PO fields, formats and punches a new deck.

```
        &ANCHOR  =  1
        BLANKS  =  '                                            '
        NAMEANDPO  =  TAB(3)  TAB(29) . NAME  TAB(35)  REM . PO
LOOP    CARD  =  TRIM(INPUT)                                :F(END)
        CARD  NAMEANDPO                                     :F(ERROR)
        NAME  =  TRIM(NAME)
        BLANKS  LEN(44 - (SIZE(NAME) + SIZE(PO))) . PAD     :F(ERROR)
        OUTPUT  =  NAME  PAD  PO
        PUNCH  =  OUTPUT                                    :(LOOP)
END
```

Fields are broken out of the input cards using the pattern NAMEANDPO. The NAME field has trailing blanks which are trimmed before the output line is formatted. The post office address is obtained using REM and does not have trailing blanks since the input card was initially trimmed. LEN is used to determine the number of padding blanks required between NAME and PO to properly format the output. Output from the program is

```
WILLIAM T. CAHILL              COLLINGSWOOD
THOMAS C. MCGRATH, JR.         MARGATE CITY
JAMES J. HOWARD                        WALL
        .
        .
        .
DOMINICK V. DANIELS             JERSEY CITY
EDWARD J. PATTEN                PERTH AMBOY
```

A bead diagram illustrating the match of NAMEANDPO and the first data card is shown below.



K. POS and RPOS

POS(integer) and RPOS(integer) are primitive functions whose values are pattern structures. These pattern structures match the null string if the cursor is at a point in the subject string specified by the integer argument. POS(N) succeeds, matching the null string, only if the cursor is positioned at the right of the Nth character. RPOS(N) succeeds, matching the null string, only if the cursor is positioned to the left of the Nth character from the end of the subject string. POS and RPOS never cause the cursor to be moved; they test its position. For example, in the statements

```
        &ANCHOR  =  1
        STR  SPAN(' ')  POS(7)
```

pattern matching succeeds only if the first seven characters are blanks and the eighth is not a blank. In the following example,

```
        &ANCHOR  =  1
        STR  SPAN(' ')  RPOS(7)
```

pattern matching succeeds only if the seventh character from the end of STR is nonblank and everything preceeding it is blank.

POS(0) is a pattern that succeeds only if the cursor is at the left of the subject string. RPOS(0) succeeds only if the cursor is at the right of the subject string. POS(0) and RPOS(0) can serve as left and right anchors for any pattern P, as in

```
        ENTIRE  =  POS(0)  P  RPOS(0)
```

In the statement

```
        STR  ENTIRE
```

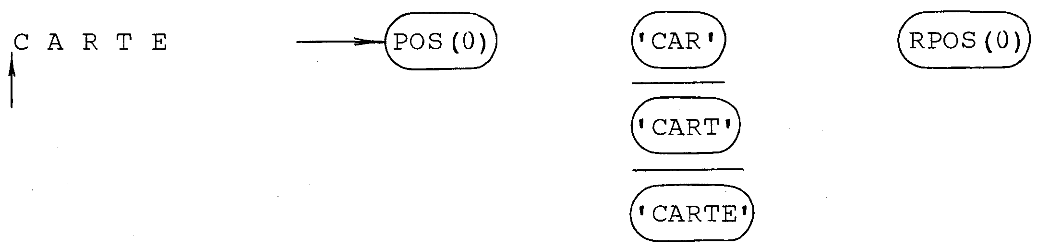
pattern matching succeeds only if P can match all of STR. If at the time ENTIRE is built, P has the value

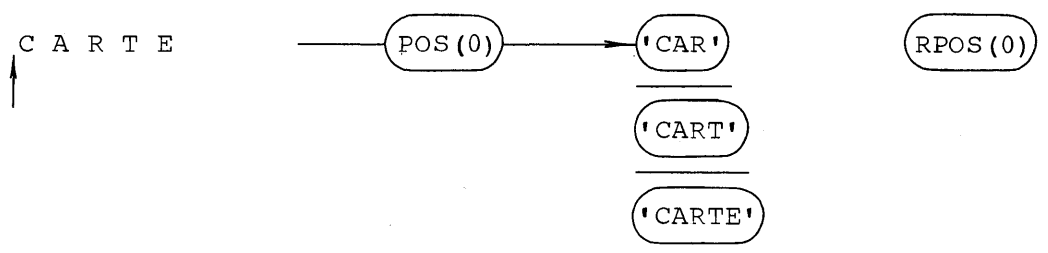
```
        'CAR'  |  'CART'  |  'CARTE'
```

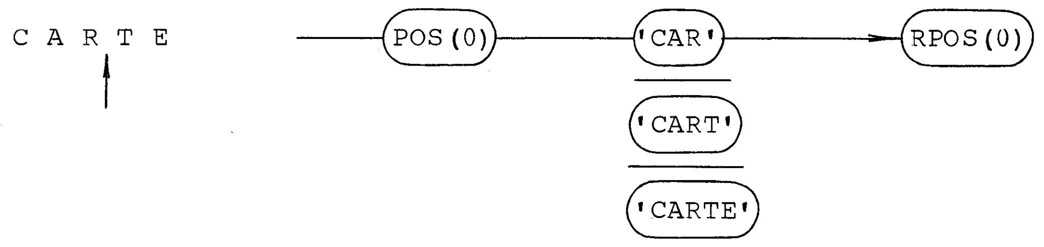
Matching in the statement

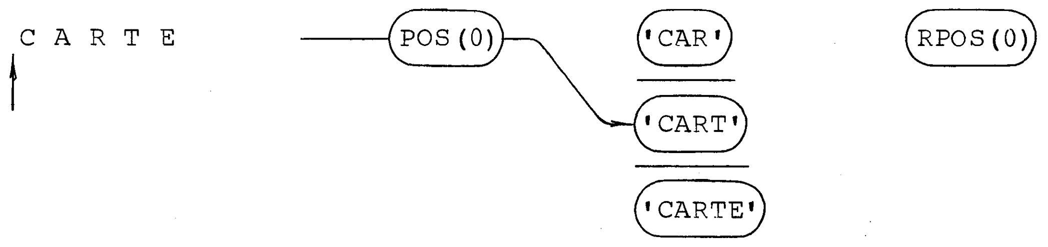
```
        'CARTE'  ENTIRE
```

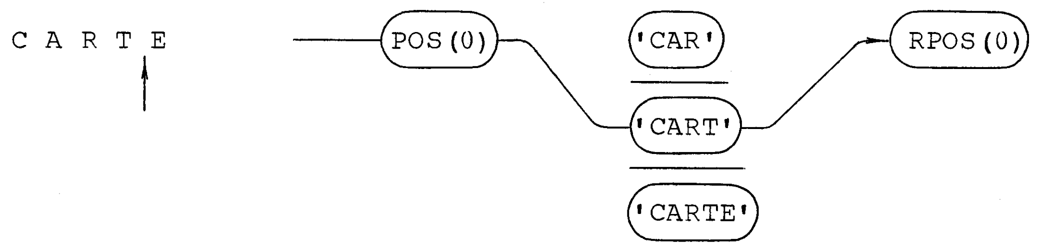
is illustrated by the bead diagram:

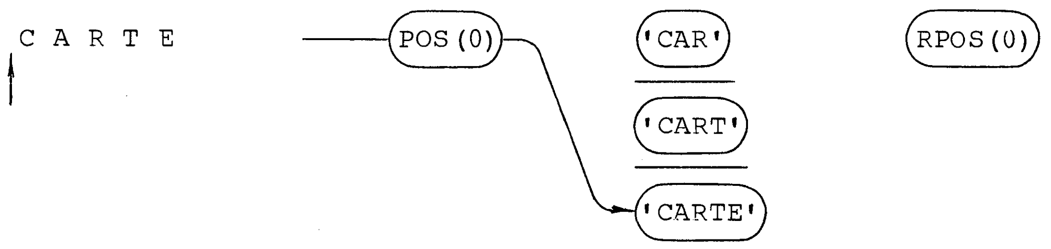


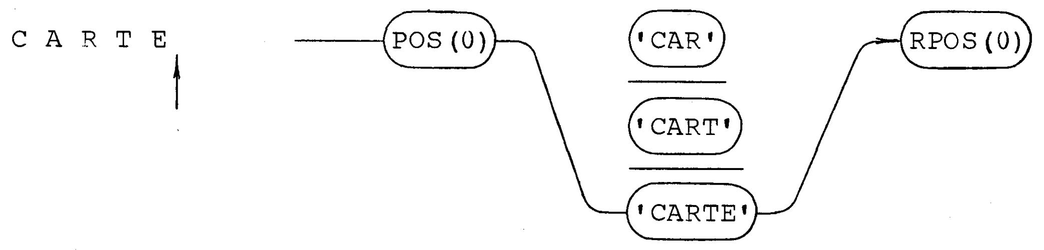

C A R T E
POS(0)
'CAR'
'CART'
'CARTE'
RPOS(0)

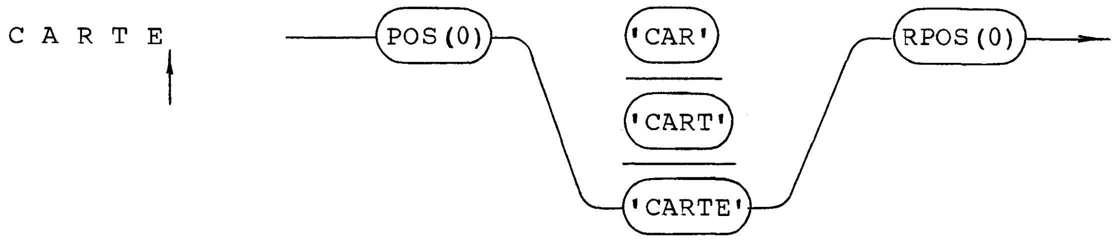

C A R T E
POS(0)
'CAR'
'CART'
'CARTE'
RPOS(0)


C A R T E
POS(0)
'CAR'
'CART'
'CARTE'
RPOS(0)


C A R T E
POS(0)
'CAR'
'CART'
'CARTE'
RPOS(0)


C A R T E
POS(0)
'CAR'
'CART'
'CARTE'
RPOS(0)


C A R T E
POS(0)
'CAR'
'CART'
'CARTE'
RPOS(0)



Arguments for POS and RPOS must have nonnegative integer values when pattern matching is performed. Negative or noninteger arguments cause error termination.

The following program uses POS, RPOS, SPAN, and BREAK to list cards which do not conform to a specific format. Cards, when properly punched, have three fields left justified at columns 1, 10 and 20. A field consists of a run of nonblank characters followed by a run of blanks. Cards not conforming are printed by the program.

```
        OUTPUT  =  'CARDS WITH IMPROPER FORMAT ARE:'
        FIELD  =  BREAK(' ')  SPAN(' ')
        FIELDS  =  POS(0) FIELD POS(9) FIELD POS(19) FIELD RPOS(0)
LOOP    CARD  =  INPUT                                  :F(END)
        CARD  FIELDS                                    :S(LOOP)
        OUTPUT  =  CARD                                 :(LOOP)
END
```

A pattern FIELD is defined as a run of zero or more nonblank characters followed by a run of blanks. FIELDS is defined using FIELD three times with POS and RPOS, which check that the fields matched are positioned properly. If the following data are provided as input

```
EXPR     PROC      ,
         SAVLNK
         RCALL     XPTR,EXPRS
       BRANCH EXRTN1
       BRANCH EXRTN2
EXRTN3   RSTURN    3
EXRTN1   RSTURN    1
EXRTN2   RSTURN  2
```

the output is

```
CARDS WITH IMPROPER FORMAT ARE:
         SAVLNK
       BRANCH EXRTN1
       BRANCH EXRTN2
EXRTN2   RSTURN  2
```

L. FAIL

FAIL is a variable whose initial value is a pattern structure that always fails. FAIL does not terminate pattern matching, but causes the scanner to seek alternatives.

Consider the following statements.

```
        &ANCHOR  =   0
        'MISSISSIPPI'  ('IS' | 'SI' | 'IP' | 'PI') $ OUTPUT  FAIL
```

Normally, the pattern would match the first IS , print it, and terminate successfully. However, FAIL causes the scanner to back up after printing the IS to look for another alternative. SI is found and printed, and again FAIL causes the scanner to back up. Thus, FAIL causes the scanner to find and print all six substrings of MISSISSIPPI that the pattern

```
        ('IS' | 'SI' | 'IP' | 'PI')
```

matches before terminating in failure.

In general, the behavior of the scanner during any pattern match may be observed using a statement of the form

```
        STR  PAT $ OUTPUT  FAIL
```

FAIL is generally used when a programmer wishes to force the scanner to try a number of alternatives even though some may succeed. In the following example words or phrases are read from cards. Cards are printed if they

1) begin with the characters SIDE ,
2) contain either a hyphen or a blank, and
3) have length less than or equal to eleven.

For example, SIDE DISH and SIDE-KICK are acceptable while SIDEBOARD and SIDE-WHEELER are not.

```
        &ANCHOR  =  1
        OUTPUT  =  'ACCEPTABLE WORDS ARE:'
        PAT  =  NULL $ P1 $ P2 $ P3
.               ('SIDE' $ P1  |  BREAK('- ') $ P2  |  LEN(12) $ P3)
.               FAIL
LOOP    CARD  =  TRIM(INPUT)                                :F(END)
        CARD  PAT
        DIFFER(P1)  DIFFER(P2)  IDENT(P3)                   :F(LOOP)
        OUTPUT  =  CARD                                     :(LOOP)
END
```

PAT is a complicated pattern that, because FAIL forces the scanner to back up, checks all three conditions. The initial portion of PAT,

```
        NULL $ P1 $ P2 $ P3
```

matches the null string, thereby immediately assigning the null value to variables P1, P2, and P3. If SIDE matches, P1 gets a nonnull value. If BREAK('- ') also matches, P2 also gets a nonnull value. Finally, if LEN(12) fails, as it should, P3 keeps its null value. The values of P1, P2, and P3 are checked in the statement following pattern matching.

M. FENCE

The variable FENCE has a pattern structure as its initial value. FENCE matches the null string when first encountered by the scanner moving left to right through a pattern. If a subsequent failure causes the scanner to back up to FENCE seeking an alternative, the pattern match is terminated. Considering FENCE as a bead, the needle passes freely from left to right. Attempting to pull the needle back through FENCE causes failure of pattern matching.

Consider the following statements:

```
    &ANCHOR  =   1
    'BERATES'  ('BE' | 'GE' | 'FRE')  ('TS' | 'T')
```

BE matches, and both TS and T fail. At this point the scanner backs up and tries GE and FRE, both of which fail. Looking at the pattern, it is obvious that GE and FRE should not be tried because the first two characters are known to be BE.

Inserting FENCE between the groups of alternatives eliminates the problem.

```
    'BERATES'  ('BE' | 'GE' | 'FRE')  FENCE  ('TS' | 'T')
```

Now, if BE matches, FENCE keeps the scanner from needlessly backing up to look at GE and FRE.

FENCE can be used to temporarily anchor the scanner in a program which otherwise operates in the unanchored mode. Inserting FENCE before PAT in the statement

```
        STR  (FENCE PAT)
```

causes pattern matching to fail if PAT does not match beginning with the first character of STR .

N. ABORT

ABORT is a variable whose initial value is a pattern structure that causes immediate termination of the entire pattern match. No alternatives are tried, and the statement fails.

ABORT is useful in constructing conditional pattern matching statements. For instance, in processing SNOBOL4 source decks as data, the following pattern ignores comment cards, but matches all others against the pattern CARD.

```
        CARDFORM  =  '*'  ABORT  |  CARD
```

Similarly, the pattern

```
        SHORTPAT  =  LEN(12)  ABORT  |  PAT
```

succeeds only if the subject string is less than 12 characters long.

In general, a pattern described by a statement of the form, "has characteristics of P but not Q," can be implemented by

```
        PNOTQ  =  Q  ABORT  |  P
```

O. Patterns with Implicit Alternatives

When failure to match a pattern component starts the scanner pulling the needle back, the scanner seeks alternatives to components that matched. So far, the only way described for creating alternatives uses the binary operator | . Components, if "backed into," either terminate the pattern match (FENCE), pass the needle to an alternative (as indicated by |), or, if no alternative exists, pass the needle still farther back to seek alternatives. Four primitive pattern structures, ARB, BAL, ARBNO(P), and SUCCEED behave differently. These patterns have implicit alternatives. Rather than pass the needle back or to an alternative, they attempt to find another suitable substring. Only when all implicit alternatives fail is the needle passed to an explicit alternative or passed back.

1. ARB

ARB is a variable whose initial value is a pattern structure that matches zero or more characters. When first encountered by the scanner moving from left to right, ARB matches the null string. When 'backed into' on subsequent occasions, ARB increases the size of the substring it matches by one. ARB fails only when it can no longer increase the length of the substring it matches.

ARB is used in the construction of patterns typified by the statement, "any string containing both CAT and DOG." Nothing is said about the order in which they appear or their separation. A suitable pattern is

```
        CATANDDOG  =  'CAT'  ARB  'DOG'  |  'DOG'  ARB  'CAT'
```

Matching CATANDDOG against the strings

```
        CATALOG FOR SEADOGS
        DOGS HATE POLECATS
        CATDOG
```

ARB matches the substrings

```
        ALOG FOR SEA
        S HATE POLE
```

and the null string, respectively.

ARB, although natural, cannot be used with impunity. For example, it should not be used as the first component of a pattern unless associated with a variable for value assignment. The statement

```
        STR  ARB  PAT
```

should be replaced by

```
        STR  PAT
```

which, when executed in the unanchored mode, behaves in exactly the same way, but is much faster.

ARB should not be used to break fields out of a string if they are separated by known delimiters. For example, the statement

```
        STR  BREAK(',') . FIELD  ','  =
```

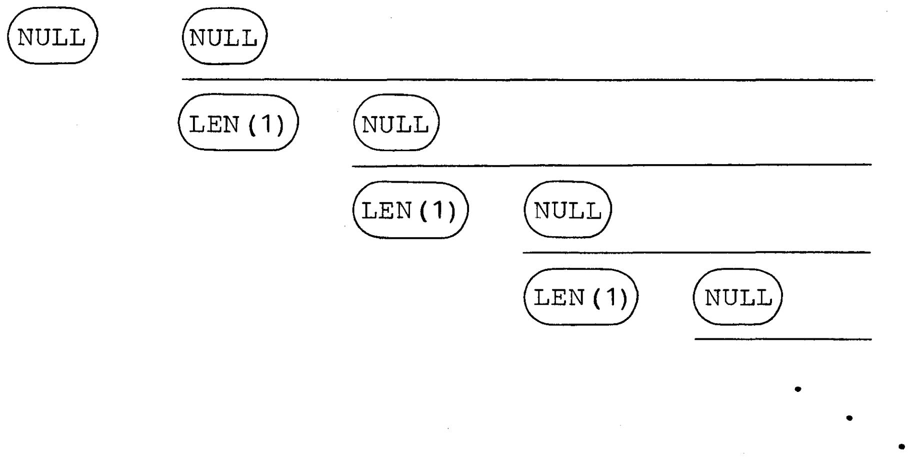
is much faster than the statement

```
        STR  ARB . FIELD  ','  =
```

although they accomplish the same thing.

The following bead diagram gives a representation of ARB. It can be seen from the diagram that

1) the null string is matched on the first attempt,
2) subsequent attempts increase the substring matched by one character, and
3) failure occurs when the size of the substring cannot be increased.



2. BAL

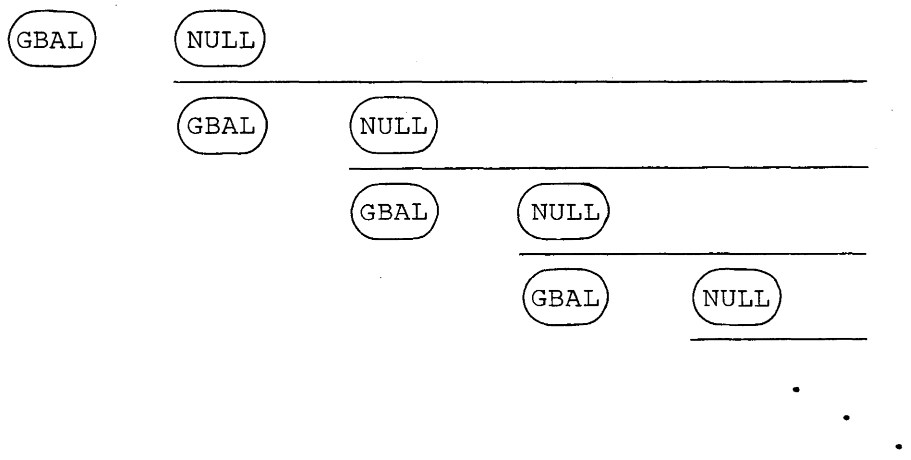
The initial value of the variable BAL is a pattern structure which matches any nonnull string of characters balanced with respect to parentheses. BAL matches

```
X
XYZ
(A+B)
A(B*C)(E/F)G+H
```

BAL does not match

```
)A+B(
((A+B)
```

A bead diagram for BAL resembles the one for ARB except that the null string is not acceptable.



GBAL is a routine that

1) fails if no characters remain in the subject string,

2) fails if the first character examined is) ,

3) matches any character except) or (,

4) matches all characters from (up to and including the balancing) , and

5) fails if a balancing) does not occur.

In the statement

```
        'A(B*C)(E/F)'  BAL  RPOS(0)
```

GBAL is called three times. First it matches the A but RPOS(0) fails. Next, GBAL extends the string matched by BAL to include (B*C), but again RPOS(0) fails. Finally GBAL matches (E/F), which brings the total string matched by BAL to A(B*C)(E/F) .

Insight into the behavior of BAL can be gained from use of ALLBAL:

```
        ALLBAL  =  BAL $ OUTPUT  FAIL
```

When used in the unanchored mode, a statement such as

```
        '((A+(B*C))+D)'  ALLBAL
```

prints out every balanced expression. The output for this case is

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

BAL facilitates the manipulation of algebraic and functional expressions. Programs using BAL to translate algebraic expressions from Polish to infix notation, and vice versa, appear in Chapter 4.

3. ARBNO

ARBNO is a mnemonic for "arbitrary number of." ARBNO(pattern) is a primitive function whose value is a pattern structure that matches zero or more consecutive occurrences of strings matched by its argument. When encountered by the scanner in the forward direction, ARBNO(pattern) matches the null string. When 'backed into,' it tries to increase the length of the substring matched by its argument. In the statements

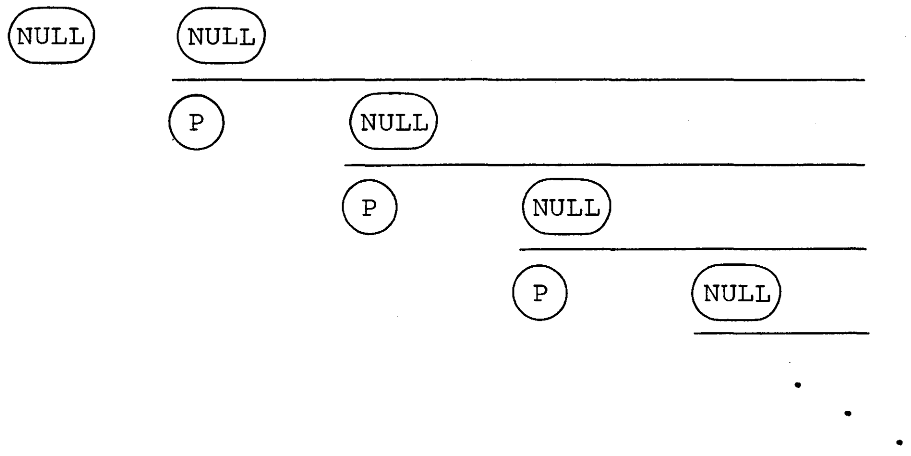
```
     &ANCHOR  =  1
     SUBSTR  ARBNO(LEN(3))  RPOS(0)
```

the pattern match succeeds only if the size of SUBSTR is zero or a multiple of three.

ARBNO(P) may be thought of as the infinite pattern

```
     NULL  |  P  (NULL  |  P  (NULL  |  P  (NULL  |  P  (......))))
```

A bead diagram is perhaps more illuminating.

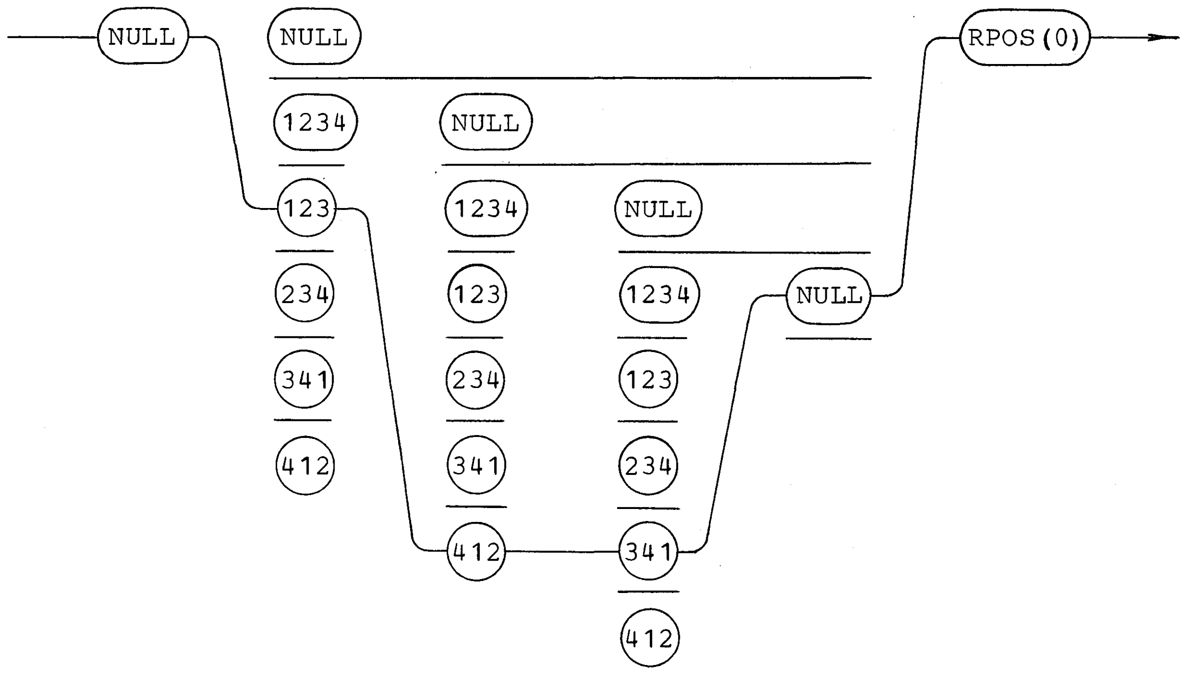


In the following example the argument of ARBNO has several alternatives.

```
        &ANCHOR  =  1
        P  =  '1234'  |  '123'  |  '234'  |  '341'  |  '412'
        ARBNOTEST  =  ARBNO(P)  $  OUTPUT  RPOS(0)
        '123412341'  ARBNOTEST
END
```

The following bead diagram for ARBNOTEST illustrates how alternatives are handled. The output from the program above is a blank line (resulting from the null string), and then

```
1234
12341234
1234123
123
123412
123412341
```



BREAK and SPAN can frequently be used in place of ARBNO. For example,

```
ARBNO(' ')
```

can usually be replaced by

```
SPAN(' ')
```

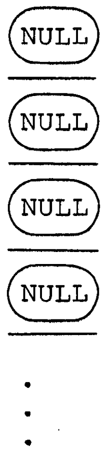
or, if necessary,

```
NULL  |  SPAN(' ')
```

ARBNO is relatively slow and should be avoided if some other pattern will suffice.

4. SUCCEED

The variable SUCCEED has a pattern structure as its initial value. SUCCEED matches the null string when first encountered by the scanner moving left to right through a pattern. If a subsequent failure causes the scanner to back up to SUCCEED seeking an alternative, SUCCEED again matches the null string. Thus, SUCCEED always matches the null string, both in the forward direction and when alternatives are sought. SUCCEED has a bead representation where all implicit alternatives are the null string.



Since the number of implied alternatives is infinite, the scanner can never back through SUCCEED.

Practical uses for SUCCEED seem limited. However, the light-hearted programmer can use SUCCEED and FAIL to produce pattern matches that never terminate:

```
SAWTOOTH  =  SUCCEED  (LEN(1)  ARB) $ OUTPUT  FAIL
```

Since FAIL repeatedly causes the scanner to back up and retry ARB, LEN(1) ARB matches first one character, then two, and so on up to the length of the subject string. Each substring matched by LEN(1) ARB is printed. Eventually ARB cannot match a longer string and fails, causing the scanner to back into SUCCEED. SUCCEED matches the null string and the entire process repeats itself.

If the pattern SAWTOOTH is used in the statement

```
'XXXXXX'  SAWTOOTH
```

pattern matching does not terminate, and the following output is produced.

```
X
XX
XXX
XXXX
XXXXX
XXXXXX
X
XX
XXX
XXXX
XXXXX
XXXXXX
X
XX
.
.
.
```

SAWTOOTH can never terminate successfully because of the FAIL, and can never terminate in failure because of the SUCCEED.

P. Cursor Position

The unary operator @ is called the cursor position operator. Its operand is a variable. The value of @X is a pattern structure that matches the null string and assigns the current cursor position as an integer value of the variable X. Assignment of the cursor postion to the operand of the @ operator takes place as immediate value assignment. Value is assigned when the cursor position operator is encountered during pattern matching, not following successful completion.

Execution of the following statements assign the integer value 5 to the variable HEAD.

```
        &ANCHOR  =  0
        'TEST AT OPERATOR'  @HEAD  'AT'
```

Pattern matching finally succeeds when the cursor is initially positioned to the left of the AT. The cursor position at this point is 5, the value assigned to HEAD.

Locating the rightmost instance of a pattern in a string is relatively easy utilizing the cursor position operator. The following statements can be used to locate and remove the rightmost blank in a string of characters.

```
        &ANCHOR  =  0
        STR  ' '  @RTPOS  FAIL
        STR  TAB(RTPOS - 1) . HEAD  ' '  =  HEAD
```

Since the unanchored mode is used, the first pattern matching statement assigns a cursor position to RTPOS for each blank in STR. Although failing ultimately, the final value of RTPOS is the cursor position to the right of the last blank. The replacement statement uses TAB(RTPOS - 1) to locate and remove the rightmost blank.

Q. Unevaluated Expressions

The unary operator * postpones the evaluation of its operand. If E is an expression, then *E is an unevaluated expression. The unevaluated expression is evaluated when

1) the scanner encounters *E as part of a pattern structure, or
2) *E is used as the argument of the primitive function EVAL.

In this chapter, unevaluated expressions, often simply called expressions, are considered only in the context of pattern matching. A detailed discussion of EVAL appears in Chapter 4.

If an unevaluated expression appears as part of a pattern, the expression is evaluated when encountered during pattern matching. If evaluation of the expression is successful, the value becomes part of the pattern structure and pattern matching continues. If evaluation of the expression fails, the scanner backs up seeking alternatives. Failure during evaluation of an expression does not cause termination of pattern matching.

A typical use for unevaluated expressions is motivated by the following example. A deck of data cards indexed in the first three columns with numbers from 1 to 999 is to be checked for the proper sequence.

```
        &ANCHOR  =  1
        N  =  1
        BLANKS  =  '  '
LOOP    CARD  =  INPUT                                :F(OK)
        CARD  (BLANKS N)  . SW  |  '    '  NULL . SW  :F(NOGOOD)
        N  =  DIFFER(SW)  N + 1
        EQ(SIZE(N)  + SIZE(BLANKS),3)                 :S(LOOP)
        BLANKS  ' '  =                                :(LOOP)
OK      OUTPUT  =  'DECK IS WELL ORDERED.'            :(END)
NOGOOD  OUTPUT  =  'CARD  '  CARD  '  IS OUT OF ORDER.' :(END)
END
```

Typical data are the following cards listing the best selling nonfiction books for 1965.

```
Column 1
↓
  1. MARKINGS, DAG HAMMARSKJOLD
  2. THE ITALIANS, LUIGI BARZINI
  3. SIXPENCE IN HER SHOE,
       PHYLLIS MCGINLEY
  4. REMINISCENCES, DOUGLAS MACARTHUR
       .
       .
       .
 10. JOURNAL OF A SOUL, POPE JOHN XXIII
 11. THE OXFORD HISTORY OF THE AMERICAN
       PEOPLE, SAMUEL ELIOT MORISON
 12. THE WORDS, JEAN-PAUL SARTRE
       .
       .
       .
```

The main loop is executed once for each card. Matching for sequence numbers or leading blanks is done using the pattern

```
          (BLANKS N) . SW  |  '   '  NULL . SW
```

The value of N is the number sought. BLANKS has a value of zero, one or two blanks such that SIZE(BLANKS N) is 3. SW is a variable which, following a successful match, is nonnull if the sequence number is found, and is null if three blanks are found instead. SW is used to determine if N should be incremented for the next iteration. When the SIZE(N) changes, as it does when N changes from 9 to 10 , a blank is removed from BLANKS in order to keep SIZE(BLANKS N) equal to 3.

The important point to observe in the example is the changing of the pattern. During execution, the value of N changes frequently and the value of BLANKS changes occasionally. As written, the pattern is evaluated for every iteration, and a new pattern structure is built.

N and BLANKS are the only portions of the pattern which change. Suppose a new pattern utilizing unevaluated expressions is specified outside of the loop.

```
          SEQNO  =  (*BLANKS  *N) . SW  |  '   '  NULL . SW
```

The pattern matching statement inside the loop becomes

```
          CARD  SEQNO                                   :F(NOGOOD)
```

The expressions *BLANKS and *N are not evaluated when the pattern is built. They remain unevaluated until SEQNO is used in a pattern matching statement.

During pattern matching the values of BLANKS and N are found and inserted into the already existing pattern structure. Thus, the pattern structure is built once, and only the continually changing values of BLANKS and N are updated on every iteration.

The following example incorporates the modifications using unevaluated expressions.

```
          &ANCHOR  =  1
          N  =  1
          BLANKS  =  '  '
          SEQNO  =  (*BLANKS  *N) . SW  |  '   '  NULL . SW
LOOP      CARD  =  INPUT                                :F(OK)
          CARD  SEQNO                                   :F(NOGOOD)
          N  =  DIFFER(SW)  N + 1
          EQ(SIZE(N) + SIZE(BLANKS),3)                  :S(LOOP)
          BLANKS  ' '  =                                :(LOOP)
OK        OUTPUT  =  'DECK IS WELL ORDERED.'            :(END)
NOGOOD    OUTPUT  =  'CARD '  CARD  '  IS OUT OF ORDER.' :(END)
END
```

Unevaluated expressions are valid arguments for primitive pattern-valued functions. The pattern structure for the function is built, but the argument remains unevaluated until pattern matching is performed. The following example uses an unevaluated expression as the argument of LEN, and thereby avoids the repeated formation of a pattern structure. The program takes input cards with left-adjusted data of length less than 40 characters, and produces output cards with the data right adjusted at column 40. For example, the cards

```
AKRON BEACON JOURNAL
ATLANTA CONSTITUTION
ATLANTA JOURNAL
BALTIMORE NEWS AMERICAN
```

become

```
                    AKRON BEACON JOURNAL
                    ATLANTA CONSTITUTION
                         ATLANTA JOURNAL
                 BALTIMORE NEWS AMERICAN
```

```
       BLANKS  =  '                                         '
       PADPAT  =  LEN(*(40 - SIZE(CARD)))  . PAD
LOOP   CARD  =  TRIM(INPUT)                                :F(END)
       GT(SIZE(CARD),40)                                   :S(PRINT)
       BLANKS  PADPAT                                      :F(ERROR)
       PUNCH  =  PAD  CARD                                 :(LOOP)
PRINT  OUTPUT  =  CARD                                     :(LOOP)
END
```

PADPAT is constructed once and only once. The argument of LEN is evaluated for each iteration of the loop.

In pattern matching, unevaluated expressions can be used in a variety of ways, as illustrated by the following examples.

Example 1

PAIR is a pattern that matches any two consecutive identical characters. PAIR uses LEN(1) to match any character, and immediate value assignment to assign the character as value of X. The expression *X that follows must match the same character as LEN(1).

```
       PAIR  =  (LEN(1)  $ X  *X)  . OUTPUT
       'COOK'  PAIR
       'COMMON'  PAIR
       'AARON'  PAIR
       'CHICKADEE'  PAIR
END
```

Output from the program is:

```
OO
MM
AA
EE
```

Example 2

Given any subject string STR and any pattern P, BIGP finds the longest substring of STR that P matches.

```
        BIGP  =  (*P $ TRY *GT(SIZE(TRY),SIZE(BIG))) $ BIG FAIL
```

BIGP uses two variables, BIG and TRY. During pattern matching, the value of BIG is the largest substring found. Before pattern matching, BIG must be initialized to the null string. TRY is assigned every substring that the pattern P matches. If TRY is longer than BIG, the value of BIG is updated.

BIGP utilizes unevaluated expressions in two ways. *P allows BIGP to be constructed without specifying the value of P. The value of P is determined during pattern matching. The predicate *GT(SIZE(TRY),SIZE(BIG)) is evaluated during pattern matching whenever *P matches a substring. It compares the size of TRY with the size of BIG. If the new substring is shorter, the predicate fails. Failure of a predicate or function during pattern matching causes the scanner to back up seeking alternatives. If the new substring is longer, the predicate succeeds, returning the null string as value. This null string is immediately matched. The variable BIG is then assigned the new substring as value. FAIL causes the scanner to back up and look for another substring matched by P.

The following is a test program for BIGP.

```
        BIGP  =  (*P $ TRY  *GT(SIZE(TRY),SIZE(BIG))) $ BIG  FAIL
        STR  =  'ON JANUARY 1, 1965, THE UNITED STATES MERCHANT '
.               'FLEET HAD 2529 VESSELS TOTALLING '
.               '29,632,000 DEADWEIGHT TONS.'
        P  =  SPAN('0123456789,')
        BIG  =
        STR  BIGP
        OUTPUT  =  'LARGEST NUMBER IS  '  BIG
        P  =  SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        BIG  =
        STR  BIGP
        OUTPUT  =  'LARGEST WORD IS  '  BIG
END
```

The output is

```
LARGEST NUMBER IS  29,632,000
LARGEST WORD IS  DEADWEIGHT
```

Example 3

Recursive definitions of patterns are possible using unevaluated expressions. The pattern structure for

```
        P  =  P  'Z'  |  'Y'
```

is constructed using the previous value of P. If P was null, the new value of P matches the strings Y and Z.

If the value of P is left unevaluated as in

```
        P  =  *P  'Z'  |  'Y'
```

the value of P at pattern matching time (which is *P, 'Z' | 'Y') replaces *P, giving rise to a recursive definition. The pattern P matches either Y or anything matched by P followed by Z. Therefore, since P matches Y, it also matches YZ. Since P matches YZ, it also matches YZZ, etc. Thus, P matches strings of the form

```
        Y
        YZ
        YZZ
        YZZZ
        .
        .
        .
```

A test program for the recursive definition of P follows.

```
        P  =  *P  'Z'  |  'Y'
        PO  =  P . OUTPUT
        'Y'  PO
        'YZZZ'  PO
        'XYZ'  PO
        'YZZX'  PO
        'AYZZZZB'  PO
END
```

Output from the program is

```
Y
YZZZ
YZ
YZZ
YZZZZ
```

<u>Example 4</u>

Recursive definitions can be quite complicated, as in the following example which recognizes a simple class of arithmetic expressions.

```
       &ANCHOR  =  1
       VARIABLE  =  ANY('XYZ')
       ADDOP  =  ANY('+-')
       MULOP  =  ANY('*/')
       FACTOR  =  VARIABLE  |  '('  *EXP  ')'
       TERM  =  FACTOR  |  *TERM  MULOP  FACTOR
       EXP  =  ADDOP  TERM  |  TERM  |  *EXP  ADDOP  TERM
LOOP   STRING  =  TRIM(INPUT)                            :F(END)
       STRING  EXP RPOS(0)                               :F(NOGOOD)
       OUTPUT  =  STRING  '  IS AN EXPRESSION.'          :(LOOP)
NOGOOD OUTPUT  =  STRING  '  IS NOT AN EXPRESSION.'      :(LOOP)
END
```

Output for typical data is

```
X+Y*(Z+X)  IS AN EXPRESSION.
X+Y+Z  IS AN EXPRESSION.
XY  IS NOT AN EXPRESSION.
```

Example 5

A call to a programmer-defined function is an expression and can appear in a pattern as an unevaluated expression. Evaluation of the function takes place during pattern matching. Failure of the function call causes the scanner to back up seeking alternatives. On success, the value of the function call is treated as a pattern, and matching continues. There are no special restrictions on the procedure called by the function, so pattern matching may be used within the called procedure.

The following program uses one statement to match a number of different patterns against a single subject string. The patterns are read from input cards one at a time.

```
        DEFINE('NEWPAT()')
        DEFINE('BUMP()')                        :(TEST)
*
NEWPAT  OUTPUT  =  TRIM(INPUT)                  :F(NEWEND)
        NEWPAT  =  ARB  OUTPUT                  :(RETURN)
NEWEND  NEWPAT  =  ABORT                        :(RETURN)
*
BUMP    X  =  X + 1                             :(RETURN)
*
        TEST     STR  =  'ABCDACDBADBCDB'
        STR  SUCCEED  *NEWPAT()  *BUMP()  FAIL
        OUTPUT  =
        OUTPUT  =  X  '  OF THE PATTERNS ABOVE MATCHED  ' STR
END
```

Two functions, NEWPAT and BUMP are defined. NEWPAT reads a pattern from the input, prints it, and returns the pattern preceded by ARB as the value of the function. If no patterns are left on the input, the pattern ABORT is returned as value of NEWPAT. The function BUMP increases the value of the variable X by one each time it is called.

In the test pattern, the functions NEWPAT and BUMP appear as unevaluated expressions bounded by SUCCEED and FAIL. Each time NEWPAT() is evaluated during pattern matching, a new pattern structure is returned as value. Since the first element of the pattern structure is ARB, the entire string STR is examined for the input pattern. If the pattern structure for NEWPAT() fails, the scanner backs up to SUCCEED and restarts causing NEWPAT() to be re-evaluated, reading in a new pattern. If matching succeeds, BUMP() is evaluated causing X to be incremented. FAIL then causes the scanner to back up to SUCCEED continuing the process. Pattern matching terminates when input is exhausted and the value of NEWPAT() is the pattern structure for ABORT.

Output from the program consists of the patterns read from the input followed by a summary line printing the number of patterns matched successfully.

```
ABCD
ABDC
ACBD
ACDB
ADBC
ADCB
BACD
BADC
BCAD
BCDA
BDAC
BDCA
CABD
CADB
CBAD
CBDA
CDAB
CDBA
DABC
DACB
DBAC
DBCA
DCAB
DCBA

5 OF THE PATTERNS ABOVE MATCHED  ABCDACDBADBCDB
```

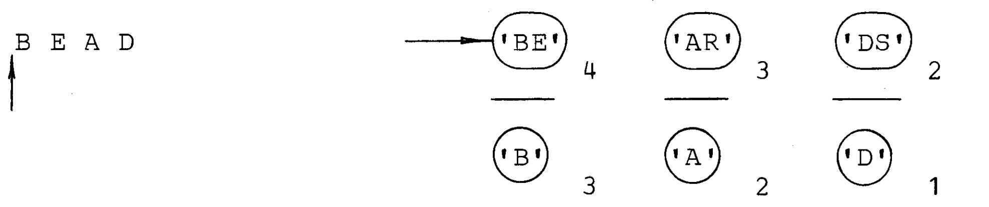
R. Quickscan Mode

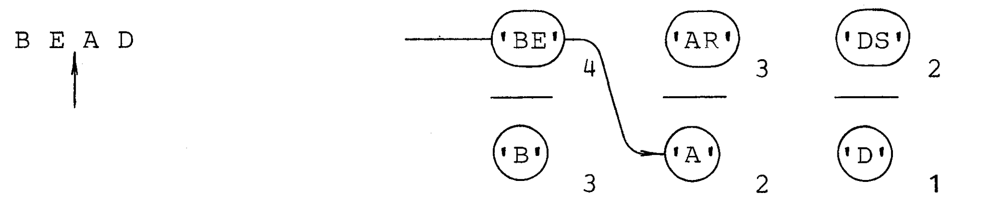
The keyword &FULLSCAN initially has a zero value, signifying the normal or quickscan mode of pattern matching. In the quickscan mode, the scanner uses a number of heuristics to avoid looking at alternatives which cannot possibly lead to a successful match. Hence, the scanner operates on the assumption that the programmer is not interested in how matching is done, but only in the outcome. Typically, patterns concerned with how matching is done employ immediate value assignment and/or unevaluated expressions. Patterns which do not use these features can and should be used in the quickscan mode. Patterns using immediate value assignment and unevaluated expressions may produce unexpected results in the quickscan mode. This section describes the heuristics used by the scanner to speed up pattern matching. It points out where unexpected results may arise and what can be done about them.

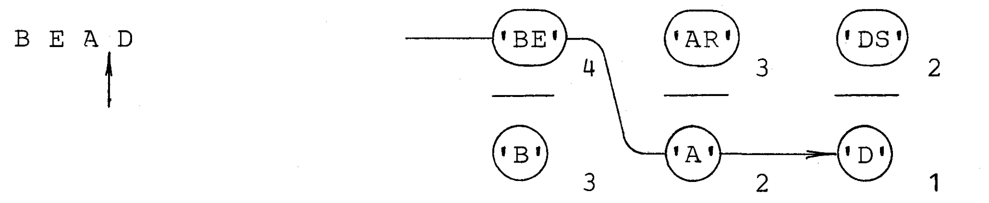
This chapter so far has been concerned with the basic components of patterns. No consideration has been given to the context in which a component occurs. The basic notion of the quickscan mode is quite simple: Before a component or bead is matched, its context is examined to see if matching should be attempted, terminated, or an alternative sought. The easiest question to answer is whether the number of characters remaining in the subject string is sufficient to successfully complete a match. Consider the following example.

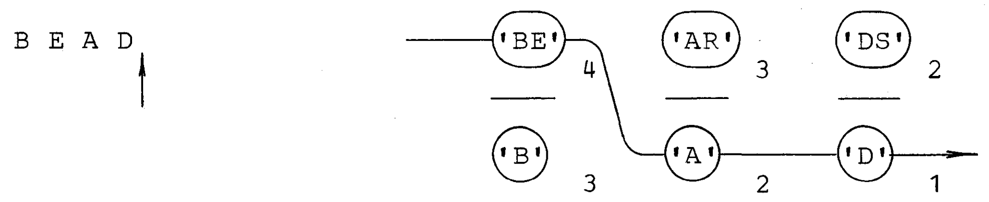
```
     BD  =   ('BE' | 'B')   ('AR' | 'A')   ('DS' | 'D')
     'BEAD'  BD
```

Three of the possible strings matched by BD are too long: BEARDS, BEARD, and BARDS. The scanner should avoid them if possible. In the bead diagram which follows, a number is associated with each bead. The number represents the minimum number of characters necessary to match that bead and anything that follows. If the number is greater than the number of characters remaining in the subject string, the scanner does not attempt to match the bead against the subject string.







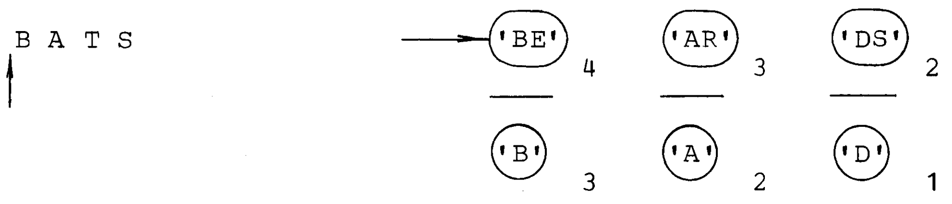


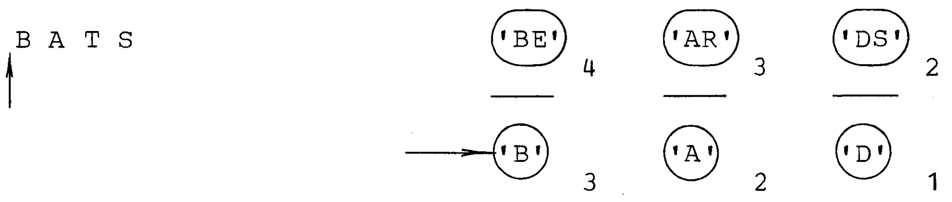
The components AR in step 2 and DS in step 3 are not tried. AR cannot match, since two characters remain in the subject string and at least three are necessary. Similarly, DS is not tried because one character remains and at least two were required.

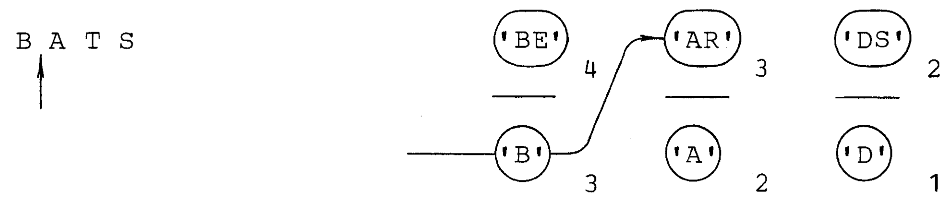
In the unanchored quickscan mode, the scanner does <u>not</u> move the initial position of the cursor if insufficient characters remain in the subject string. Consider the following example.

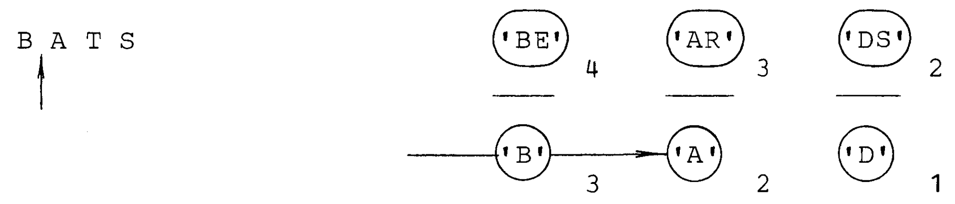
```
&ANCHOR  =   0
'BATS'  BD
```

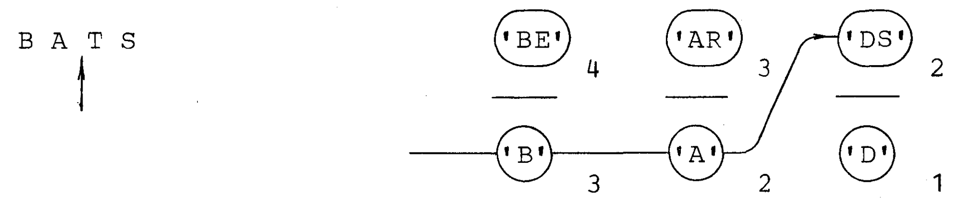
Matching fails with the cursor initially positioned to the left of the subject string. It is then moved to the left of the A. Since three characters remain in the subject string, only B is tried. Failing to match B , the scanner recognizes that further repositioning of the cursor is useless.

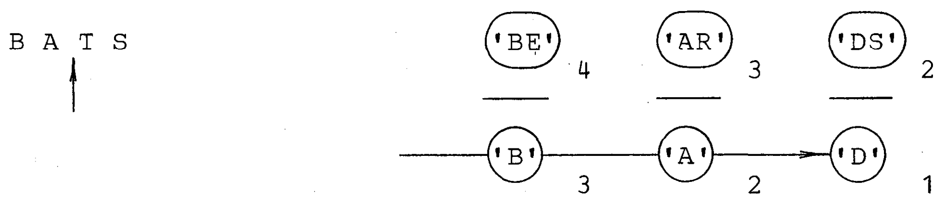


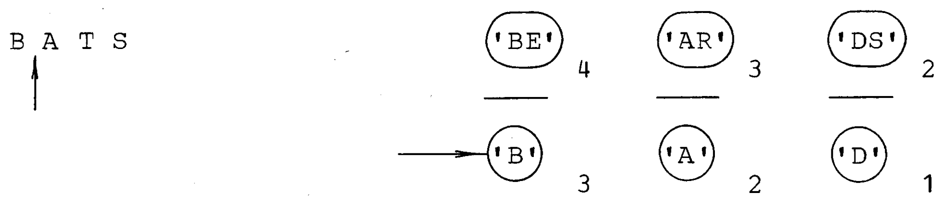








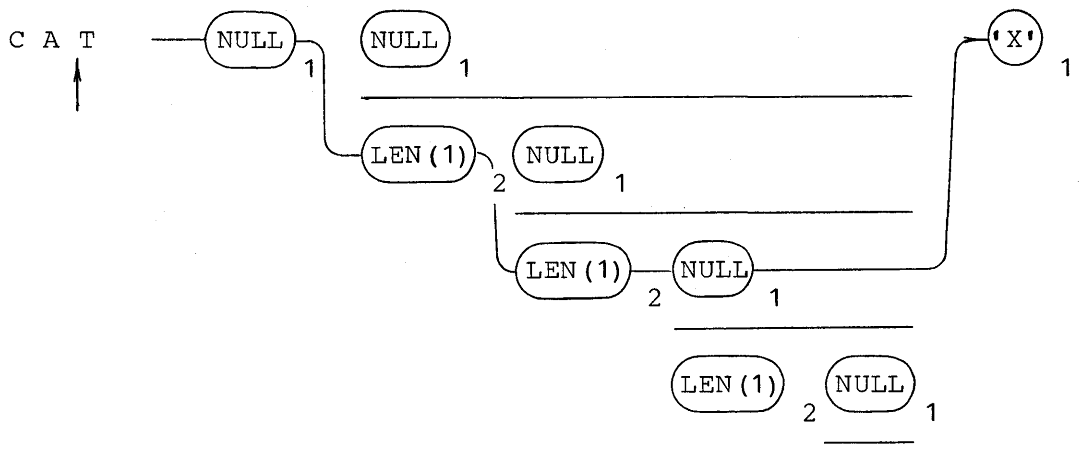




In the quickscan mode, the scanner distinguishes between two kinds of failure: 1) failure to match, as when X is compared to T; and 2) failure because too few characters remain in the subject string. In the latter case, the scanner does not allow ARB to match a longer substring, nor does it move the initial position of the cursor in unanchored mode. Consider the following pattern matching statement executed in the unanchored mode:

```
'CAT'  ARB  'X'
```

Clearly the match cannot succeed. When the scanner reaches the state shown in the diagram below, ARB can no longer extend the substring it matches. ARB indicates failure because of too few characters. The scanner does not reposition the cursor, and matching fails.



A similar situation arises in the anchored mode for such patterns as

```
'CAT'  ARB  ARB  'X'
```

The first ARB matches the null string. The second ARB matches the null string, C , and CA before it fails for lack of room. The scanner, therefore, does not seek an implicit alternative for the first ARB, and terminates pattern matching in failure.

In the quickscan mode, the scanner recognizes a special case for ARBNO. When backed into, ARBNO(P) tries to extend the substring matched by finding another instance of P. If P is null or has null alternatives, behavior like SUCCEED may result. The scanner tries to prevent this. When backed into, ARBNO(P) examines the substring matched by the last instance of P. If this substring is null, ARBNO does not try to extend the substring matched by finding an additional instance of P, but backs up to the last instance of P and seeks an alternative to the null string.

For example, in the quickscan mode, ARBNO(NULL) looks like NULL | NULL . The first NULL appears because NULL is always attempted independently of the argument to ARBNO. The second NULL comes from the argument.

Behavior of ARBNO(NULL | 'X') can be deduced from the output generated by the following statement.

```
    '*XXX'  ('*'  ARBNO(NULL | 'X')) $ OUTPUT  FAIL
```

The output is

```
*
*
*X
*X
*XX
*XX
*XXX
*XXX
```

Left recursion in a pattern structure, as illustrated by

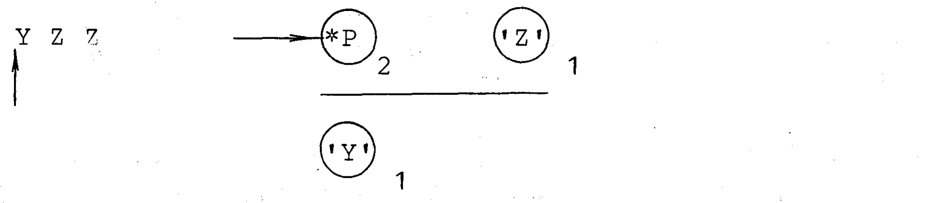
```
    P  =  *P  'Z'  |  'Y'
```

could be a problem because it might put the scanner in a loop, resulting in error termination. In the quickscan mode, recursive loops are broken whenever possible. Most looping problems are avoided by a look-ahead feature that compares the number of characters remaining with the number of characters required together with the assumption that any unevaluated expression matches at least one character.

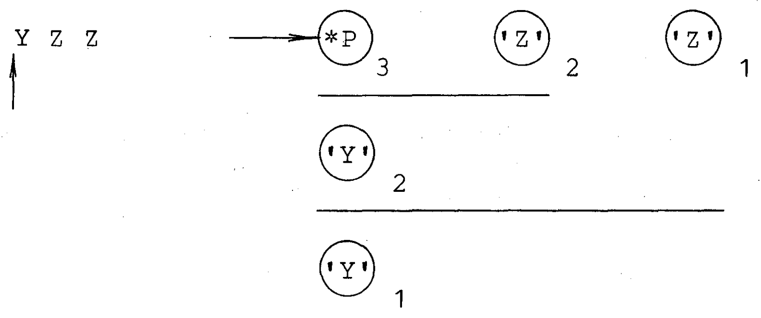
As an example, consider the following statement:

```
    'YZZ'  P
```

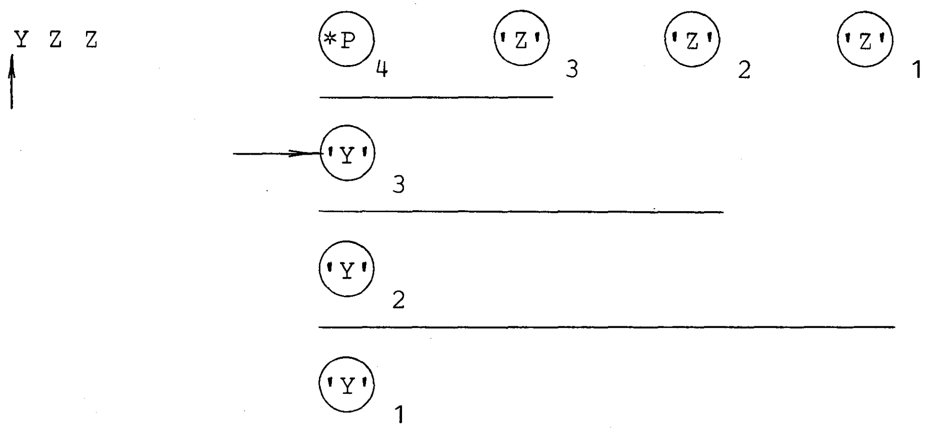
It is convenient to think that whenever the bead for *P is encountered, it expands into a bead diagram for the current definition of P. The process is illustrated by the following diagram.



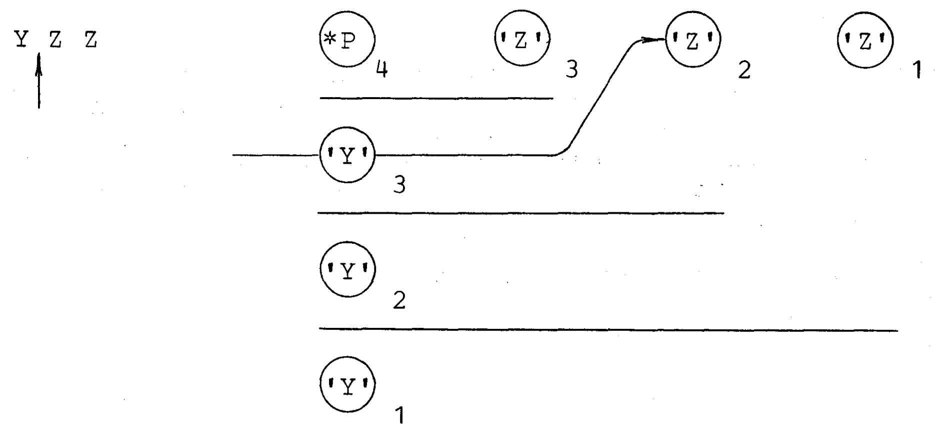
Y Z Z
*P
2
'Z'
1
'Y'
1



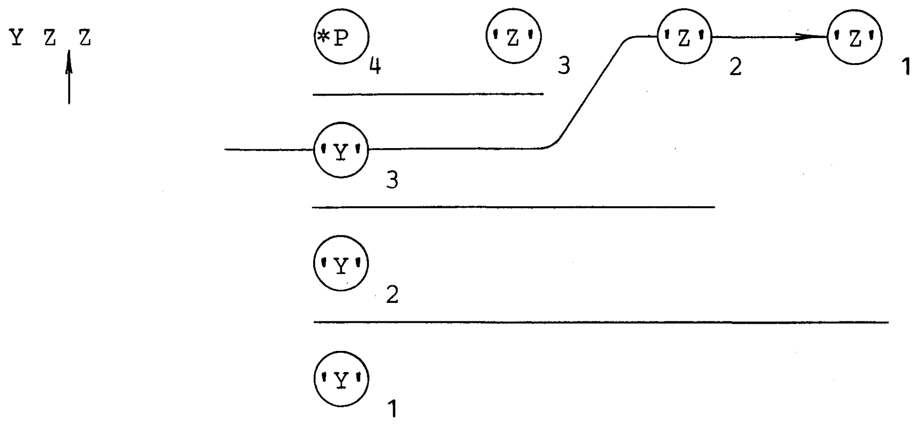
Y Z Z
*P
3
'Z'
2
'Z'
1
'Y'
2
'Y'
1



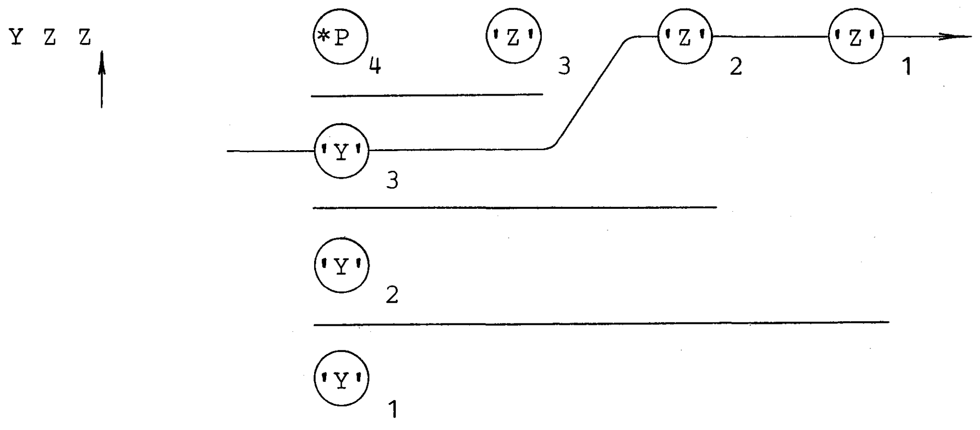
Y Z Z
*P
4
'Z'
3
'Z'
2
'Z'
1
'Y'
3
'Y'
2
'Y'
1



Y Z Z
*P
4
'Z'
3
'Z'
2
'Z'
1
'Y'
3
'Y'
2
'Y'
1

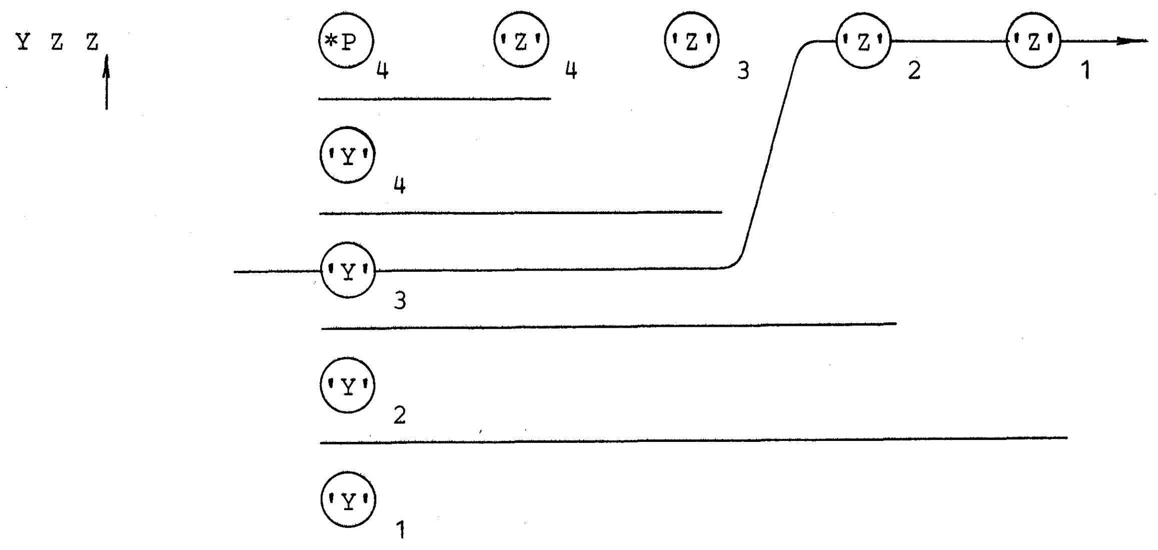


The final state is



When the minimum number of characters required by *P reaches 4, the recursive loop is broken and the alternative Y is tried, leading to a successful match.

The assumption that *P matches at least one character does not affect the outcome of the previous example. Had zero characters been assumed, one more iteration of the loop would have been required, and the final diagram would have been as follows.



However, the one-character assumption keeps the following equivalent statements from terminating in error.

```
P  =  *P  *Q  |  'Y'
Q  =  'Z'
```

If both *P and *Q can match the null string, the bead diagram grows until error termination results. With the one-character assumption, the two equivalent examples above behave similarly.

There are a number of pathological patterns which cause error termination. The following are typical.

```
P  =  *P
P  =  NULL  *P
```

Even the one-character assumption cannot interrupt the recursive loop, because as the bead diagrams grow, the minimum number of characters on the *P bead does not change.

Assuming a one-character minimum for unevaluated expressions can lead to difficulties:

```
PAT  =  *W  *X  *Y  *Z
```

The shortest string PAT matches is of length four. The following match, straightforward as it seems, fails.

```
W  =  'C'
X  =  'A'
Y  =  'T'
Z  =
'CAT'  PAT
```

As seen in the next section, the match succeeds if the fullscan mode is used.

Patterns such as BIGP, described in the section on unevaluated expressions, can produce unexpected results in the quickscan mode.

```
        BIGP  =  (*P $ TRY  *GT(SIZE(TRY),SIZE(BIG))) $ BIG  FAIL
```

The expression *GT(SIZE(TRY),SIZE(BIG)) is assumed to require one character when, in fact, it matches the null string. Therefore, the quickscan mode prevents *P from matching any substring which includes the last character of the subject string. Hence, in the statements

```
        P  =  SPAN('0123456789,')
        '1234.56  789,312'  BIGP
```

the final value of BIG is 1234 rather than the expected 789,312 . Again, as seen in the next section, the fullscan mode prevents such difficulties.

In summary, the following heuristics are used in the quickscan mode to improve the efficiency of pattern matching:

1) continual comparison of the number of characters remaining in the subject string against the number of characters required,

2) repositioning of the cursor in the unanchored mode only if sufficient characters remain,

3) refusal to extend the substring matched by ARB or to reposition the cursor if failure is caused by too few characters,

4) refusal to extend substring matched by ARBNO(P) if the last match of P was the null string, and

5) assumption that unevaluated expressions must match at least one character.

S. Fullscan Mode

The fullscan mode of pattern matching is entered by assigning a nonzero value to the keyword &FULLSCAN. In the fullscan mode, all heuristics to improve pattern matching efficiency are turned off. Each component of a pattern is matched independently of its context. Furthermore, when unanchored, the initial position of the cursor is moved through the entire subject string.

The following example, which prints all possible nonnull substrings of a subject, suggests applications of the fullscan mode.

```
        &ANCHOR  =  0
        &FULLSCAN  =  1
        '12345'  (LEN(1) ARB) $ OUTPUT  FAIL
END
```

Output from the program is:

```
1
12
123
1234
12345
2
23
234
2345
3
34
345
4
45
5
```

If &FULLSCAN had been zero, the initial position of the cursor would not have been moved, and only the first five lines would have been printed.

A more useful example, which can only be done in the fullscan mode, is back referencing. This pattern succeeds only if a subject string has two identical nonoverlapping substrings of length 3:

```
       BACKR  =  LEN(3) $ X  ARB  *X
```

The statement

```
       'ABCDEFGBCDA'  BACKR
```

succeeds and X has the value BCD . The statement above does not work in the quickscan mode. When LEN(3) matches ABC , ARB eventually matches DEFGBCD and then fails because X is assumed to match one character. The condition is recognized in the quickscan mode, preventing the initial position of the cursor from being moved. Hence, matching fails without BCD ever being matched by LEN(3).

In the fullscan mode, the tests of ARBNO for null arguments are turned off. ARBNO(NULL) and ARBNO(NULL | 'X') behave like SUCCEED, except that they eventually cause error termination. The statement

```
       '*XXX'  ('*'  ARBNO(NULL  |  'X')) $ OUTPUT  FAIL
```

generates output lines consisting of a single * until error termination.

Recursive patterns such as

```
       P  =  *P  'Z'  |  'Y'
```

do not work because the recursive loop is not broken. Execution of a statement with such a pattern results in error termination.

Patterns such as

```
        PAT  =  *W  *X  *Y  *Z
```

work for subject strings having fewer than four characters because the one-character assumption no longer holds.

The next two examples compare the results of programs run in quickscan and fullscan modes.

Example_1

This program prints combinations of characters taken three at a time from a subject string.

```
        DEFINE('F(X,Y,Z)')
        COMB3  =  LEN(1) $ A  ARB  LEN(1)  $ B  ARB  LEN(1)  $ C
.                 *F(A,B,C)  FAIL
        '123456'  COMB3                                  :(END)
F       OUTPUT  =  X Y Z                                 :(RETURN)
END
```

```
        Output from Quickscan            Output from Fullscan

                123                              123
                124                              124
                125                              125
                                                 126
                                                 134
                                                 135
                                                 136
                                                 145
                                                 146
                                                 156
                                                 234
                                                 235
                                                 236
                                                 245
                                                 246
                                                 256
                                                 345
                                                 346
                                                 356
                                                 456
```

Example_2

This program generates wallpaper. Using SUCCEED and FAIL to bracket a pattern, endless output occurs in the quickscan mode. In the fullscan mode, output is truncated by error termination.

```
        PONG  =  SUCCEED  (LEN(1)  ARBNO(LEN(1)  |  NULL))
.                $ OUTPUT  FAIL
        PING  =  'XXXXXXXXXX'
        PING  PONG
END
```

Output from Quickscan

```
X
XX
XXX
XXXX
XXXXX
XXXXXX
XXXXXXX
XXXXXXXX
XXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXX
XXXXXXXX
XXXXXXX
XXXXXX
XXXXX
XXXX
XXX
XX
X
X
XX
XXX
XXXX
XXXXX
XXXXXX
XXXXXXX
XXXXXXXX
XXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXX
.
.
.
```

Output from Fullscan

```
X
XX
XXX
XXXX
XXXXX
XXXXXX
XXXXXXX
XXXXXXXX
XXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
```

Chapter 3. Predicates and Primitive Functions

A. Introduction

A function is an operation upon a number of arguments. The value of a function is computed by a procedure. Primitive functions are implemented by procedures built into the SNOBOL4 system. Procedures for programmer-defined functions are included in the source program.

Syntactically, a function call is recognized as an identifier used for a function name, followed by a list of arguments separated by commas and enclosed in parentheses. An example is

```
      IDENT(A,'TREE')
```

An argument of a function call may be any expression. Execution of a function call causes the expressions for the arguments to be evaluated and the values passed to the procedure. Thus, the procedure gets only the values of the arguments and not the expressions. Consider the following statements:

```
        A  =  'APPLE'
        B  =  'SEED'
        APPLE  =  'FRUIT'
        SEED  =  'TREE'
        APPLESEED  =  'FRUITTREE'
        IDENT($A $B,$(A B))
```

FRUITTREE is the value of each argument to IDENT. The two strings FRUITTREE are all that the procedure for IDENT knows of its arguments.

A variable such as C is an expression, albeit a degenerate one. Thus, if

```
        C  =  'CLAW'
        D  =  'TIGER'
```

the call

```
        IDENT(C,D)
```

passes the strings CLAW and TIGER (not C and D) as arguments to the procedure for IDENT . Furthermore, since the procedure for IDENT knows nothing about C and D, it cannot possibly change their values.

Any omitted argument is assigned the null string as value. Thus, IDENT(E) compares the value of E and the null string. Too many arguments in the call of a primitive function cause error termination.

A function call is an expression and has a value. The value of a function call may be of any data type. A programmer must always be aware that a function call has a value, even if it is the null string. Otherwise, as later examples illustrate, unexpected results may arise.

A function call may succeed or fail, depending upon the outcome of the associated procedure. If the procedure for a function is successful, the value computed by the procedure becomes the value of the function call. If the procedure fails, the function call fails.

This chapter, although entitled "Predicates and Primitive Functions," describes only those primitive functions that logically cannot be described elsewhere. Those dealing with pattern matching, input/output, arrays, and programmer-defined data types are described in appropriate chapters. Functions and page references are included in the index.

B. Numerical Predicates

Several primitive functions are concerned with testing relations between arguments. These functions, which succeed or fail depending on whether the relation is true or false, are called predicates. If a predicate is successful, the value of the call is the null string.

1. LT, LE, EQ, NE, GE, and GT

A predicate test, such as GE(X,Y), succeeds if X stands in the given relation to Y. The arguments to numerical predicates must be integers or numeral strings. Thus, if

```
        X  =  17
        Y  =  '3'
```

then GE(X,Y) succeeds and LT(X,Y) fails. If an argument is omitted, it is assigned the null string, which is treated as zero. If M is 2, then EQ(M) fails, but EQ(M - 2) succeeds returning the null string.

Numerical predicates frequently are used for loop control. For example, if N has as value the number of times a loop has been executed and M is the limit on N, the following statement checks N against M, and increments N if N is less than M.

```
        N  =  LT(N,M)   N + 1                     :S(LOOP)F(OUT)
```

Evaluation of the object expression takes place before assignment is made. Thus, the evaluation of LT(N,M) takes place before N is incremented. If LT(N,M) succeeds, the value is the null string. Concatenation of the null string with N + 1 does not affect N + 1, so N is properly incremented. Furthermore, since the statement succeeds, control passes to LOOP.

If LT(N,M) fails, N is not incremented and control passes to the statement labelled OUT.

Placement of predicates in a statement is important. Consider the following statement, which looks as if it might be suitable for loop control.

```
          N  LT(N,M)  =  N + 1                        :S(LOOP)F(OUT)
```

The statement does not properly increment N. If N is 2 and M is 4, the value of N after execution of the statement is 32. The predicate LT(N,M), situated where it is, is treated as a pattern. Since LT(N,M) is null, the pattern matches the null string. The null string matched in the value of N is replaced by N + 1, leading to the unexpected result 32.

2. INTEGER

It is frequently desirable to test whether the value of a variable is an integer. The predicate test INTEGER(X) succeeds if the value of X is an integer or numeral string, and fails otherwise. Thus,

```
          INTEGER(X)
```

succeeds for

```
          X  =  3
          X  =  '3'
```

but fails for

```
          X  =  'INT'
          X  =  '3.0'
```

INTEGER is typically used to check data coming from the input stream. The following statements reject cards which do not contain a single numeral string left justified on the card.

```
LOOP      CARD  =  TRIM(INPUT)           :F(END)
          INTEGER(CARD)                  :S(PROCESS)F(REJECT)
```

Since the null string is equivalent to the integer 0, a blank card passes the integer test.

C. Object Comparison Predicates

There are several types of data predefined in the SNOBOL4 language. Programmer-defined data types can be added, as described in Chapter 5. Some data values, such as numbers, can be represented in different ways as different types of data. SNOBOL4 includes predicates to test whether two objects are identical or different.

1. __IDENT and DIFFER__

IDENT and DIFFER are functions of two arguments which may be of any data type. For the function call IDENT(X,Y) to succeed or for DIFFER(X,Y) to fail, the values of the arguments, X and Y, must be truly identical. The value of a function argument is a pointer to a data object or, in the case of integers and real numbers, the value is the data object itself.

Each distinct string of characters appears in storage once and only once. Execution of

```
          X  =  'BCD'
          Y  =  'B'  'CD'
```

results in X and Y having the same value. The string BCD appears once, and both X and Y point to it. IDENT(X,Y) therefore succeeds.

Pattern structures behave differently. Execution of the statements

```
          X  =  A  |  B
          Y  =  A  |  B
```

constructs two equivalent but physically distinct pattern structures. Thus, X and Y have different values, since they point to different copies of the pattern structure A | B . IDENT(X,Y) therefore fails.

However, if

```
          X  =  A  |  B
          Y  =  X
          IDENT(X,Y)
```

then IDENT(X,Y) succeeds since X and Y point to the same data object.

Integers and real numbers are data objects rather than pointers to data. Execution of

```
          X  =  3
          Y  =  2 + 1
```

assigns 3 to both X and Y. Comparison of X and Y by IDENT(X,Y) succeeds because the data objects are identical. Similarly, if

```
          X  =  3.0
          Y  =  3.0
```

then IDENT(X,Y) succeeds.

IDENT and DIFFER must be used with care when their arguments have different data types. If

```
          X  =  3
          Y  =  '3'
```

EQ(X,Y) succeeds as illustrated earlier. IDENT(X,Y) fails because the value of X is the integer 3, but the value of Y is the string 3.

Similarly, for

```
          X  =  3.0
          Y  =  3
```

IDENT(X,Y) fails because the values are not identical.

2. LGT

Lexical ordering can be tested using the predicate LGT(X,Y). Both arguments to LGT(X,Y) must be strings or integers. LGT(X,Y) succeeds, returning the null string, if the value of X is lexically greater than Y. Stated another way, LGT(X,Y) succeeds if X follows Y alphabetically. The order of the characters is implementation dependent. For example, on the IBM System/360 the EBCDIC encoding is used with the blank preceding letters and letters preceding digits. The value of &ALPHABET is a string of all characters in lexical order.

Consider, as an example, the problem of alphabetizing the characters in a string. That is, the string LABORATORIES is to be transformed into the string AABEILOORRST . The following program performs the conversion.

```
          &FULLSCAN  =  1
          &ANCHOR  =  1
          FLIP  =  (*HEAD  ARB) . HEAD  LEN(1) $ X  ARB . FILLER
.                  LEN(1) $ Y  *LGT(X,Y)
          STR  =  'LABORATORIES'
          OUTPUT  =  STR
LOOP      STR  FLIP  =  HEAD  Y  FILLER  X          :S(LOOP)
          OUTPUT  =  STR
END
```

Output is:

```
LABORATORIES
AABEILOORRST
```

FLIP matches the ordered portion of the string followed by two out-of-order characters with an arbitrary number of intervening characters.

```
          (*HEAD  ARB) . HEAD
```

matches the ordered portion of the string.

```
          LEN(1) $ X  ARB . FILLER  LEN(1) $ Y
```

matches any two characters. The unevaluated expression then tests if the two characters are out of order. If they are, the pattern match succeeds and a replacement is done to reverse them. If the two characters are in order, LGT(X,Y) fails, causing the scanner to back up and seek another pair of characters. By repeatedly executing the statement labelled LOOP, all unordered pairs of characters are interchanged. Pattern matching fails when the string is completely ordered.

D. Additional Primitive Functions

1. SIZE

SIZE expects a string or an integer as an argument. The value of SIZE is an integer which is the number of characters in the argument. Thus, the value of SIZE('SIZE') is 4, and the value of SIZE(16384) is 5.

2. REPLACE

One-for-one character replacement in a string may be accomplished using the function REPLACE. The value of REPLACE(X,Y,Z) is the string resulting from replacement in X of each character appearing in Y by the corresponding character in Z. As a result of executing the following statements,

```
     BINARY   =  '111001'
     ONESCOMP  =  REPLACE(BINARY,'01','10')
```

ONESCOMP has the value 000110 , obtained from BINARY by replacing all zeroes with ones, and ones with zeroes.

REPLACE normally succeeds, but it fails if

1) the second and third arguments have different length, or
2) the second or third argument is null.

Multiple occurrences of characters in the third argument are valid. Thus,

```
     REPLACE(S,'.,;:?!','      ')
```

replaces all punctuation marks with blanks.

In the case of the multiple occurrence of a character in the second argument, the rightmost correspondence holds. Thus, following execution of the statement

```
     TEXT  =  REPLACE('FEET','EE','AO')
```

the variable TEXT has value FOOT .

A particularly striking example of REPLACE is the following program that converts a deck of cards prepared on an 026 keypunch (BCD) to a deck using 029 keypunch code (EBCDIC).

```
LOOP      PUNCH  =  REPLACE(INPUT,"#@%<&","='()+")   :S(LOOP)
END
```

3. TRIM

TRIM is a primitive function whose argument must be a string or an integer. The value of TRIM is a string which is the argument value with all trailing blanks removed. Thus, the statements

```
          TEXT  =  'A PRIMITIVE FUNCTION
          SHORTTEXT  =  TRIM(TEXT)
```

gives SHORTTEXT the value A PRIMITIVE FUNCTION. The value of TEXT is not changed.

TRIM is frequently used with INPUT as its argument. Standard input reads 80 characters so TRIM(INPUT) provides a convenient way of shortening an input string.

4. DATE and TIME

DATE and TIME are primitive functions requiring no arguments. The value of DATE() is an 8 character string of the form month/day/year. For August 6, 1968, the value of DATE() is 08/06/68 .

The value of TIME() is an integer which is the elapsed time in milliseconds from the beginning of program execution. Compilation time is not included. On IBM 360 equipment the standard interval clock is updated only sixty times a second, so timing is approximate at best.

5. EVAL

EVAL is a primitive function whose argument must be an unevaluated expression or a string. If the argument is an unevaluated expression, the expression is evaluated to obtain the value of EVAL. If the argument is a string, the value of EVAL is the value of the expression represented by the string.

In the example which follows, the value of S is a string, and the value of U is an unevaluated expression. Both output statements print the integer 15 .

```
          S  =  'X + SIZE(X) * 10'
          U  =  *(X + SIZE(X) * 10)
          X  =  5
          OUTPUT  =  EVAL(S)
          OUTPUT  =  EVAL(U)
```

Any string or unevaluated expression which is a syntactically correct expression in SNOBOL4 may be evaluated by EVAL. A syntactic error in the argument of EVAL causes failure of EVAL. Thus, evaluation of E in the statements

```
          E   =   '5+6'
          SUM   =   EVAL(E)
```

fails since blanks are required around the + .

E. Negation (¬) and Interrogation (?)

Two predicates, specified by the unary operators ¬ and ? , test the success or failure resulting from evaluation of expressions. The negation operator ¬ fails if its operand succeeds, and succeeds if its operand fails. A null string is returned as value on success. The interrogation operator ? is the converse of ¬ . It succeeds, returning the null value if its operand succeeds, and fails if its operand fails.

Negation may be used to complement a predicate. For example, the following program reads an input deck and prints those cards that contain integers.

```
LOOP      CARD  =  TRIM(INPUT)                         :F(END)
          OUTPUT  =  INTEGER(CARD)   CARD              :(LOOP)
END
```

Suppose the converse program, one which prints all cards that are not integers, is desired. No predicate is available which succeeds when its argument is not an integer. However, the negation operator together with the predicate INTEGER suffices. Thus, the following program lists all noninteger cards.

```
LOOP      CARD  =  TRIM(INPUT)                         :F(END)
          OUTPUT  =  ¬INTEGER(CARD)   CARD             :(LOOP)
END
```

Complicated Boolean functions on the states of variables can be constructed using predicates and negation. For example, suppose the integer N is to be incremented provided at least one of the variables X, Y, or Z is null. The following statement tests the variables and, if the condition is satisfied, increments N.

```
          N   =   ¬(DIFFER(X)   DIFFER(Y)   DIFFER(Z))   N + 1
```

If X, Y, and Z are nonnull, the expression succeeds but the negation operation fails, and N is not incremented. If any variable is null, the corresponding DIFFER fails, causing the expression to fail. Negation succeeds and N is incremented.

Interrogation is used primarily to convert a function that returns a nonnull value into a predicate that succeeds or fails, but returns a null value. Thus, in the following statement, N is incremented if F(X) succeeds, but the value of F(X) is not concatenated with N + 1.

```
          N   =   ?F(X)   N + 1              :S(ON)F(OUT)
```

Chapter 4. Programmer-Defined Functions

A. Introduction

A programmer may define his own functions to perform specific operations. A program with programmer-defined functions must include:

1) a call to the primitive function DEFINE for each programmer-defined function, and

2) a procedure, written in SNOBOL4, for each function.

Procedures are written using formal arguments, and must adhere to special conventions for returning. Execution of the primitive function DEFINE communicates to the SNOBOL4 system:

1) the name of the function,
2) a list of formal arguments used in the programmer-defined procedure,
3) a list of variables local to the programmer-defined procedure, and
4) the entry point of the procedure.

B. The Primitive Function DEFINE

DEFINE is a primitive function of two arguments that returns a null string. The first argument is a prototype for the call of the function being defined, together with a list of local variables used by the function. The second argument is a label specifiying the entry point to the programmer-defined function. For example, execution of

```
     DEFINE('F(X,Y)L1,L2','FENTRY')
```

defines a function F with two formal arguments, X and Y. Two local variables L1 and L2 are used in the procedure whose entry point is the statement labelled FENTRY. Notice that there is no comma separating the prototype for the call from the list of local variables. Expressions may be used as arguments for DEFINE, provided their values are strings having the form shown above.

Often local variables are not needed, so it is permissible to omit the list of local variables. An example is

```
     DEFINE('G(Z)','GENT')
```

It is also permissible to omit the second argument, in which case the entry label is assumed to be the same as the function name. Thus,

```
DEFINE('COUNT(N)')
```

defines the function COUNT with entry label COUNT. Functions can be defined without any formal arguments. For example,

```
DEFINE('MARK()')
```

defines the function MARK with no formal arguments. Prototypes which are syntactically incorrect, such as those in

```
DEFINE('F')
```

and

```
DEFINE('F("A")')
```

cause error termination.

A statement containing the DEFINE function for a particular function must be executed before a call to that function is made. Thus, execution of the statements

```
X  =  F(FIRST,SECOND)
DEFINE('F(X,Y)')
```

results in error termination, since the function F is undefined at the time it is called.

C. Procedures for Programmer-Defined Functions

A procedure for a programmer-defined function is a set of SNOBOL4 statements. The label, provided explicitly or implicitly in the arguments of the associated DEFINE function, specifies the statement to which control is passed when a call is made to the function. Thus, during execution of the statement labelled ZSET in the example below, the call to UNION causes control to be passed to UN. Execution of ZSET is temporarily suspended while the value of UNION is being computed. Once the value of UNION has been computed, control returns to ZSET where computation is resumed using the value returned.

```
        DEFINE('UNION(X,Y)CH','UN')
          .
          .
          .
          .
ZSET    Z  =  SET1  UNION(SET2,SET3)  SET4
          .
          .
          .
          .
          .
          .
UN      UNION  =  X
ULOOP   Y  LEN(1) . CH  =                              :F(RETURN)
        UNION  BREAK(CH)                               :S(ULOOP)
        UNION  =  UNION  CH                            :(ULOOP)
```

The defining statement must be executed before the call is made. The procedure is called and should not be flowed into. The procedure may be transferred around or placed out of the way of program flow.

The statements constituting the procedure are written using the formal arguments whose values are supplied by arguments of a call.

Local variables should be declared when variables used in a procedure have values which should not be altered by a function call. In the definition of UNION, the value of the variable CH changes continually during evaluation of the function. The value of CH may be altered as a result of the call unless CH is declared as a local variable. Upon entry to a procedure, all local variables are given null string values. All statement labels, including labels in procedures, are global. Transfer can be made from a statement in one procedure to a statement in another.

The name of a function may be used as a variable in the procedure. The value of the function call is the value of the function name when execution of the procedure is complete. Thus, in the example above, the value of the call UNION(SET2,SET3) is the value of the variable UNION when the statement ULOOP fails, causing return to ZSET.

Return of control from a procedure to the calling statement is accomplished by transfer to one of the three system labels: RETURN, FRETURN, or NRETURN.

RETURN

Transfer to RETURN indicates that the function call is successful. The value of the function call is set to the value of the function name. Execution continues in the calling statement using the returned value.

FRETURN

Transfer to FRETURN indicates failure of the function call.

An example using both RETURN and FRETURN is the function PAL, which checks its argument to see if it is a palindromic string. PAL compares the argument string and its reverse. If they are identical, PAL transfers to RETURN, indicating success. Otherwise PAL transfers to FRETURN, indicating failure. Since the variable PAL is not used in the procedure, the value of PAL(PHRASE) is the null string on a successful return.

```
        DEFINE('PAL(STR)CH,S1,S2')
           .
           .
           .
TEST    PHRASE  =  TRIM(INPUT)                       :F(END)
        PAL(PHRASE)                                  :S(GOOD)F(NOGOOD)
           .
           .
           .
PAL     S1  =  STR
PALL    S1  LEN(1)  . CH  =                          :F(PTEST)
        S2  =  CH  S2                                :(PALL)
PTEST   IDENT(STR,S2)                                :S(RETURN)F(FRETURN)
           .
           .
           .
END
```

NRETURN

By transferring to the label NRETURN, a programmer-defined function may return a computed name rather than a value. A call to a function that returns a computed name may be used as the subject of an assignment statement. For example,

```
        F(X,Y)  =  X  Y
```

is a valid statement provided the function F returns by name using NRETURN. A further description of names is included in Chapter 5.

D. Execution of Programmer-Defined Functions

When a call to a programmer-defined function is made, the arguments of the call are evaluated first. Before execution of the procedure begins, the values of the following variables are saved on an internal stack in the order:

1) the name of the function,
2) all formal arguments, and
3) all local variables.

New values are then assigned to these variables as follows:

1) the name of the function is assigned the null string,
2) the formal arguments are assigned their values, and
3) all local variables are assigned the null string.

Consider the function UNION specified in the defining statement

```
        DEFINE('UNION(X,Y)CH','UN')
```

and called by UNION(SET2,SET3). Values of the variables UNION, X, Y, and CH at the time of a call are saved. New values for these variables are assigned as if the following statements had been executed.

```
UNION  =
X  =  SET2
Y  =  SET3
CH  =
```

Then control passes to the statement labeled UN.

When return from a procedure is made using RETURN,

1) the value of the function call is set to the value of the function name, and

2) the values of all variables saved at the time of the call are restored in reverse order.

When return is made using FRETURN,

1) the values of all variables saved at the time of the call are restored, in reverse order, and

2) the call fails.

When return is made using NRETURN,

1) the function call becomes a variable whose name is taken from the value of the function name, and

2) the values of all variables saved at the time of the call are restored, in reverse order.

A programmer-defined function may be called with more or fewer arguments than specified in the corresponding defining statement. If too few arguments are specified, the trailing omitted arguments are assigned null strings. If too many arguments are specified, the extra arguments are evaluated, but their values are ignored.

Example - Union, Intersection, and Negation

This example includes three functions that perform the union, intersection, and negation of sets of characters, and a short test program. Notice that the procedures follow the defining statements in the listing. However, by transferring around the procedures, the defining statements are executed one after another. The test program then makes calls to the procedures.

```
START
UNION      DEFINE('UNION(X,Y)CH','UN')                     :(INTER)
*
UN         UNION  =  X
ULOOP      Y  CHAR  =                                      :F(RETURN)
           UNION  CHTEST                                   :S(ULOOP)
           UNION  =  UNION  CH                             :(ULOOP)
*
*
INTER      DEFINE('INTER(X,Y)CH','IN')                     :(NEG)
*
IN         X  CHAR  =                                      :F(RETURN)
           Y  CHTEST                                       :F(IN)
           INTER  =  INTER  CH                             :(IN)
*
*
NEG        DEFINE('NEG(X)CH,HEAD','NG')                    :(PATDEF)
*
NG         NEG  =  UNIVERSE
NLOOP      X  CHAR  =                                      :F(RETURN)
           NEG  CHLOC  =  HEAD                             :(NLOOP)
*
*
PATDEF     CHAR  =  LEN(1)  . CH
           CHTEST  =  BREAK(*CH)
           CHLOC  =  BREAK(*CH)  . HEAD  LEN(1)
*
*
TEST       UNIVERSE  =  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
           VOWELS  =  UNION('A',UNION('E',UNION('I',UNION('O','U'))))
           OUTPUT  =  'VOWELS  =  '  VOWELS
           CONS  =  NEG(VOWELS)
           OUTPUT  =  'CONS  =  '  CONS
           WORD  =  'COMPILER'
           OUTPUT  =  'VOWELS IN ' WORD ' ARE:  "'
.                     INTER(WORD,VOWELS)  '"'
           OUTPUT  =  'CONSONANTS IN ' WORD ' ARE:  "'
.                     INTER(WORD,CONS)  '"'
END
```

Output from the program is:

```
VOWELS  =  AEIOU
CONS  =  BCDFGHJKLMNPQRSTVWXYZ
VOWELS IN "COMPILER" ARE:  "OIE"
CONSONANTS IN "COMPILER" ARE:  "CMPLR"
```

Example - Pseudo-Random Number Generator

A simple pseudo-random number generator, based on the power residue method of generation [4], is shown below.

```
        DEFINE('RANDOM(N)','RAN')                   :(RANEND)
RAN     RAN.VAR  =  RAN.VAR * 1061 + 3251
        RAN.VAR  RTAB(5)  =
        RANDOM  =  (RAN.VAR * N)  /  100000         :(RETURN)
RANEND
```

RANDOM(N) returns a value uniformly distributed over the integers 0,1,...,N-1 . The variable RAN.VAR is not local. On successive calls to RANDOM, the value of RAN.VAR cycles through all nonnegative integers less than 100,000. Thus, the initial value of RAN.VAR determines the output sequence from RANDOM.

When the random number generator is used in statistical experiments, such as games of chance, the player should have the opportunity to select the initial value of RAN.VAR . In some cases, the selection process should be unstable. It should be very difficult for a player to consistently initialize the random number generator with the same value.

The following definition of RANDOM was written specifically for the IBM System/360, and assumes that the user has access to the machine console while the program is running. A player can initialize RAN.VAR by flipping the Interval Timer Switch on the operator's console. TIME returns as value the number of milliseconds elapsed since the beginning of program execution. When the switch is on, as it normally is, the internal clock that TIME reads is running. When the switch is turned off, the clock stops but the program continues to run. Thus, with the switch on, successive calls of TIME return different values. With the switch off, successive calls of TIME return the same value.

```
        DEFINE('RANDOM(N)M','RAN1')                 :(GAME)
*
RAN1    RAN.VAR  =  TIME()
TIMON   M  =  LT(M,10)  M  +  1                     :S(TIMON)
        RAN.VAR  =  GT(TIME(),RAN.VAR)  TIME()      :F(TIMOFF)
        M  =  0                                     :(TIMON)
TIMOFF  EQ(RAN.VAR,TIME())                          :S(TIMOFF)
        RAN.VAR  =  TIME()
        RAN.VAR  RTAB(5)  =
        OUTPUT  =  'INITIAL VALUE OF RAN.VAR IS  '  RAN.VAR
        OUTPUT  =
        DEFINE('RANDOM(N)','RAN2')
*
RAN2    RAN.VAR  =  RAN.VAR * 1061 + 3251
        RAN.VAR  RTAB(5)  =
        RANDOM  =  (RAN.VAR * N) / 100000           :(RETURN)
*
GAME
```

In this example, the function RANDOM is defined twice. The first definition of RANDOM includes a local variable M and the entry point RAN1. The first call to RANDOM enters the definition at RAN1. Since the switch is on, the program enters the loop at TIMON and stays there because the predicate

GT(TIME(),RAN.VAR) always succeeds. If the switch is now turned off, the predicate fails, and control passes to the loop at TIMOFF. With the switch off, the predicate EQ(RAN.VAR,TIME()) always succeeds, causing a program loop at TIMOFF. When the switch is turned back on, RAN.VAR is truncated to 5 digits and the initial value printed. Thus, a flip of the switch initializes RAN.VAR.

Before computing the desired random number, RANDOM is redefined with entry point RAN2 so that subsequent calls to RANDOM do not go through the initialization process. A random number is then computed and returned.

The following program and output illustrate a statistical experiment utilizing RANDOM.

```
GAME     POINT  =  RANDOM(6) + RANDOM(6)  +  2
         NE(POINT,7)  NE(POINT,11)                     :F(NATURAL)
         NE(POINT,2)  NE(POINT,3)  NE(POINT,12)        :F(CRAPS)
         OUTPUT  =  'YOUR POINT IS   '  POINT
ROLL     ROLL  =  RANDOM(6) + RANDOM(6)  +  2
         EQ(POINT,ROLL)                                :S(MADE)
         NE(ROLL,7)  NE(ROLL,11)                       :F(LOSE)
         OUTPUT  =  '                '  ROLL  '   ROLL AGAIN.'
.                                                      :(ROLL)
NATURAL  OUTPUT  =  '                '  POINT  '   NATURAL,'
.                   ' YOU WIN.'                        :(UWIN)
MADE     OUTPUT  =  '                '  ROLL  '   MADE YOUR'
.                   ' POINT.'                          :(UWIN)
CRAPS    OUTPUT  =  '                '  POINT  '   CRAPS, YOU'
.                   ' LOSE.'                           :(ULOSE)
LOSE     OUTPUT  =  '                '  ROLL  '   TOO BAD.'
.                                                      :(ULOSE)
UWIN     WIN  =  WIN  +  1                             :(LIMIT)
ULOSE    LOSE  =  LOSE  +  1                           :(LIMIT)
LIMIT    OUTPUT  =  LT(WIN + LOSE,100)                 :F(PAY)S(GAME)
PAY      OUTPUT  =
         OUTPUT  =  'YOU LOSE  '  GT(LOSE,WIN)  LOSE - WIN
.                   '  DOLLARS.'                       :S(END)
         OUTPUT  =  'YOU WIN  '  GT(WIN,LOSE)  WIN - LOSE
.                   '  DOLLARS.'                       :S(END)
         OUTPUT  =  'YOU BREAK EVEN.'                  :(END)
END
```

```
INITIAL VALUE OF RAN.VAR IS  3877

                 7   NATURAL, YOU WIN.

                 7   NATURAL, YOU WIN.

YOUR POINT IS    9
                 6   ROLL AGAIN.
                 5   ROLL AGAIN.
                 12   ROLL AGAIN.
                 2   ROLL AGAIN.
                 7   TOO BAD.

YOUR POINT IS    9
                 5   ROLL AGAIN.
                 7   TOO BAD.
```

```
YOUR POINT IS   6
                10   ROLL AGAIN.
                8   ROLL AGAIN.
                9   ROLL AGAIN.
                8   ROLL AGAIN.
                3   ROLL AGAIN.
                6   MADE YOUR POINT.

YOUR POINT IS   4
                4   MADE YOUR POINT.

YOUR POINT IS   4
                7   TOO BAD.

         .

         .

         .

YOUR POINT IS   4
                8   ROLL AGAIN.
                8   ROLL AGAIN.
                5   ROLL AGAIN.
                8   ROLL AGAIN.
                9   ROLL AGAIN.
                7   TOO BAD.
                7   NATURAL, YOU WIN.

YOU BREAK EVEN.
```

E. Recursive Functions

Many functions are conveniently defined recursively. For example, factorials may be defined as

```
      fact(0)  =  1
      fact(n)  =  n*fact(n-1)    for  n > 0
```

Using Pascal's triangle, a recursive definition for the binomial coefficients is easily deduced.

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

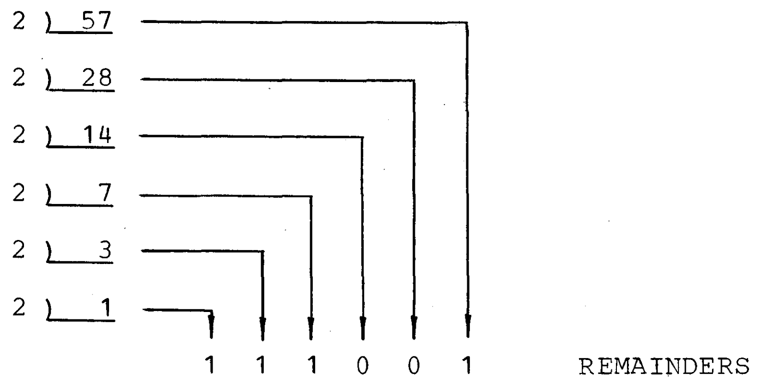
```
binc(n,0)   =  1
binc(n,n)   =  1
binc(n,k)   =  binc(n-1,k-1)+binc(n-1,k)    0 < k < n
```

A recursive procedure has the property that the function itself is called in the procedure. While convenient, recursive procedures may lead to computational inefficiencies. Nevertheless, recursion is frequently the most natural way of expressing a function, and may considerably simplify programming.

Programmer-defined functions in SNOBOL4 may be recursive. Since values of the function name, arguments, and local variables are all saved when a function is called, a procedure can include recursive coding.

Example - Decimal to Binary Conversion

The next program converts decimal integers to their binary representation by successive divisions. For example, to compute the binary representation of 57, it is repeatedly divided by 2 and the remainders are concatenated.



$57_{10} = 111001_2$

The binary representation of 57 is the binary representation of 28 (11100_2) followed by the remainder of 57/2. A recursive definition of the process is

```
binary(57)  =  binary(28)  remainder(57/2)
```

where concatenation is implied.

In SNOBOL4, the results of integer division are truncated. Thus,

```
        57 / 2  is  28
```

The remainder of any integer division N / M is

```
        N  -  (N / M)  *  M
```

Thus, the recursive definition can be written in the more general form

```
        binary(n)  =  binary(n/2)  n-(n/2)*2    for  n > 1
```

with the terminal cases

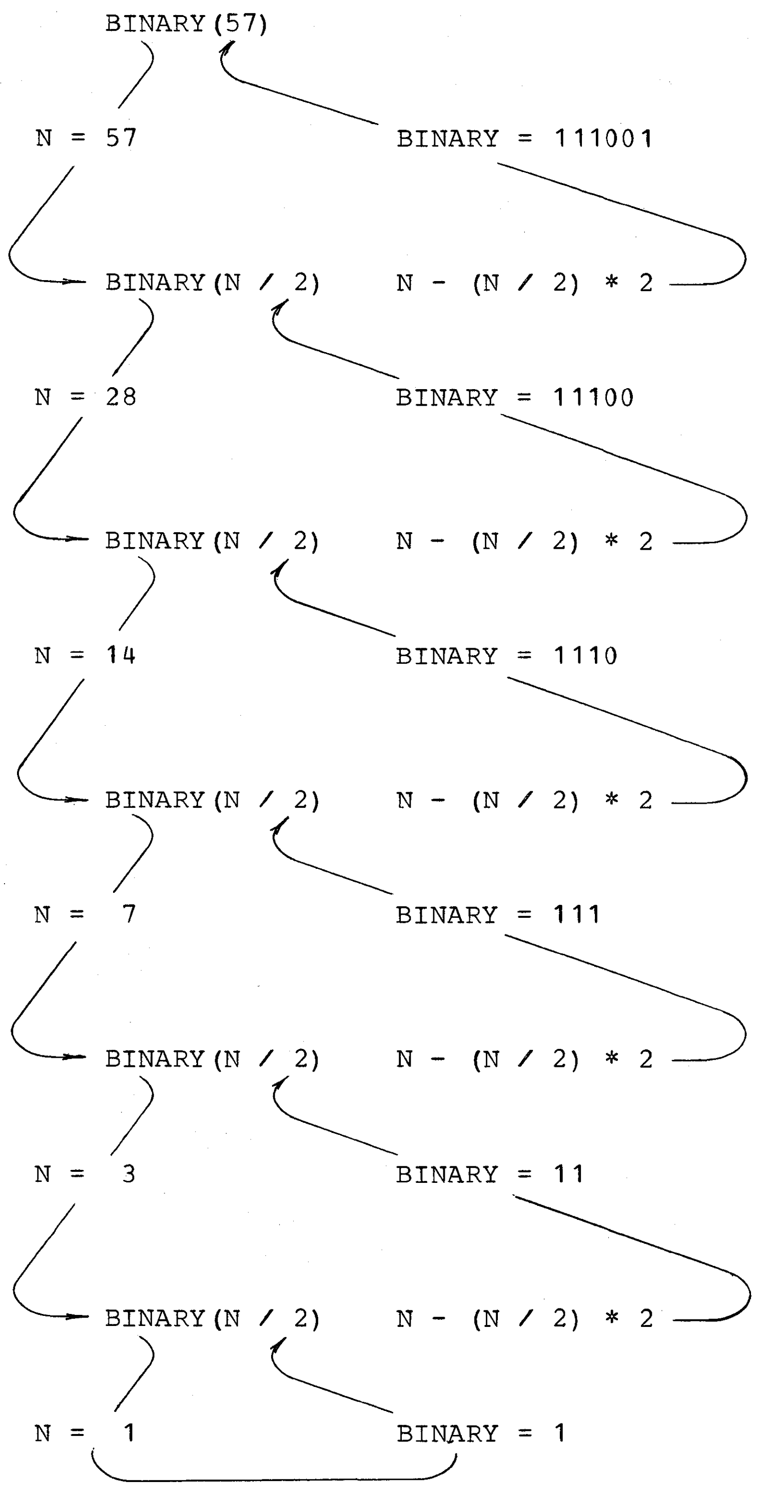
```
        binary(1)  =  1
        binary(0)  =  0
```

A procedure for BINARY is

```
         DEFINE('BINARY(N)')                      :(BINEND)
*
BINARY   BINARY  =  GT(N,1)  BINARY(N / 2)  N - (N / 2) * 2
.                                                 :S(RETURN)
         BINARY  =  N                             :(RETURN)
BINEND
```

On entry to BINARY, the value of N is tested by the predicate GT(N,1) which fails for the two terminal cases N = 0 and N = 1. If either of these cases is true, the first statement fails and N is returned as the value of BINARY. If N is greater than 1, a recursive call is made to BINARY with N / 2 as the argument. The value of BINARY(N / 2) then has the remainder of N / 2 concatenated with it, to get the final value of BINARY(N).

The following diagram illustrates the recursive calls made during evaluation of BINARY(57). The recursion plunges six levels before reaching the terminal case of N = 1. On returning, the value of BINARY evolves.



It is important to notice the necessity of preserving values before a function call and restoring them upon completion. At the first level down, BINARY(28) is called with N having value 57. During the course of evaluating BINARY(28), N takes on values 28, 14, 7, 3, and 1. Following evaluation of BINARY(28), N must once again have the value 57 in order to compute the remainder of 57 / 2.

An improvement is possible in the definition of BINARY. SNOBOL4 permits use of a function name as one of the formal arguments in a function definition. Thus,

```
          DEFINE('BINARY(BINARY)')
```

is a valid statement. The procedure of BINARY can be rewritten substituting BINARY for N.

```
BINARY    BINARY  =  GT(BINARY,1)  BINARY(BINARY / 2)
.                    BINARY - (BINARY / 2) * 2      :(RETURN)
```

The second statement would become

```
          BINARY  =  BINARY
```

which is redundant. For the terminal cases recognized by the failure of GT(BINARY,1), BINARY has the proper value, 0 or 1, and an unconditional RETURN is made.

```
          DEFINE('BINARY(BINARY)')
*
          OUTPUT  =  '    0  =  '  BINARY(0)
          OUTPUT  =  '   13  =  '  BINARY(13)
          OUTPUT  =  '   57  =  '  BINARY(57)
          OUTPUT  =  '  472  =  '  BINARY(472)
          OUTPUT  =  ' 8192  =  '  BINARY(8192)
          OUTPUT  =  '13279  =  '  BINARY(13279)
          OUTPUT  =  '99999  =  '  BINARY(99999)     :(END)
*
BINARY    BINARY  =  GT(BINARY,1)  BINARY(BINARY / 2)
.                    BINARY - (BINARY / 2) * 2      :(RETURN)
END
```

```
    0  =  0
   13  =  1101
   57  =  111001
  472  =  111011000
 8192  =  10000000000000
13279  =  11001111011111
99999  =  11000011010011111
```

Example - Polish to Infix Translation

Arithmetic expressions such as

```
          X + Y
        A / B / C
    V - W - X + Y * Z
```

are written using an infix notation. They can also be written in Polish prefix notation [5,6], resembling conventional functional notation. Here the binary operators appear to the left of their arguments. Prefix notation for the expressions is

```
          +(X,Y)
        /(/(A,B),C)
    +(-(-(V,W),X),*(Y,Z))
```

Conversion from Polish prefix form to infix form, and vice versa, can be performed using recursive programmer-defined functions. The first of the two programs to follow converts strings from Polish to infix form. The recursive rules for specifying the function INF are:

1. If the argument to INF is a simple variable, then

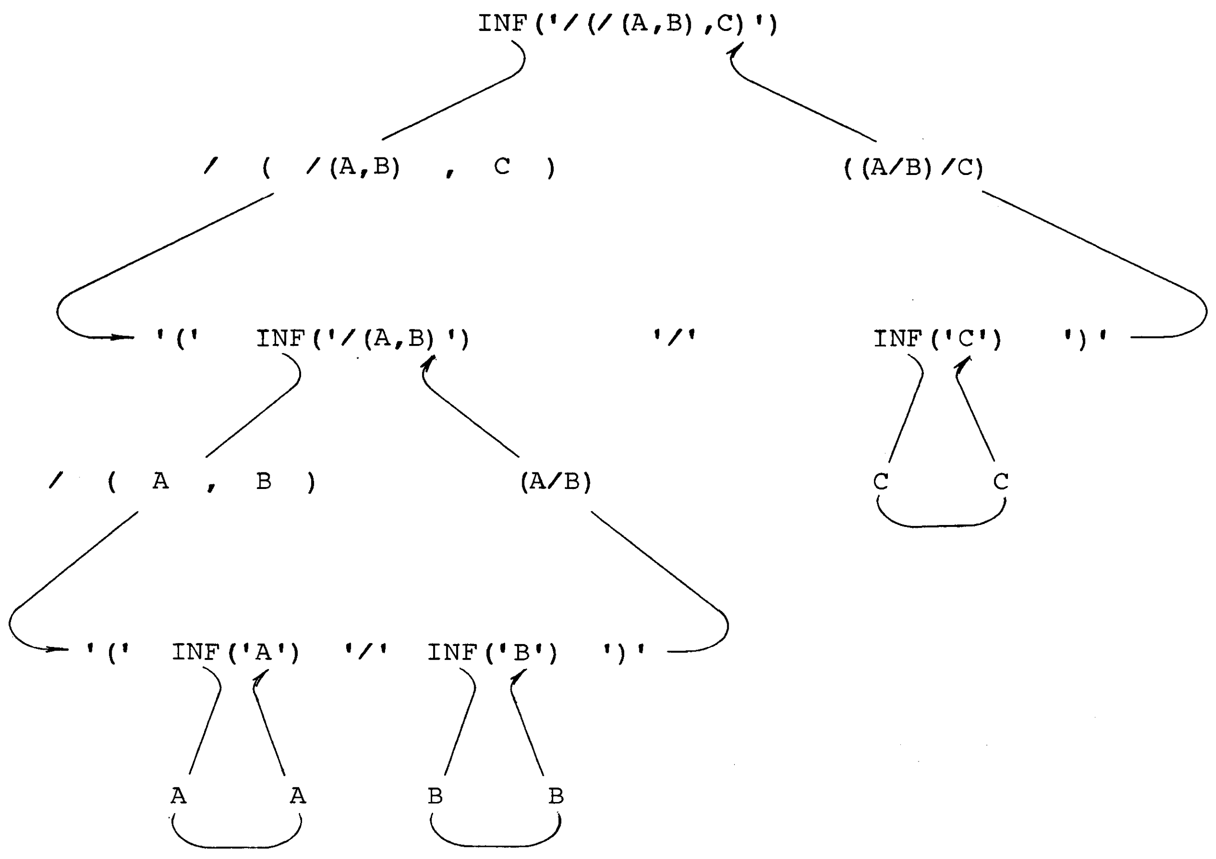
   INF(VAR) = VAR

2. If the argument to INF is a Polish expression of the form OP(EX1,EX2), then

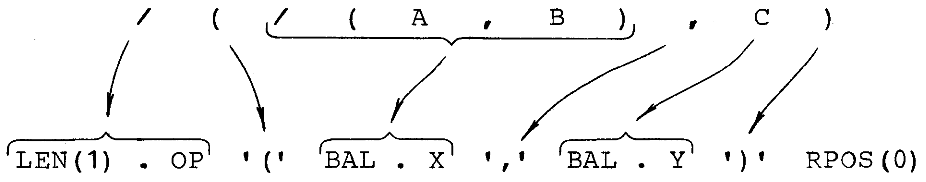
   INF(OP(EX1,EX2)) = (INF(EX1) OP INF(EX2))

The conversion consists of finding the operator and its two arguments, which may be expressions. The operator is inserted between its two arguments and parentheses are placed around the resulting expression. Of course, the arguments are still in Polish form, so each must be converted to infix by a recursive call of INF .

The following diagram depicts the conversion of /(/(A,B),C) to ((A/B)/C) .



In the program to follow, the procedure for INF consists of one line. The pattern INPAT is used to break a Polish expression into an operator and two arguments.



If INPAT matches INF, it matches the entire string, which is then rearranged into infix notation. If INPAT fails to match, INF must be a variable and is returned unchanged as value.

```
        &ANCHOR  =  1
        INPAT   =   LEN(1)  . OP   '('  BAL . X   ','  BAL . Y   ')'
.                   RPOS(0)
*
*
        DEFINE('INF(INF)X,Y,OP')
*
*
        PADPAT   =  LEN(*(40 - SIZE(STRING)))  . PAD
        BLANKS   =  '                                          '
LOOP    STRING   =  TRIM(INPUT)                       :F(END)
        BLANKS  PADPAT
        OUTPUT   =  STRING   PAD   INF(STRING)        :(LOOP)
*
*
INF     INF  INPAT   =  '('   INF(X)   OP   INF(Y)   ')'
.                                                     :(RETURN)
END
```

Output from the program follows. The Polish prefix form of the input is shown on the left, and the infix form appears on the right.

```
-(*(A,+(B,C)),/(D,E))                    ((A*(B+C))-(D/E))
-(-(-(-(-(A,B),C),D),E),*(F,G))          (((((A-B)-C)-D)-E)-(F*G))
-(+(ALPHA,*(BETA,GAMMA)),/(DELTA,PI))    ((ALPHA+(BETA*GAMMA))-(DELTA/PI))
```

Example - Infix to Polish Translation

Conversion of arithmetic expressions from infix to Polish form is harder than the converse. A function POL which performs the conversion is of the form:

```
        POL(EX1 OP EX2)  =  OP '(' POL(EX1) ',' POL(EX2) ')'
```

Ambiguities can arise when attempting to separate an unparenthesized expression into two expressions and an operator. For example, the expression

```
        A - B * C - D
```

can be separated many ways, including

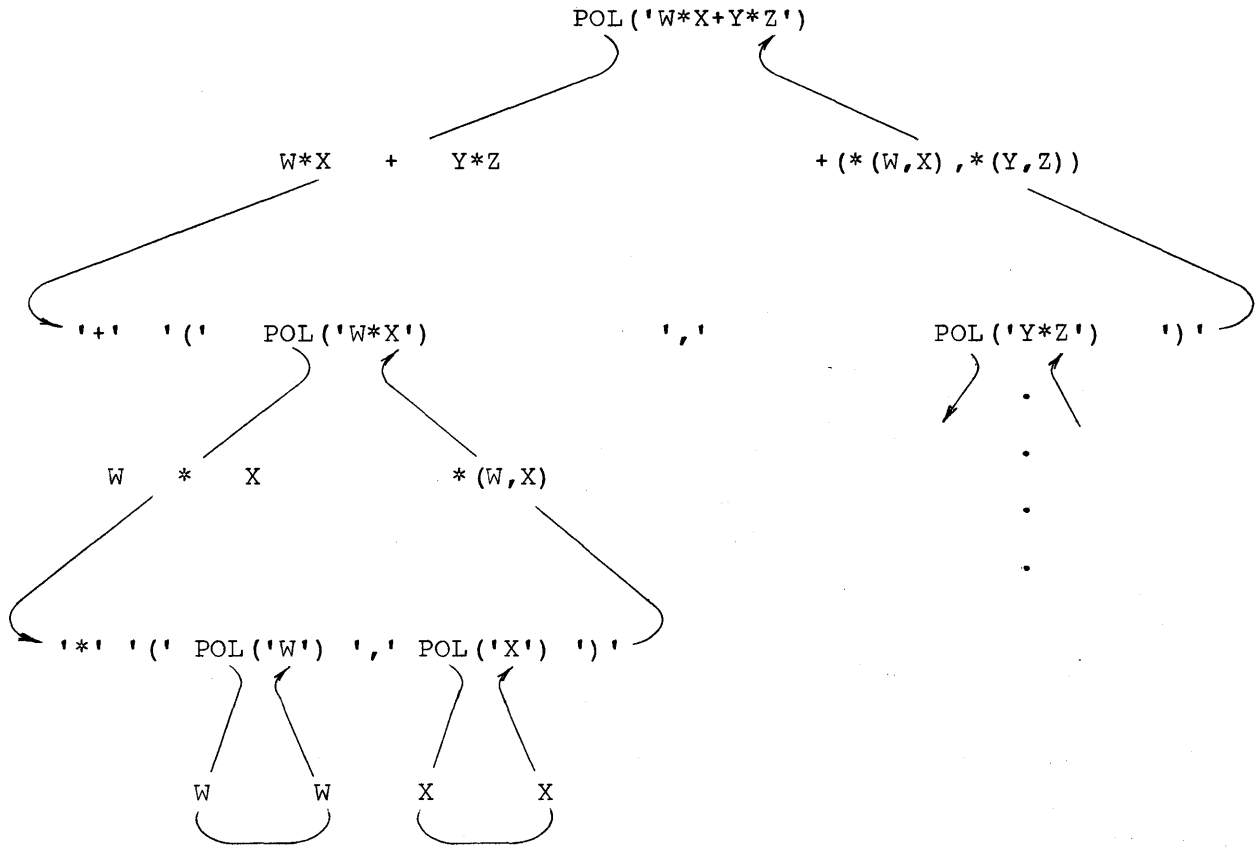
```
        A - (B * C - D)

        (A - B) * (C - D)

        (A - B * C) - D
```

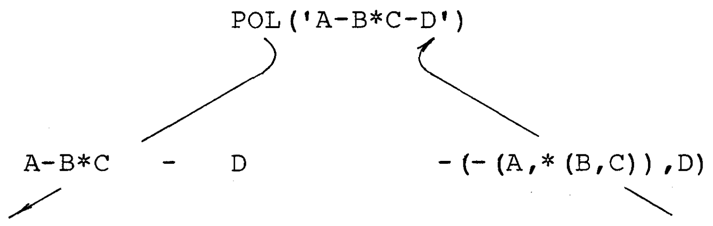
Normal conventions for the precedence and association of operators require that multiplication and division have precedence over addition and subtraction and that operators associate to the left. Thus, of the three choices above, the first is incorrect because subtraction associates to the right, the second is incorrect because subtraction is given higher precedence than multiplication, and the third is correct. The expression (A - B * C) must be parenthesized as (A - (B * C)) to conform to the conventions.

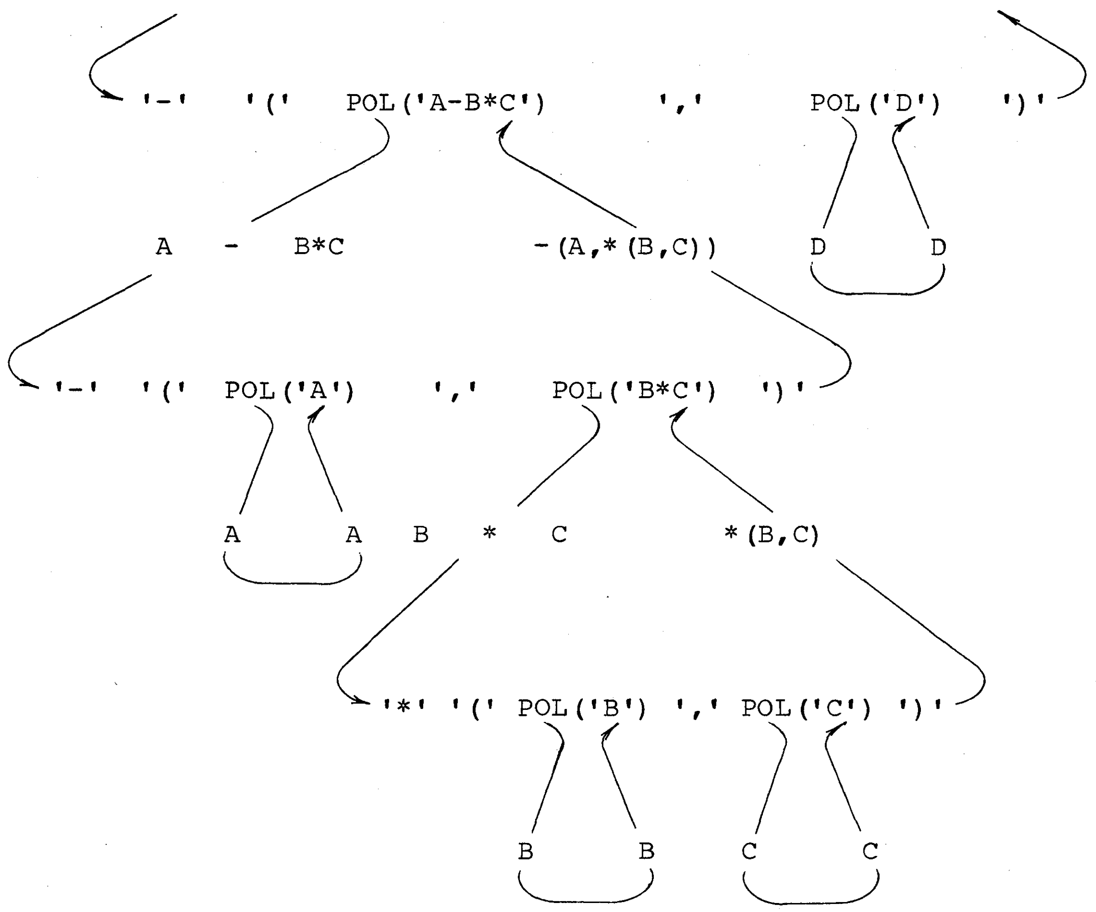
In defining the function POL, the precedence of multiplicative over additive operators can be assured by dealing with the additive operators first. For example:



POL('W*X+Y*Z')
W*X + Y*Z
+(*(W,X),*(Y,Z))
'+' '(' POL('W*X') ',' POL('Y*Z') ')'
W * X
*(W,X)
'*' '(' POL('W') ',' POL('X') ')'
W W X X

Left association of operators is assured by selecting the rightmost operator in a string of operators having equal precedence. For example




'-' '(' POL('A-B*C') ',' POL('D') ')'
A - B*C
-(A,*(B,C))
D D
'-' '(' POL('A') ',' POL('B*C') ')'
A A B * C
*(B,C)
'*' '(' POL('B') ',' POL('C') ')'
B B C C

Thus, the rules prescribing the behavior of POL are:

1. Remove any enclosing parentheses from the infix string.

2. If possible, separate the argument into two expressions which are balanced with respect to parentheses and separated by the rightmost additive operator. The value of POL then becomes

```
OP '(' POL(EX1) ',' POL(EX2) ')'
```

   If this is not possible, perform Step 3.

3. If possible, separate the argument into two expressions balanced with respect to parentheses and separated by the rightmost multiplicative operator. The value of POL then becomes

```
OP '(' POL(EX1) ',' POL(EX2) ')'
```

   If this is not possible, perform Step 4.

4. The infix string must be a simple variable, which becomes the value of POL.

A complete program for infix-to-Polish conversion and test results follow.

```
        &ANCHOR  =  1
        PMPAT    =  (ARBNO(BAL ANY('+-')) $ X FAIL  |  *DIFFER(X)
.                   TAB(*(SIZE(X) - 1))) . X  LEN(1) . OP  REM . Y
        MDPAT    =  (ARBNO(BAL ANY('*/')) $ X FAIL  |  *DIFFER(X)
.                   TAB(*(SIZE(X) - 1))) . X  LEN(1) . OP  REM . Y
        STRIP    =  '('  BAL . POL  ')'  RPOS(0)
*
*
        DEFINE('POL(POL)X,Y,OP')
*
*
        PADPAT   =  LEN(*(40 - SIZE(STRING))) . PAD
        BLANKS   =  '                                        '
LOOP    STRING   =  TRIM(INPUT)                        :F(END)
        BLANKS  PADPAT
        OUTPUT   =  STRING  PAD  POL(STRING)           :(LOOP)
*
*
POL     POL  STRIP                                     :S(POL)
        POL  PMPAT  =  OP  '('  POL(X)  ','  POL(Y)  ')'
.                                                      :S(RETURN)
        POL  MDPAT  =  OP  '('  POL(X)  ','  POL(Y)  ')'
.                                                      :(RETURN)
END
```

```
((A*(B+C))-(D/E))                          -(*(A,+(B,C)),/(D,E))
A-B-C-D-E-F*G                              -(-(-(-(-(A,B),C),D),E),*(F,G))
((ALPHA+(BETA*GAMMA))-(DELTA/PI))          -(+(ALPHA,*(BETA,GAMMA)),/(DELTA,PI))
```

The pattern STRIP removes the outer parentheses from the infix expression. The patterns PMPAT and MDPAT separate the infix expression into two expressions and an operator according to the convention for left association. The patterns are identical except that PMPAT looks for addition or subtraction and MDPAT looks for multiplication or division.

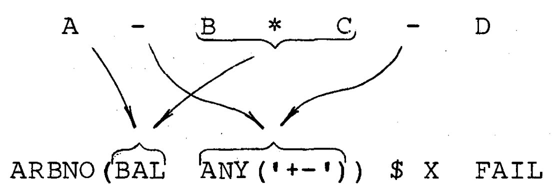
PMPAT has three parts, corresponding to the first balanced expression, the operator, and the second balanced expression. The pattern for the first expression is complicated by the fact that the operator must be the rightmost in the string of operators. Consider the pattern for the first expression:

```
        (ARBNO(BAL ANY('+-')) $ X  FAIL  |  *DIFFER(X)
.       TAB(*(SIZE(X) - 1))) . X
```

It consists of two alternatives. The first,

```
        ARBNO(BAL ANY('+-')) $ X  FAIL
```

is used to locate the rightmost operator by matching a sequence of balanced strings followed by additive operators. FAIL forces ARBNO to match the longest such string and eventually causes failure of the alternative. Thus, for the expression A-B*C-D , the last match of the first alternative is



On entry to the second alternative

```
*DIFFER(X)  TAB(*(SIZE(X) - 1))
```

the value of X is checked to see if it is the null string. If so, no match is possible. If it is not null, the first balanced expression must be all but the last character of X. The first expression is matched by

```
TAB(*(SIZE(X) - 1))
```

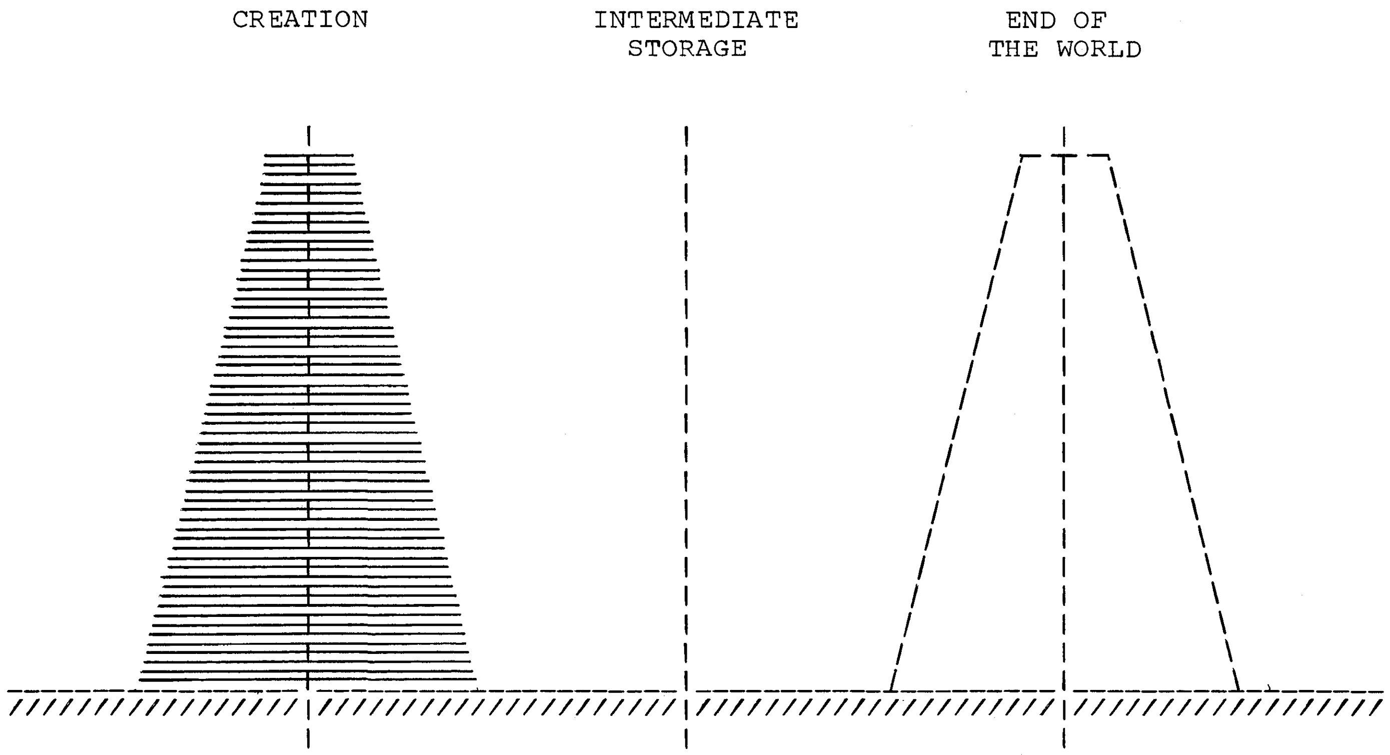
The remainder of PMPAT consists of the expression

```
LEN(1) . OP  REM . Y
```

LEN(1) is used to match the operator and REM matches the remainder of the string which is the second balanced expression.

Example - Tower of Hanoi

The Tower of Hanoi is a game derived from the ancient Tower of Brahma, a ritual allegedly practiced by Brahman priests to predict the end of the world. At the time of creation, 64 golden discs of decreasing size appeared stacked on a diamond needle. Nearby were two other diamond needles, both empty. The Brahman priests, created at the same time, were set to the task of moving the discs from their original needle to a second needle using, when necessary, the third needle as temporary storage. Before all 64 discs are moved to the second needle and stacked in decreasing size, the end of the world will be upon us.



Movement of the discs is governed by the rules:

1) only one disc may be moved at a time,
2) a disc may be moved from any needle to any other, and
3) at no time may a larger disc rest upon a smaller disc.

A solution to the Tower of Hanoi is a recursive function which prints out the steps necessary to move N discs from one needle to another (where N is hopefully a good deal smaller than 64). A program that defines the function HANOI and tests it by moving 5 discs from needle A to needle C follows.

```
         DEFINE('HANOI(N,NS,ND,NI)')                   :(HANOI.END)
*
HANOI    EQ(N,0)                                       :S(RETURN)
         HANOI(N - 1,NS,NI,ND)
         OUTPUT  =  'MOVE DISC  ' N '  FROM  ' NS ' TO ' ND
         HANOI(N - 1,NI,ND,NS)                         :(RETURN)
HANOI.END
*
TEST     HANOI(5,'A','C','B')
END
```

```
MOVE DISC  1  FROM  A TO C
MOVE DISC  2  FROM  A TO B
MOVE DISC  1  FROM  C TO B
MOVE DISC  3  FROM  A TO C
MOVE DISC  1  FROM  B TO A
MOVE DISC  2  FROM  B TO C
MOVE DISC  1  FROM  A TO C
MOVE DISC  4  FROM  A TO B
MOVE DISC  1  FROM  C TO B
MOVE DISC  2  FROM  C TO A
MOVE DISC  1  FROM  B TO A
MOVE DISC  3  FROM  C TO B
MOVE DISC  1  FROM  A TO C
MOVE DISC  2  FROM  A TO B
MOVE DISC  1  FROM  C TO B
MOVE DISC  5  FROM  A TO C
MOVE DISC  1  FROM  B TO A
MOVE DISC  2  FROM  B TO C
MOVE DISC  1  FROM  A TO C
MOVE DISC  3  FROM  B TO A
MOVE DISC  1  FROM  C TO B
MOVE DISC  2  FROM  C TO A
MOVE DISC  1  FROM  B TO A
MOVE DISC  4  FROM  B TO C
MOVE DISC  1  FROM  A TO C
MOVE DISC  2  FROM  A TO B
MOVE DISC  1  FROM  C TO B
MOVE DISC  3  FROM  A TO C
MOVE DISC  1  FROM  B TO A
MOVE DISC  2  FROM  B TO C
MOVE DISC  1  FROM  A TO C
```

The program logic can be seen by induction. Clearly, moving no discs requires no steps. Moving one disc from needle A to needle C requires one step.

```
MOVE DISC  1  FROM  A TO C
```

Moving two discs from A to C requires three steps.

```
MOVE DISC  1  FROM  A TO B
MOVE DISC  2  FROM  A TO C
MOVE DISC  1  FROM  B TO C
```

Moving three discs from A to C requires seven steps.

```
MOVE DISC  1  FROM  A TO C
MOVE DISC  2  FROM  A TO B
MOVE DISC  1  FROM  C TO B
MOVE DISC  3  FROM  A TO C
MOVE DISC  1  FROM  B TO A
MOVE DISC  2  FROM  B TO C
MOVE DISC  1  FROM  A TO C
```

The general solution is:

```
        MOVE N-1 DISCS FROM  A TO B
        MOVE DISC  N  FROM  A TO C
        MOVE N-1 DISCS  FROM  B TO C
```

The implementation is simple. HANOI is defined with four arguments:

1) N is the number of discs to be moved,
2) NS is the starting needle,
3) ND is the destination needle, and
4) NI is the intermediate storage needle.

On entry to HANOI, the value of N is compared with zero. If N is zero, no discs are moved and the function returns. If N is not zero, HANOI is called recursively to move N-1 discs from the starting needle to the intermediate storage needle. Having done that, the command to move the Nth disc from the starting needle to the destination needle is printed. Finally, HANOI is called a second time to move the N-1 discs from intermediate storage to the destination needle.

F. OPSYN

It is sometimes convenient to provide synonyms for existing functions. The primitive function OPSYN can be used for this purpose. The general format of OPSYN is

```
        OPSYN(new,old)
```

For example,

```
        OPSYN('SAME','IDENT')
```

defines SAME to be a synonym for the function name IDENT.

A call using a synonym for a primitive function must have the correct number of arguments. Trailing arguments may not be omitted. For example,

```
     SAME(X)
```

causes error termination.

Consider a program using the pattern BIGP of Chapter 2.

```
        BIGP  =  (*P $ TRY  *GT(SIZE(TRY),SIZE(BIG))) $ BIG  FAIL
```

This program prints the values of TRY and BIG, whose sizes are compared by GT. The printing can be done by providing a new programmer-defined function for GT. However, since GT must still be used, it is OPSYNed to another function name, GTHAN.

```
         OPSYN('GTHAN','GT')
         DEFINE('GT(X,Y)')                                 :(TEST)
*
GT       OUTPUT  =  'TRY = ' TRY ',    BIG = '  BIG
         GTHAN(X,Y)                                        :S(RETURN)F(FRETURN)
*
TEST     BIGP  =   (*P $ TRY  *GT(SIZE(TRY),SIZE(BIG))) $ BIG  FAIL
         STR  =  'IN 1964 NFL ATTENDANCE JUMPED TO 4,807,884; '
.                'AN INCREASE OF 401,810.'
         P  =  SPAN('0123456789,')
         BIG  =
         STR  BIGP
         P  =  SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
         BIG  =
         STR  BIGP
END
```

```
TRY = 1964,   BIG =
TRY = 964,   BIG = 1964
TRY = 64,   BIG = 1964
TRY = 4,   BIG = 1964
TRY = 4,807,884,   BIG = 1964
TRY = ,807,884,   BIG = 4,807,884
TRY = 807,884,   BIG = 4,807,884
TRY = 07,884,   BIG = 4,807,884
TRY = 7,884,   BIG = 4,807,884
TRY = ,884,   BIG = 4,807,884
TRY = 884,   BIG = 4,807,884
TRY = 84,   BIG = 4,807,884
TRY = 4,   BIG = 4,807,884
TRY = 401,810,   BIG = 4,807,884
TRY = 01,810,   BIG = 4,807,884
TRY = 1,810,   BIG = 4,807,884
TRY = ,810,   BIG = 4,807,884
TRY = 810,   BIG = 4,807,884
TRY = 10,   BIG = 4,807,884
TRY = 0,   BIG = 4,807,884
TRY = IN,   BIG =
TRY = N,   BIG = IN
TRY = NFL,   BIG = IN
TRY = FL,   BIG = NFL
TRY = L,   BIG = NFL
TRY = ATTENDANCE,   BIG = NFL
TRY = TTENDANCE,   BIG = ATTENDANCE
TRY = TENDANCE,   BIG = ATTENDANCE
TRY = ENDANCE,   BIG = ATTENDANCE
TRY = NDANCE,   BIG = ATTENDANCE
TRY = DANCE,   BIG = ATTENDANCE
TRY = ANCE,   BIG = ATTENDANCE
TRY = NCE,   BIG = ATTENDANCE
TRY = CE,   BIG = ATTENDANCE
TRY = E,   BIG = ATTENDANCE
TRY = JUMPED,   BIG = ATTENDANCE
TRY = UMPED,   BIG = ATTENDANCE
TRY = MPED,   BIG = ATTENDANCE
TRY = PED,   BIG = ATTENDANCE
TRY = ED,   BIG = ATTENDANCE
TRY = D,   BIG = ATTENDANCE
TRY = TO,   BIG = ATTENDANCE
TRY = O,   BIG = ATTENDANCE
TRY = AN,   BIG = ATTENDANCE
TRY = N,   BIG = ATTENDANCE
TRY = INCREASE,   BIG = ATTENDANCE
TRY = NCREASE,   BIG = ATTENDANCE
TRY = CREASE,   BIG = ATTENDANCE
TRY = REASE,   BIG = ATTENDANCE
TRY = EASE,   BIG = ATTENDANCE
TRY = ASE,   BIG = ATTENDANCE
TRY = SE,   BIG = ATTENDANCE
TRY = E,   BIG = ATTENDANCE
TRY = OF,   BIG = ATTENDANCE
```

G. APPLY

APPLY is a primitive function that creates and executes a function call. APPLY(f,a_1,...,a_n) calls the function f with the arguments a_1,...,a_n. The value of APPLY is the value returned by the function it calls. The function f may be a primitive function or a programmer-defined function. Like OPSYN, a use of APPLY on a primitive function must specify the correct number of arguments.

An important use of APPLY is to call various functions depending on the current value of data. Execution of the statements

```
        X  =  'SIZE'
        Y  =  57
        OUTPUT  =  APPLY(X,Y)
```

calls SIZE(57) and prints 2 . Execution of

```
        X  =  'BINARY'
        Y  =  57
        OUTPUT  =  APPLY(X,Y)
```

calls BINARY(57), defined earlier, and prints 111001 .

Chapter 5. Arrays, Data Types, and Keywords

A. Arrays

An array is an indexed aggregate of variables. Arrays are created by the execution of the primitive function ARRAY. ARRAY(p,e) returns an array whose bounds and dimensions are described by the prototype p . Every element is initialized to the value of the expression e . For example,

```
          VECTOR   =   ARRAY(10)
```

assigns a one-dimensional array of length 10 to VECTOR . Since the second argument is omitted, each element of the array has the null string value. Indexing ordinarily starts at 1. Other lower bounds may be specified by using a colon to separate the upper and lower limits.

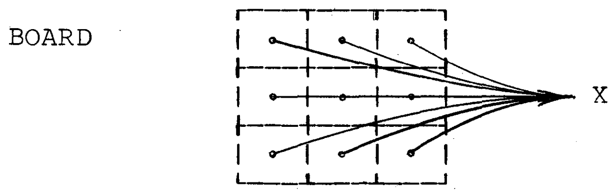
```
          LINE   =   ARRAY('-5:5')
```

creates an array with lower bound -5 and upper bound 5.

Additional dimensions in a prototype are separated by commas. Thus,

```
          BOARD   =   ARRAY('3,3','X')
```

defines a three-by-three array with all elements having the value X .

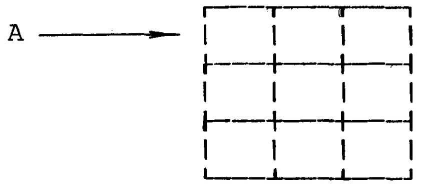


There is no intrinsic limit on the size or dimensionality of an array.

Warning: The first argument of ARRAY is the prototype, and the second is a value which is given to each element of the resulting array. Thus,

```
          A   =   ARRAY('3,3')
```

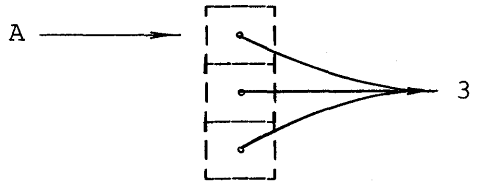
creates a two-dimensional array with each element having the null string as value.



On the other hand,

```
          A    =    ARRAY(3,3)
```

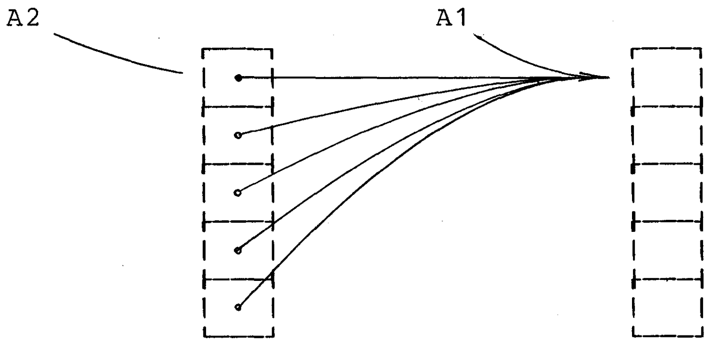
creates a one-dimensional array with each element having the value 3.



Each element of an array is given the same value. Consequently, execution of the instructions

```
          A1    =    ARRAY(5)
          A2    =    ARRAY(5,A1)
```

creates only two arrays. Each element of A2 has the same array, A1, as value.



1. Array References

If the value of a variable is an array, as is the case with VECTOR, BOARD, A, A1, and A2 above, an element in the array may be referenced through the variable. Angular brackets following the array-valued variable are used to specify the element. Array references such as VECTOR<8> or BOARD<2,3> , are variables. For example,

```
          VECTOR<8>    =    EXP
```

assigns the value of EXP to the eighth element of VECTOR.

```
          OUTPUT    =    BOARD<2,3>
```

prints the value of the (2,3)-element of BOARD.

```
          FIELD    =    BREAK(' ') . LINE<-3,4> ' '
```

defines a pattern that breaks out a field of data and assigns it to the (-3,4)-element of LINE.

Each element of an array may have any type of data object as value. There is no requirement that all elements of an array have the same data type. For example, the first element of an array may be an integer, the second a pattern, and so forth.

If an index referring to an element of an array falls outside the range of the array, the array reference fails. Thus,

```
          OUTPUT    =    VECTOR<12>
```

fails. This failure may be used to control iteration through the elements of an array without knowing its size. A function SUM, whose value is the sum of all the elements of an array, could have the defining statement

```
          DEFINE('SUM(ARRAY)N')
```

with the procedure

```
SUM       N    =    N + 1
          SUM    =    SUM + ARRAY<N>      :S(SUM)F(RETURN)
```

The summation loop continues until N exceeds the range of ARRAY. This function does not need to know the size of ARRAY, but only that it is a one-dimensional array with a lower bound of one.

Example - Bubble Sort

A simple application of one-dimensional arrays is illustrated in the following example which puts strings in lexical order. A bubble sort is much like an exchange sort. When two elements are found to be out of order, they are switched. However, the lexically smaller item is bubbled up to its proper place.

```
*                     BUBBLE SORT PROGRAM
*
          DEFINE('SORT(N)I')
          DEFINE('SWITCH(I)TEMP')
          DEFINE('BUBBLE(J)')
*
*                     GET NUMBER OF ITEMS TO BE SORTED
*
          N           =  TRIM(INPUT)                :F(ERROR)
          A           =  ARRAY(N)
*
*                     READ IN THE ITEMS
*
READ      I           =  I + 1
          A<I>        =  TRIM(INPUT)                :F(GO)S(READ)
*
*                     SORT THE LIST
*
GO        SORT(N)
*
*                     PRINT SORTED LIST
*
          M           =  1
PRINT     OUTPUT      =  A<M>                       :F(END)
          M           =  M + 1                      :(PRINT)
*
*                     FUNCTIONS
*
SORT      I           =  LT(I,N - 1)  I + 1         :F(RETURN)
          LGT(A<I>,A<I + 1>)                        :F(SORT)
          SWITCH(I)
          BUBBLE(I)                                 :(SORT)
*
SWITCH    TEMP        =  A<I>
          A<I>        =  A<I + 1>
          A<I + 1>    =  TEMP                       :(RETURN)
*
BUBBLE    J           =  GT(J,1)  J - 1             :F(RETURN)
          LGT(A<J>,A<J + 1>)                        :F(RETURN)
          SWITCH(J)                                 :(BUBBLE)
*
END
```

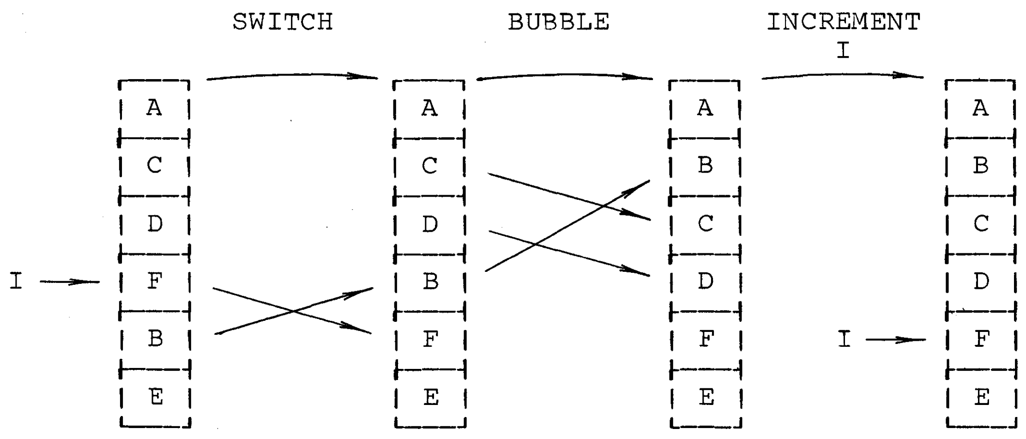
For the input

```
15
ADDSIB
BUKINT
ADJTTL
BUCKET
ADREAL
BKSPCE
APDSP
ARRAY
BKSIZE
ALTERN
BRANCH
ADJUST
BUFFER
ADDSON
ADDLG
```

the output is

```
ADDLG
ADDSIB
ADDSON
ADJTTL
ADJUST
ADREAL
ALTERN
APDSP
ARRAY
BKSIZE
BKSPCE
BRANCH
BUCKET
BUFFER
BUKINT
```

One iteration of SORT is:



Elements above I are properly ordered. If elements at I and I + 1 are out of order, they are switched. The new element at I (B) is bubbled by means of interchanges to its proper place above I. I is incremented and the process continues.

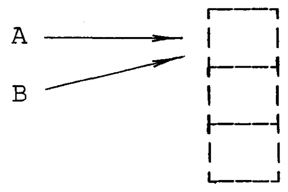
2. Primitive Functions for Use with Arrays

COPY

The value of the ARRAY function is an object whose data type is ARRAY. This value may be assigned to one or more variables.

```
A    =    ARRAY(3)
B    =    A
```

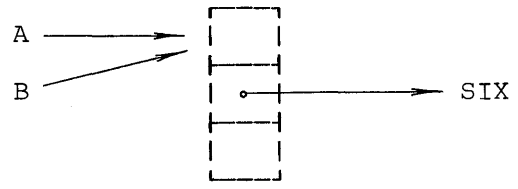
A and B have the same array as value.



Thus,

```
B<2>     =    'SIX'
OUTPUT    =    A<2>
```

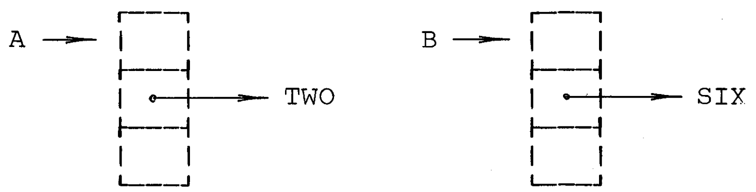
print SIX .



The COPY function produces a copy of an array. Executing the statements

```
A     =    ARRAY(3)
A<2>    =    'TWO'
B     =    COPY(A)
B<2>    =    'SIX'
```

creates distinct arrays. Unlike the previous example, assigning a value to B<2> does not affect the value of A<2>.



COPY may be used with other types of data, as illustrated in the section on data types.

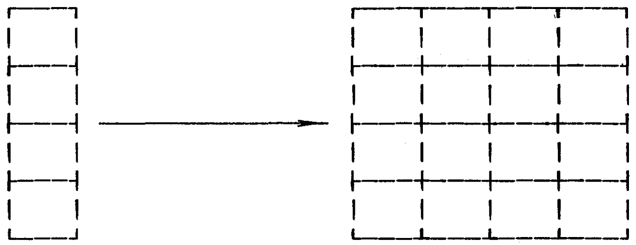
PROTOTYPE

The value of the dimension or range of an array is sometimes needed. The primitive function PROTOTYPE is used to obtain the prototype used to define the array. PROTOTYPE has an array-valued argument and returns the prototype string. Thus, if

```
          A    =    ARRAY('-5:5','X')
```

then the value of PROTOTYPE(A) is the string -5:5 .

An example utilizing PROTOTYPE is the following function named SQUARE. The argument of SQUARE is any singly-dimensioned array. The value of SQUARE is a two-dimensional square array whose dimensions equal that of the argument, and whose elements are null strings.



```
          DEFINE('SQUARE(A)')                          :(SQEND)
*
SQUARE    SQUARE  =  ARRAY(PROTOTYPE(A) ',' PROTOTYPE(A))    :(RETURN)
SQEND
```

The argument of ARRAY is a string formed from two occurences of PROTOTYPE(A) separated by a comma. Thus, the index range is the same for both dimensions of the new array.

ITEM

In order to reference an array element by means of angular brackets, the array must be the value of a known identifier. Sometimes this is not the case. For example,

```
        $X    =    ARRAY(10)
```

is an acceptable assignment statement. But $X<2> and ($X)<2> do not reference the second element of the array. In the first expression, the unary $ operates on the value of X<2>; the second is syntactically erroneous.

There are two ways to refer to an element of such an array. The array can be assigned to a known identifier:

```
        TEMP    =    $X
        TEMP<2>    =    'SIX'
```

Alternatively, the primitive function ITEM can be used. The value of ITEM($a,i_1,\ldots,i_n$) is the $(i_1,\ldots,i_n)$-element of the array a.

```
        ITEM($X,2)    =    'SIX'
```

assigns SIX to the second element of the array.

Similarly, if

```
        A<1>    =    ARRAY(100)
```

is executed, the fiftieth item of this array may be referenced by ITEM(A<1>,50).

If an index referring to an element of an array falls outside the range of the array, the call of ITEM fails.

B. Names

A variable can be assigned a value during an assignment statement or by pattern matching through use of the cursor position operator @ or the binary value assignment operators . and $. In SNOBOL4, variables fall into two major classes, natural variables and created variables.

A natural variable is any variable whose name is a nonnull string. Thus,

```
                    A
                    $'AB'
                    $',,,('
```

are examples of natural variables, whose names, respectively, are the strings

```
                    A
                    AB
                    ,,,(
```

The variable ,,,(cannot appear explicitly in an assignment statement such as

```
        ,,,(  =  'X'
```

because it is syntactically incorrect. However,

```
        $',,,('  =  'X'
```

is syntactically correct and performs the desired assignment. Every string except the null string is the name of a natural variable. Natural variables are available at the start of a program without any conscious act of creation on the part of the programmer. All natural variables with the exception of ABORT, ARB, BAL, FAIL, FENCE, REM, and SUCCEED have the null string as their initial value.

Created variables are generated during execution of a program when, for example, an array is created. The statement

```
        A  =  ARRAY(10)
```

creates an array of ten variables. These variables are referred by A<1>, A<2>, ..., A<10>.

1. Passing Names

Consider a function BUMP which increments the value of any variable by 1. If the value of variable N is to be incremented, the call

```
        BUMP(N)
```

is not suitable because the value of N, not the name N, is passed to the procedure for BUMP. The form of the call must be

```
        BUMP('N')
```

which passes the string N to the BUMP procedure. Since the string N is the name of the variable N , indirect reference may be used to increment the value.

The defining statement and procedure for BUMP are:

```
        DEFINE('BUMP(VAR)')
            .
            .
BUMP    $VAR  =  $VAR  +  1          :(RETURN)
```

2. The Unary Name Operator

Suppose BUMP is to increment the value of a created variable, such as the second element of the array A. The call

```
        BUMP(A<2>)
```

is not suitable, since only the value of A<2> is passed. The call

```
        BUMP('A<2>')
```

is not suitable either, since the string A<2> is passed, and

```
        $'A<2>'
```

is a natural variable which bears no relation to the array element. The difficulty arises because there is no explicit name for created variables. However, implicit names for created variables can be obtained through use of a unary name operator.

The unary name operator . applied to any variable returns as value the name of that variable. Thus, the value of

```
        .A<2>
```

is the name of the second array element. The call

```
        BUMP(.A<2>)
```

passes the name of the second array element to BUMP, so that incrementing is done properly.

The name operator serves much the same purpose for created variables as quotation marks do for natural variables. Furthermore, the name operator applied to a natural variable behaves the same as quotation marks. Thus, the value of

```
        .LINE
```

is the string LINE . Both of the following pairs of statements assign the value 2 to MAY.

```
         WORD   =  'MAY'
         $WORD  =   2

         WORD   =  .MAY
         $WORD  =   2
```

If the argument of the name operator is a natural variable, the value returned by the name operator is a string which is an explicit name. If the argument of the name operator is a created variable, the value returned is an implicit name. If the argument is not a variable, error termination occurs. For example,

```
         .SIZE(X)
         .(A + B)
         .+A
```

are erroneous because the arguments are not variables. If A and B are integers or numeral strings,

```
         .$(A + B)
```

is valid because $(A + B) is a natural variable.

3. Returning a Variable

When returning from a programer-defined function via RETURN, the value of the function name becomes the value of the function call. If NRETURN is used, the value of the function name is returned as a variable, not as a value. The function call may thus be used freely in any context that requires a variable.

Consider, for example, the function NEXT which returns the first unused element of an array. The array is given as an argument and is assumed to have a zeroth element which indicates the last used element.

```
         DEFINE('NEXT(A)')                      :(NEXT.END)
*
NEXT     A<0>  =  A<0> + 1
         NEXT  =  .A<A<0>>                      :S(NRETURN)F(FRETURN)
*
NEXT.END
```

Thus, executing the four statements

```
         B  =  ARRAY('0:100')
         NEXT(B)   =   'A'
         NEXT(B)   =   'THE'
         'STILL'  'T'  REM . NEXT(B)
```

assigns to B<0> through B<3> values 3, A, THE, and ILL, respectively.

When NEXT returns, the value of NEXT is .B<B<0>>, which is the name of the first available array element. NEXT(B) becomes the variable B<B<0>> .

C. Gotos, Labels, and Code

Flow of control is governed by unconditional, success, and failure gotos. In the goto field, variables indicate the next statement to which control is passed based on the outcome of the current statement.

If a variable is used as a statement label, a label attribute pointing to the statement is assigned to the variable. This label attribute is independent of the value of the variable. Thus, a variable can be used in the label field and the goto field, as well as in the subject field of a single statement. The statement

```
DELAY     DELAY  =  LT(DELAY,N)  DELAY  +  1     :S(DELAY)F(ONWARD)
```

is acceptable and unambiguous.

If a variable has no label attribute, its use in a goto field causes error termination with the message, "UNDEFINED OR ERRONEOUS GOTO."

It is possible, as illustrated in the next section, to change the label attribute of a variable. In this way, a particular label variable, such as that appearing in

```
                  :S(LOOP)
```

may cause transfer to one statement at the beginning of execution and an entirely different statement later on.

In the first phase of a SNOBOL4 run (compilation), the source program is converted into Polish-prefix object code. In the second (execution) phase this object code is interpreted. Object code is a type of data just as are strings, patterns, and arrays. During the execution phase, it is possible, using the primitive function CODE(string), to convert a string of characters into object code. The argument to CODE is a string representing one or more SNOBOL4 statements. The value of a call to CODE is executable object code.

1. Creation and Execution of Code

A string to be compiled into object code consists of SNOBOL4 statements terminated by semicolons. For example, if the variable GET has a string value assigned by

```
      GET  =  '        N  =  10;'
.             '        LINE  =  ;'
.             'LOOP    N  =  GT(N,0)  N  -  1      :F(OUT);'
.             '        LINE  =  LINE  TRIM(INPUT)  :(LOOP);'
```

then

```
      NUCODE  =  CODE(GET)
```

causes the statements in the value of GET to be compiled. The value of CODE(GET) becomes the value of NUCODE.

Blanks are as important in strings to be converted to code as they are in the program itself. A statement without a label must begin with a blank.

Execution of statements in the value of NUCODE can be accomplished in two ways:

1) transfer to a labelled statement appearing in NUCODE, and

2) execution of a direct goto which passes control to the first statement in NUCODE, whether labelled or not.

Thus, execution of the goto

```
                    :(LOOP)
```

causes transfer to the statement labelled LOOP inside of NUCODE, even if the original program had a statement labelled LOOP.

A direct goto is a special construction in the goto field which permits transfer directly to the beginning of a block of object code rather than through a label. The direct goto uses enclosing angular brackets rather than parentheses. The expression enclosed in the angular brackets must be code valued. Execution of the direct goto

```
                    :<NUCODE>
```

causes transfer to the first statement

```
          N  =  10
```

Flowing off the end of a block of compiled object code results in normal termination, just as if there were an end statement.

The following statement illustrates the use of the function CODE in the goto itself.

```
                    :<CODE(' OUTPUT  =  "RECOMPILED"  :(RESTART);')>
```

The angular brackets indicate transfer to the beginning of the newly compiled block of CODE, which prints RECOMPILED and transfers to the statement labelled RESTART.

The primitive function CODE fails if its argument has a syntactic error.

It is an error for the same label to appear more than once in the source program. Statements compiled using CODE, however, may have the same labels as statements compiled earlier. The label attribute for the corresponding variable becomes the new statement. For example, the following program segment is used to call a function PROCESS(N) with various values of N.

```
BEGIN    N  =  5
LOOP     N  =  LT(N,10)   N  +  1                    :F(OUT)
         PROCESS(N)                                  :(LOOP)
OUT
         NEWLOOP  =  'LOOP    N  =  GT(N,0)   N  -  1  :F(END);'
         CODE(NEWLOOP)                               :(BEGIN)
       .
       .
       .
END
```

Within the two-statement loop, PROCESS(N) is called with N having values 6, 7, 8, 9, and 10 before control passes to the statement labelled OUT. At that point, a new block of code is compiled consisting of the statement

```
LOOP     N  =  GT(N,0)   N  -  1        :F(END)
```

Following compilation, control passes to the statement labelled BEGIN. It is intended that PROCESS(N) be called for N with values 4, 3, 2, 1, and 0, but this is not the case. The original statement labelled LOOP is still in the program. It is not overwritten by the compilation. The label attribute of LOOP no longer points to it. The label attribute now points at the newly compiled statement. The new compilation is a second program which can freely communicate with the original. Execution of the program proceeds as if the following programs were compiled.

```
BEGIN    N  =  5
         N  =  LT(N,10)   N  +  1                    :F(OUT)
         PROCESS(N)                                  :(LOOP)
OUT
         NEWLOOP  =  'LOOP    N  =  GT(N,0)   N  -  1  :F(END);'
         CODE(NEWLOOP)                               :(BEGIN)
       .
       .
       .
END


LOOP   N  =  GT(N,0)   N  -  1   :F(END)
END
```

After compilation of NEWLOOP, transfer to BEGIN causes N to be assigned the value 5. Control flows into the statement originally labelled LOOP, which increments N to 6. PROCESS(N) is called and, on completion, control passes to the new statement labelled LOOP. N is decremented to 5, but PROCESS cannot be called as intended, since the new statement does not overwrite the old, and no way is provided for getting back to the original program.

The program segment can be rewritten to perform as intended by using explicit gotos to control program flow rather than relying on the sequence of statements to control flow.

```
BEGIN    N  =  5                               :(LOOP)
LOOP     N  =  LT(N,10)   N  +  1              :F(OUT)
PROC     PROCESS(N)                            :(LOOP)
OUT
         NEWLOOP  =  'LOOP   N  =  GT(N,0)   N  -  1  :F(END)S(PROC);'
         CODE(NEWLOOP)                         :(BEGIN)
        .
        .
        .
END
```

Following compilation of NEWLOOP, execution proceeds as if the following programs were compiled.

```
BEGIN    N  =  5                               :(LOOP)
         N  =  LT(N,10)   N  +  1              :F(OUT)
PROC     PROCESS(N)                            :(LOOP)
OUT
         NEWLOOP  =  'LOOP   N  =  GT(N,0)   N  -  1  :F(END)S(PROC);'
         CODE(NEWLOOP)                         :(BEGIN)
        .
        .
        .
END


LOOP   N  =  GT(N,0)   N  -  1  :F(END)S(PROC)
END
```

After assigning 5 to N, control passes from the statement labelled BEGIN to the new statement labelled LOOP. N is properly decremented to 4 and control passes to the statement labelled PROC which calls PROCESS. The loop continues until N is 0.

D. Programmer-Defined Data Types

SNOBOL4 allows the programmer to define his own types of data objects. A programmer-defined data object is an ordered set of variables called fields. A call of DATA(p) defines a new data type described by the prototype p . The prototype p is a string denoting the name of the data type and the fields. For example, a complex number can be said to consist of two fields, the real and the imaginary. The call

```
DATA('COMPLEX(R,I)')
```

defines a data type COMPLEX, with two fields R and I. There is no intrinsic limit to the number of fields.

To create an object which has the data type COMPLEX, a call of the form

```
COMPLEX(e1,e2)
```

is made, where e1 and e2 are any expressions. For example, to assign the complex number "1.5 + 2.0i" to the variable C, the statement

```
         C  =  COMPLEX(1.5,2.0)
```

is executed. Each call of the function COMPLEX creates two new variables corresponding to the real and imaginary parts. These variables may be referenced by using the field name as a function. After executing the statement above, the value of C is a complex number; the real part is referenced by R(C) and the imaginary part by I(C). Thus,

```
         A  =  R(C)
```

assigns the value 1.5 to A. Since R(C) is a variable, it may be assigned a value. If

```
         R(C)   =  3.2
```

is executed, the complex number "3.2+2.0i" is assigned to C.

Operations on complex quantities can be defined using programmer-defined functions. A function to compute the sum of two complex quantities is

```
         DEFINE('SUM(C1,C2)')                        :(SUM.END)
SUM      SUM  =  COMPLEX(R(C1)  + R(C2),I(C1)  + I(C2))   :(RETURN)
SUM.END
```

If C has the value "3.2 + 2.0i", execution of the statement

```
         C  =  SUM(C,COMPLEX(1.0,1.0))
```

assigns "4.2 + 3.0i" to C.

Example - Text Processing

There is no intrinsic limit to the length of a string in SNOBOL4, but there is often a practical limit. For example, scanning a string for a pattern can be time consuming if the string is long. However, many string applications require reading in and retaining long passages of text. For such cases, a new data type called TEXT can be defined.

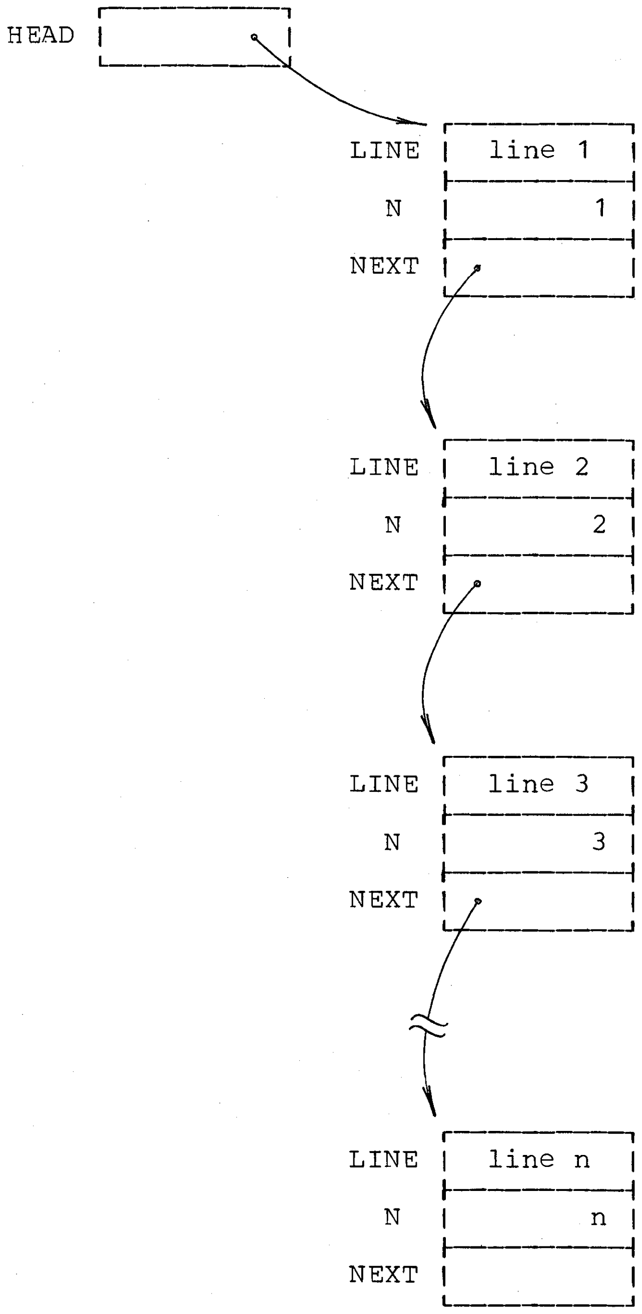
```
         DATA('TEXT(LINE,N,NEXT)')
```

The first field is a line of text, the second field indicates the line number, and the third field points to the next line of text.

A passage of text is read as follows:

```
        I   =   1
        HEAD  =   TEXT(INPUT,I)                    :F(EMPTY)
        CURRENT  =  HEAD
LOOP    I   =   I + 1
        NEXT(CURRENT)   =   TEXT(INPUT,I)          :F(DONE)
        CURRENT  =  NEXT(CURRENT)                  :(LOOP)
DONE
```

The resulting data structure has the form:



The statement

```
          LINE(HEAD)   'EVERY'                        :S(YES)F(NO)
```

examines the first line for the word EVERY .

The following section of program prints the lines and line numbers where EVERY occurs.

```
          CURRENT  =  HEAD
TEST      LINE(CURRENT)  'EVERY'                    :F(BUMP)
          OUTPUT  =  N(CURRENT)   ': '  LINE(CURRENT)
BUMP      CURRENT  =  NEXT(CURRENT)
          IDENT(CURRENT)                            :F(TEST)
```

The same field names may exist for several data types. Thus,

```
          DATA('LIST(VALUE,NEXT)')
```

defines a data type LIST which can coexist with the previous definition of the data type TEXT. Although NEXT is a field name for both TEXT and LIST, NEXT(X) is not ambiguous because the data type of the argument X indicates the usage.

<u>VALUE</u>

VALUE is a primitive field function defined on strings and names which refers to their value. If

```
        RADIX    =    'HEX'
```

then

```
        V    =    VALUE('RADIX')
```

assigns the string HEX to V . Similarly,

```
        VALUE('RADIX')    =    'DEC'
```

assigns the string DEC as the value of RADIX .

VALUE is supplied so that a programmer may define the field VALUE on programmer-defined data types, and then apply VALUE to strings and names as well as the defined types. This permits a uniform treatment of "value" without the necessity for checking data type. If

```
        DATA('LIST(VALUE,TEXT)')
        DATA('NODE(FATHER,LSON,RSIB,VALUE)')
```

are used to define the data types LIST and NODE, then VALUE can be applied to objects with data type LIST and NODE as well as names and strings.

E. Summary of Data Types

Data objects are classified by type. The string used to refer to a data type within the language is called the formal identification of the data type. The types of data are

Data Type	Formal Identification
string	STRING
integer	INTEGER
real number	REAL
pattern structure	PATTERN
array	ARRAY
created name	NAME
unevaluated expression	EXPRESSION
object code	CODE
programmer-defined data type	data type name

1. DATATYPE

The call

```
DATATYPE(e)
```

returns the formal identification of the data type of the value of the expression e . For example, the value of DATATYPE('A' | 'B') is the string PATTERN . Similarly,

```
DATATYPE(37)
DATATYPE(.ARB)
DATATYPE(.A<I>)
```

return

```
INTEGER
STRING
NAME
```

respectively.

If the argument to DATATYPE is a programmer-defined data type, the data type name is returned. Referring to the data types defined in the previous section, the function calls

```
DATATYPE(C)
DATATYPE(CURRENT)
```

return

```
COMPLEX
TEXT
```

respectively.

2. Data Type Conversion

In some cases, it is reasonable to speak of the conversion of a data object of one data type into a corresponding data object of some other data type. This can be accomplished using the CONVERT function. For example, an integer can be converted to a real number by the statement

```
R  =  CONVERT(2,'REAL')
```

As a result, R has the real number 2.0 as its value. To convert R to a string, so that it may be printed, the statement

```
OUTPUT  =  CONVERT(R,'STRING')
```

may be used.

CONVERT has the form

```
CONVERT(expression,datatype)
```

The first argument is any expression and the second is a string-valued expression corresponding to a formal identification of a data type. CONVERT evaluates the first argument and then, if possible, converts the result to the data type given by the second argument. The value of CONVERT is the value of expression converted to the new data type.

Not all conversions are possible or meaningful. The CONVERT function fails if a specified conversion cannot be made. The following table indicates those conversions that are implemented. The conversion from STRING to CODE performs the same task as the CODE function used earlier. That is,

```
CODE(S)
```

and

```
CONVERT(S,'CODE')
```

may be used interchangeably.

data type of argument	data type of object returned								
	S	I	R	P	A	N	E	C	D
STRING	X	X	X				X	X	
INTEGER	X	X	X						
REAL	X		X						
PATTERN				X					
ARRAY					X				
NAME						X			
EXPRESSION							X		
CODE								X	
Defined Data Type									X

An object of one programmer-defined data type cannot be converted to an object of a different programmer-defined data type.

3. COPY

COPY was described earlier in connection with arrays. The value of

```
     COPY(A)
```

is a new array identical in every respect to the array which was the value of A. The COPY function can also be used for data objects other than ARRAYS. Objects with data type PATTERN, CODE, and all programmer-defined data types can be copied. In all cases, the value of COPY is a new instance of the data object which is its argument.

F. Keywords

Certain identifiers prefixed by an ampersand (&) provide the programmer with access to, and in some cases control of, information used internally by the SNOBOL4 system. For example, the programmer may determine, at some point, how many statements have been executed. The value of &STCOUNT is an integer equal to the number of statements executed. If the statement

```
     GT(&STCOUNT,40000)                        :S(CLEAN.UP)
```

is executed after more than 40,000 statements have been executed, a transfer to the statement labelled CLEAN.UP is made. As another example, &STLIMIT is a variable whose value is the number of statements which may be executed before the SNOBOL4 system unconditionally terminates the program. The initial value of &STLIMIT is 50000, but it can be changed during execution.

```
     &STLIMIT  =  &STLIMIT * 2
```

doubles this limit.

Whereas the value of &STLIMIT can be changed, the value of &STCOUNT cannot. Those keywords which can be modified by programmer action are called unprotected keywords; those which cannot are called protected keywords. An attempt to set the value of a protected keyword results in error termination.

Protected Keywords

There are two kinds of protected keywords, varying and constant. As their names suggest, the values of varying protected keywords change automatically during execution of a program. The constant protected keywords do not change.

1. Varying Protected Keywords

a. &FNCLEVEL. The value of &FNCLEVEL is the level of programmer-defined function call.

b. &LASTNO. The compiler numbers each statement. These numbers are used principally for diagnostic purposes. The value of &LASTNO is the number of the last statement executed.

c. &RTNTYPE. The value of &RTNTYPE is the string RETURN, FRETURN, or NRETURN, depending on the kind of return last made by a programmer-defined function.

d. &STCOUNT. The value of &STCOUNT is the number of statements which have been entered during program execution. If

```
          N  =  &STCOUNT
```

is the first statement executed in a program, then N has the value 1.

e. &STFCOUNT. The value of &STFCOUNT is the number of statements which have failed. If

```
          N  =  &STFCOUNT
```

is the first statement executed, the value of N is 0.

f. &STNO. The value of &STNO is the compiler-assigned number of the statement currently being executed. (See &LASTNO.)

2. Constant Protected Keywords

a. &ALPHABET. The value of &ALPHABET is a string consisting of all the characters of the machine on which SNOBOL4 is implemented. The characters are ordered according to their internal coding.

b. &ARB. The value of &ARB is the primitive pattern structure which matches any string of characters. &ARB and ARB have the same value at the beginning of program execution. The value of ARB may be changed, however, while the value of &ARB is protected.

c. &ABORT. &ABORT has the same value as ABORT at the beginning of program execution. See &ARB.

d. &BAL. As above.

e. &FAIL. As above.

f. &FENCE. As above.

g. &REM. As above.

h. &SUCCEED. As above.

Unprotected Keywords

There are two kinds of unprotected keywords, switches and parameters. A switch is a keyword requiring an integer value. A switch is considered off if its value is 0, and is considered on otherwise. All switches are off at the beginning of program execution.

1. Switches

a. &ABEND. If &ABEND is on when program execution terminates, a system core dump is provided.

b. &ANCHOR. If &ANCHOR is on, a pattern can match only an initial substring. See Chapter 2.

c. &DUMP. If &DUMP is on at program termination, natural variables and their values are printed.

d. &FTRACE. If &FTRACE is on, calls to and returns from all programmer-defined functions are traced. See Chapter 7.

e. &FULLSCAN. If &FULLSCAN is on, the pattern matching scanner attempts to match a complex pattern against a string even though it can be predetermined that the attempt will fail. See Chapter 2.

f. &TRACE. Tracing capabilities are available if &TRACE is on. See Chapter 7.

2. Parameters

a. &MAXLNGTH. The value of &MAXLNGTH is an integer equal to the largest string (measured in characters) which may be formed. The initial value of &MAXLNGTH is 5000, but this value may be changed. Thus,

```
&MAXLNGTH  =  1000
```

limits the maximum length of subsequent strings to 1000 characters. An attempt to form a string longer than the limit results in error termination of the program. All types of string formations are included in this limit: concatenation, replacement, value assignment as a result of pattern matching, and string input.

b. <u>&STLIMIT</u>. The value of &STLIMIT is the limit on the number of statements that may be executed (see &STCOUNT). The initial value of &STLIMIT is 50000. Exceeding the limit on statement execution results in error termination.

Chapter 6. Details of Evaluation

A. The Components of a Statement

There are three major types of statements: assignment, pattern matching, and replacement. These have the forms:

```
label  subject   =  object   goto
label  subject  pattern   goto
label  subject  pattern   =   object   goto
```

Labels and gotos are optional. The object may be explicitly omitted, in which case the object is taken to be an expression that has the null string as value.

There are two degenerate statement forms as well:

```
label  subject   goto
label  goto
```

Labels and gotos are optional in these forms as well. Thus a blank line is an acceptable statement.

B. Statement Evaluation

An understanding of the sequence of evaluation requires an understanding of the overall evaluation of a statement in terms of its major components. The replacement statement is the most complicated and general form and is used for illustration. All other statement forms can be considered formally as degenerate replacement statements, and the evaluation of the degenerate forms can be understood from the evaluation of the replacement statement by skipping the missing components. The sequence of evaluation is:

1. The label requires no evaluation, and in fact is not part of the statement at all. It merely serves to identify the statement.

2. The subject is evaluated first. If the evaluation of the subject fails, the statement fails, the goto is processed, and evaluation of all other components is skipped. If no failure goto is specified, control passes to the next statement.

3. The pattern is evaluated next. If this evaluation fails, the statement fails and the goto is processed as in the case of subject failure.

4. The pattern match is performed next. If the pattern match fails, the statement fails, conditional value assignment is not performed, the replacement is skipped, and the goto is processed. Immediate value assignment, and other effects which occur dynamically during pattern matching, may take place before the pattern match fails.

5. The object is evaluated. If this evaluation fails, the statement fails, no replacement is performed, and the goto is processed.

6. The replacement is performed.

7. The goto is processed. Goto processing depends on the structure of the goto and whether or not the statement failed. If the statement succeeded, only an unconditional or success goto in the statement is evaluated. If the statement failed, only an unconditional goto or failure goto in the statement is evaluated. Transfer is made to the evaluated goto if there is one, or control is passed to the next statement. If evaluation of a goto fails, error termination results.

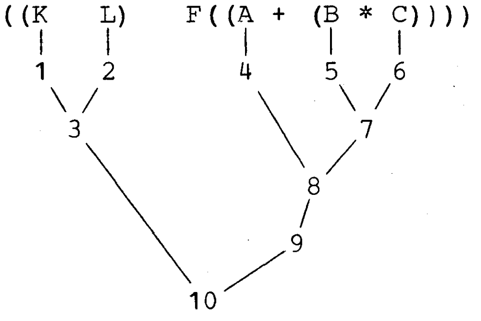
Any of the components of a statement may be arbitrarily complicated and may invoke all kinds of processes. Calls to programmer-defined functions can occur, for example, in any component of a statement (except the label), and even take place in the middle of pattern matching as the result of the evaluation of unevaluated expressions.

Within an expression, the order of evaluation depends on the order of the components and the operations performed on them. Evaluation of the components of an expression is from left to right. In complicated expressions, components are nested, and the order of evaluation may be determined by examining the fully parenthesized form of the expression as determined from the rules of precedence and association. Consider the expression

```
(K   L   F(A + B * C))
```

which has the fully parenthesized form

```
((K   L)   F((A + (B * C))))
```



The order of evaluation of this expression is as indicated. If F is a programmer-defined function, its evaluation involves the execution of other statements and may in itself be very complicated.

In order to understand how failure is handled, it is important to know what operations can fail.

1. Obtaining the value of a variable fails if the variable has an input association and an end-of-file condition is encountered. Such failure occurs only if the value of the variable is required, not merely because the variable appears in a statement. Thus, neither

```
INPUT   =   '0'
```

nor

```
     LT(N,M)      :S(INPUT)
```

requires the value of INPUT and hence no attempt is made to read a record.

2. Primitive predicates fail if the stated condition is not met. The unary negation operator, for example, fails only if its operand does not fail.

3. Some primitive functions such as REPLACE fail for certain argument values.

4. Array references fail if an index is out of bounds.

5. Pattern matching may fail for a variety of reasons.

6. Programmer-defined functions fail by transferring to FRETURN.

Failure is a condition that causes a process to terminate and return to the process that called it, which in turn terminates and passes the failure condition back, until eventually the statement itself fails. The exception is the unary negation operator that converts a failure condition into successful evaluation, and conversely.

Details of function evaluation deserve special note. All the arguments to a programmer-defined function are evaluated before the function is called. If too many arguments are provided to the call of a programmer-defined function, the extra arguments are evaluated, but not passed. If the evaluation of any argument fails, a failure condition is returned and the function is not entered.

Primitive functions are called before their arguments are evaluated, and each function evaluates its own arguments. If the are too many arguments in the call of a primitive function, error termination results. If too few arguments are provided in the call of a primitive function, null strings are provided for the omitted arguments. An exception to this rule concerns functions invoked by APPLY or called through an OPSYNed synonym. Such calls must contain the correct number of arguments or error termination results.

Integers and Strings

Integers can occur as literals and as the result of integer-valued operations. An integer literal consists of an unsigned sequence of digits. Some integer literals are

```
          35
     2760520
       00006
```

Leading zeroes are ignored; 00006 and 6 are equivalent. A sign in front of an integer literal is a unary operator and not part of the literal. Thus -6 is an integer-valued expression.

The maximum magnitude of integers is implementation dependent. On the IBM System/360, integers can range from -2^{31} to $2^{31}-1$,

Numeral strings are strings that represent integers. Numeral strings consist of a sequence of digits and can have an initial sign. Some numeral strings and their equivalent integer values are

```
      '23'            23
      '-7'            -7
    '+303'           303
   '00001'             1
```

The null string is also a numeral string and is equivalent to the integer zero. The following strings are _not_ numeral strings:

```
       '+'
     '++3'
   '1,378'
     '36-'
       ' '
     '2.0'
```

Many operations require integer-valued arguments. An integer-valued argument can be specified by either an integer or a numeral string. Both

```
LEN(8)
```

and

```
LEN('8')
```

are correct. In most cases integers and numeral strings can be used interchangeably, and the programmer need not concern himself with the difference. In fact, numeral strings are automatically converted to equivalent integers in contexts where integers are required.

Similarly, integers can be used in operations that require string-valued arguments. Integers are automatically converted to numeral strings in contexts where strings are required. In the statement

```
SEQNO    0    =    1
```

the pattern and object are integers. The pattern is converted into the _string_ 0 for the purpose of pattern matching and the object is converted into the _string_ 1 for the purpose of replacement. An equivalent statement is

```
SEQNO    '0'    =    '1'
```

Conversion of integers to strings produces a _normalized_ result with no leading zeroes and without a leading plus for positive integers. Printing requires strings, for example. Thus,

```
OUTPUT    =    8
```

and

```
        OUTPUT    =    00008
```

both print

```
8
```

but

```
        OUTPUT    =    '00008'
```

prints

```
00008
```

The effects of this conversion are most likely to be noticeable when conversion from a numeral string to an integer is followed by conversion back to a string.

```
        OUTPUT    =    '-00007' + 00009
```

prints

```
2
```

Real Numbers

Real numbers can occur as literals and as the result of real-valued operations. A real number consists of an unsigned sequence of digits, followed by a period, optionally followed by another sequence of digits. Some real literals are

```
        20.05
        0.00001
        3.
```

A sign in front of a real literal is a unary operator and not part of the literal. Thus -3.14159 is a real-valued expression.

On the IBM System/ 360, the range of real numbers is on the order of 10^{-78} to 10^{75}.

Real numbers are automatically converted to strings for the purpose of printing or punching. No other automatic conversions are made. Real numbers cannot be concatenated. To perform mixed arithmetic on integers and real numbers, explicit conversions must be made using the CONVERT function.

Operators

Unary and binary operators are functions of one and two arguments, respectively. Operators have a special status by virtue of their syntactic representation as distinguished symbols. The following sections discuss details of the operators and the relation between their operands and values.

Unary Operators

There are eleven unary operators.

operator	operation
+	plus
-	minus
$	indirect reference
*	expression
.	name
¬	negation
?	interrogation
&	keyword
@	cursor position
#	(not used)
%	(not used)
/	(not used)

The following sections describe permissible operands for the unary operators. Only data types indicated in these sections are permitted. Other data types result in error termination. Abbreviations for the data types correspond to the usage in Chapter 5. In the tables that follow, the left column indicates the permissible operand data types and the right column indicates the data types resulting from the operation.

plus and minus

Plus and minus accept the same types of operands and return the same types of values.

S	I
I	I
R	R

Strings occurring as operands in these arithmetic operations must be numeral strings.

indirect reference

Indirect reference requires an operand that is either a name or a string, and returns the corresponding variable. This variable in turn may have any type of data as value.

S	(variable)
I	(variable)
N	(variable)

expression

The expression operator may have any expression as an operand. A pointer to this expression is returned, but the operand is not evaluated. The pointer has data type EXPRESSION. Subsequent evaluation of the expression (during pattern matching, e.g.) may yield a variable or a value of any data type.

name

The name operator must have a variable as an operand. A pointer to this variable is returned. If the operand is a natural variable, the resulting data type is STRING; otherwise it is NAME.

(natural variable)	S
(other variable)	N

negation and interrogation

Negation and interrogation accept any expression as operand. If the operations succeed, they return the null string as value.

keyword

The keyword operator accepts as an operand only certain natural variables. The data type of the value depends on the particular keyword. The natural variable operand need not appear explicitly, but can be computed. Consequently,

```
        KEYWORD   =    'STCOUNT'

            .
            .
            .

        OUTPUT   =    &$KEYWORD
```

prints the number of statements executed up to the time the output statment occurs.

cursor position

The cursor position operator must have a variable as operand. A pattern structure is returned.

unused operators

The symbols /, #, and % are reserved for future use. They are accepted syntactically as unary operators, but have no meaning. Execution of one of these operations causes error termination.

Binary Operators

There are twelve binary operators. Exponentiation associates to the right. All other operations associate to the left. The operators are listed below in order of decreasing precedence. Notice that multiplication has higher precedence than division, contrary to common practice in other programming langauges.

operator	operation
$.	immediate and conditional value assignment
**	exponentiation
%	(not used)
*	multiplication
/	division
#	(not used)
+ -	addition and subtraction
@	(not used)
	concatenation
|	alternation

The following sections describe permissible operands for the binary operators. In the tables that follow, the left column indicates the permissible left operand data types, the top row indicates the permissible right operand data types, and the body of the table indicates the data types resulting from the operation. Blanks in the body of the table indicate a combination of operand data types that is not permitted.

addition, subtraction, multiplication, and division

Addition, subtraction, multiplication, and division all accept the same types of operands and return the same types of values.

	S	I	R
S	I	I	
I	I	I	
R			R

Strings occurring as operands in these arithmetic operations must be numeral strings.

exponentiation

Exponentiation is similar to the other arithmetic operations except that real operands are not permitted.

	S	I
S	I	I
I	I	I

Strings must be numeral strings.

concatenation

Concatenation is an operation of central importance in SNOBOL4. The permissible data type combinations are:

	S	I	P	E
S	S	S	P	P
I	S	S	P	P
P	P	P	P	P
E	P	P	P	P

Concatenation treats the null string in a special way. If either operand is the null string, concatenation is not performed and the other operand is returned as value. Thus, if one operand is the null string, the other operand may have any data type. This treatment of the null string permits full use of predicates in expressions containing various types of data.

alternation

The permissible data type combinations for alternation are:

	S	I	P	E
S	P	P	P	P
I	P	P	P	P
P	P	P	P	P
E	P	P	P	P

Notice that the result of alternation is always a pattern. The null string has no special status in alternation.

immediate and conditional value assignment

The value-assignment operations require a right operand that is a variable. This variable, not its value, is used in constructing a pattern. An exception to this requirement permits the right operand to be an unevaluated expression. This expression is then evaluated at the time of value assignment to obtain the variable to which assignment is made. If such an unevaluated expression does not produce a variable at the time of value assignment, error termination occurs. The permissible left operands are:

```
  ┌───
S | P
I | P
P | P
E | P
```

<u>unused operators</u>

The symbols %, #, and @ are reserved for future use. They are accepted syntactically as binary operators, but have no meaning. Execution of one of these operations causes error termination.

<u>Variables and Values</u>

Some expressions yield variables when evaluated. Such variables are called generated variables, and values can be assigned to them in the same manner that values can be assigned to variables that appear explicitly. In the statements

```
     M    =    2
     $('N' M)    =    'INVOICE'
```

the subject $('N' M) generates the variable N2 which is assigned the value INVOICE . Array references, field functions on programmer-defined data types, and programmer-defined functions that return by NRETURN are examples of expressions that generate variables.

Other expressions, for example arithmetic operations, yield values but not variables. Thus, execution of the statement

```
     (A + B)    =    2
```

causes error termination with the message "VARIABLE NOT GIVEN WHERE REQUIRED."

Gotos require natural variables. These natural variables may also be generated. The indirect goto

```
                    :S($TRIM(INPUT))
```

is an example.

Some expressions, such as indirect references, always yield variables. Others, such as literals, always yield only values. Some expressions may or may not yield variables. For example,

```
     F(X)    =    2
```

may or may not be erroneous depending on the function F. To allow for such cases, the syntax of SNOBOL4 permits any kind of expression as the subject of assignment. Statements such as

```
    2 = 3
```

are syntactically acceptable even though they result in error termination if executed.

Chapter 7. Tracing

Tracing facilities are provided to permit the programmer to get diagnostic information about the execution of his program without interfering with its logic or structure. The tracing mode is entered by turning on the keyword &TRACE. When this mode is in effect, certain types of program actions can be sensed, causing corresponding messages to be printed. The types of actions sensed are:

1) change in the value of a variable,
2) call of a defined function,
3) return from a defined function,
4) transfer to a label, and
5) change in the value of certain keywords.

A. Standard Trace Procedures

The TRACE function is used to make specific trace requests.

```
TRACE(name,type,tag)
```

associates the name with the type of action for tracing purposes. The tag provides identifying information which is included in the trace printout if the name is not a natural variable. If the name is a natural variable, the tag is ignored. One trace association must be made for each name and type desired. Trace printout includes the statement number in which the action occurs, the result of the action, and the time of the action in milliseconds measured from the beginning of program execution.

If &TRACE is off, there is no tracing, even though trace requests have been made. The value of &TRACE is decremented by one every time an action is traced, and tracing is automatically turned off when the value of &TRACE reaches zero. Therefore the value assigned to &TRACE may be chosen to limit the amount of trace printout.

1. Value Tracing

```
TRACE(name,'VALUE',tag)
```

causes trace printout whenever the value of the name is changed. Consider the following program.

```
         TRACE('I','VALUE')
         TRACE('J','VALUE')
         &TRACE   =   1000000
*   LET THE FIRST DATA CARD SPECIFY THE MAXIMUM NUMBER OF
*   CARDS TO BE SORTED.  GENERATE AN ARRAY.
*
         A   =   ARRAY(TRIM(INPUT))
*
*   DEFINE THE FUNCTION INSERT.
*
         DEFINE('INSERT(J)TEMP')
*
*   READ THE CARDS INTO THE ARRAY.
*
INIT     I   =   I + 1
         A<I>    =    TRIM(INPUT)                :F(SORT)
         OUTPUT   =   A<I>                       :(INIT)
*
*   LET N BE THE NUMBER OF CARDS.  INITIALIZE THE INDEX AND
*   THEN SORT.
*
SORT     N   =   I - 1
         I   =   1
*
*   COMPARE TWO SUCCESSIVE CARDS.
*
SORTA    LGT(A<I>,A<I + 1>)                      :S(SORTC)
*
*    IF THEY ARE IN THE PROPER ORDER,INCREMENT THE INDEX
*    (UNLESS SORTING IS FINISHED) AND CONTINUE.
*
SORTB    I   =   LT(I,N - 1)   I + 1             :S(SORTA)F(DONE)
*
*   OTHERWISE, INSERT THE CARD IN ITS PROPER PLACE.
*
SORTC    INSERT(I + 1)                           :(SORTB)
*
*   PUNCH SORTED CARDS.
*
DONE     I   =   1
         OUTPUT   =
PUNCH    OUTPUT   =   A<I>
         PUNCH   =   OUTPUT
         I   =   LT(I,N)   I + 1                 :S(PUNCH)F(END)
*
*   FUNCTION DEFINITION
*
INSERT   TEMP   =   A<J - 1>
         A<J - 1>   =   A<J>
         A<J>   =   TEMP
         J   =   GT(J,2)   J - 1                 :F(RETURN)
         LGT(A<J - 1>,A<J>)                      :S(INSERT)F(RETURN)
END
```

Given the data

```
10
ACOMPC
ACOMP
INTRL
SPECEQ
SUM
FORMAT
STREAM
ZERBLK
SETAV
SETVA
```

the printed output is

```
    STATEMENT 6: I = 1,TIME = 17
ACOMPC
    STATEMENT 6: I = 2,TIME = 17
ACOMP
    STATEMENT 6: I = 3,TIME = 50
INTRL
    STATEMENT 6: I = 4,TIME = 67
SPECEQ
    STATEMENT 6: I = 5,TIME = 84
SUM
    STATEMENT 6: I = 6,TIME = 84
FORMAT
    STATEMENT 6: I = 7,TIME = 100
STREAM
    STATEMENT 6: I = 8,TIME = 117
ZERBLK
    STATEMENT 6: I = 9,TIME = 117
SETAV
    STATEMENT 6: I = 10,TIME = 134
SETVA
    STATEMENT 6: I = 11,TIME = 233
    STATEMENT 10: I = 1,TIME = 233
    STATEMENT 12: I = 2,TIME = 250
    STATEMENT 12: I = 3,TIME = 250
    STATEMENT 12: I = 4,TIME = 250
    STATEMENT 12: I = 5,TIME = 267
    STATEMENT 22: J = 5,TIME = 267
    STATEMENT 22: J = 4,TIME = 283
    STATEMENT 22: J = 3,TIME = 283
    STATEMENT 12: I = 6,TIME = 283
    STATEMENT 22: J = 6,TIME = 300
    STATEMENT 12: I = 7,TIME = 300
    STATEMENT 12: I = 8,TIME = 300
    STATEMENT 22: J = 8,TIME = 317
    STATEMENT 22: J = 7,TIME = 317
    STATEMENT 22: J = 6,TIME = 317
    STATEMENT 22: J = 5,TIME = 333
    STATEMENT 12: I = 9,TIME = 333
    STATEMENT 22: J = 9,TIME = 350
    STATEMENT 22: J = 8,TIME = 350
    STATEMENT 22: J = 7,TIME = 367
    STATEMENT 22: J = 6,TIME = 367
    STATEMENT 14: I = 1,TIME = 367
```

```
ACOMP
    STATEMENT 18: I = 2,TIME = 533
ACOMPC
    STATEMENT 18: I = 3,TIME = 533
FORMAT
    STATEMENT 18: I = 4,TIME = 550
INTRL
    STATEMENT 18: I = 5,TIME = 550
SETAV
    STATEMENT 18: I = 6,TIME = 550
SETVA
    STATEMENT 18: I = 7,TIME = 566
SPECEQ
    STATEMENT 18: I = 8,TIME = 566
STREAM
    STATEMENT 18: I = 9,TIME = 566
SUM
    STATEMENT 18: I = 10,TIME = 583
ZERBLK
```

If the name is not a natural variable, the tag is printed to identify the name being traced. For example,

```
       TRACE(.SUM<3>,'VALUE','SUM<3>')
```

traces the third element of the array SUM. Here the tag SUM<3> (chosen to correspond to the created variable SUM<3>) provides a string that identifies the name of the trace request. As an example, consider the following program which forms sums in several bins as given on data cards. The trace association must appear after creation of the array SUM , since the name .SUM<3> does not exist before the array is created.

```
             &ANCHOR   =   1                                                  1
             &TRACE   =   1000                                                2
             CARDPAT   =   BREAK(' ') . BIN LEN(1) BREAK(' ') . NUMBER        3
     *       THE FIRST CARD GIVES THE NUMBER OF BINS
             SUM   =   ARRAY(TRIM(INPUT),0)                  :F(ERR)          4
     *       TRACE THE THIRD BIN.
             TRACE(.SUM<3>,'VALUE','SUM<3>')                                  5
     *       SUBSEQENT CARDS CONTAIN A BIN NUMBER FOLLOWED BY A BLANK AND THEN
     *       THE NUMBER TO BE ADDED TO THE BIN.
     READ    CARD   =   INPUT                                :F(DISPLAY)      6
             CARD   CARDPAT                                  :F(ERR)          7
             SUM<BIN>   =   SUM<BIN> + NUMBER                :S(READ)F(ERR)   8
     *       PRINT OUT THE SUMS
     DISPLAY                                                                  9
             I   =   1                                                       10
     PRINT   OUTPUT   =   'SUM<' I '> = ' SUM<I>             :F(END)         11
             I   =   I + 1                                   :(PRINT)        12
     END                                                                     13
```

For the input data

```
10
3 25
1 27
9 -75
5 +65
3 77
7 -89
2 75
10 0
3 -756
7 499
2 76
4 23
1 456
5 87
2 33
10 23
3 0025
8 657
3 -45
```

the printed output is:

```
    STATEMENT 8: SUM<3> = 25,TIME = 17
    STATEMENT 8: SUM<3> = 102,TIME = 50
    STATEMENT 8: SUM<3> = -654,TIME = 100
    STATEMENT 8: SUM<3> = -629,TIME = 183
    STATEMENT 8: SUM<3> = -674,TIME = 300
SUM<1> = 483
SUM<2> = 184
SUM<3> = -674
SUM<4> = 23
SUM<5> = 22
SUM<6> = 0
SUM<7> = 410
SUM<8> = 657
SUM<9> = -75
SUM<10> = 23
```

2. Function Tracing

There are three types of tracing for programmer-defined functions: CALL, RETURN, and FUNCTION. CALL and RETURN cause trace printout on the call to and return from a function. FUNCTION causes trace printout for both call and return.

CALL tracing gives the level from which the call is made, the function name, and the value of its arguments. RETURN tracing gives the level to which the return is made. The following examples indicate the three types of tracing applied to a program that computes the number of combinations of N things taken M at a time.

```
          &TRACE    =   1000                                                   1
          TRACE('C','CALL')                                                    2
          NM    =  BREAK(',') . N ',' BREAK(' ') . M                           3
          DEFINE('C(N,M)')                                                     4
 READ     INPUT   NM                        :F(END)                            5
          OUTPUT   =   'C(' N ',' M ')=' C(N,M)     :(READ)                    6
 *
 C        M  =  LT(N - M,M)  N - M                                             7
          C  =  EQ(M,0)  1                  :S(RETURN)                         8
          C  =  N * C(N - 1,M - 1) / M  :(RETURN)                              9
 END                                                                           10
```

produces the output

```
    STATEMENT 6: LEVEL 0 CALL OF C('15','6'),TIME = 200
    STATEMENT 9: LEVEL 1 CALL OF C(14,5),TIME = 200
    STATEMENT 9: LEVEL 2 CALL OF C(13,4),TIME = 216
    STATEMENT 9: LEVEL 3 CALL OF C(12,3),TIME = 216
    STATEMENT 9: LEVEL 4 CALL OF C(11,2),TIME = 216
    STATEMENT 9: LEVEL 5 CALL OF C(10,1),TIME = 233
    STATEMENT 9: LEVEL 6 CALL OF C(9,0),TIME = 233
C(15,6)=5005
```

With RETURN tracing, the output is

```
    STATEMENT 8: LEVEL 6 RETURN OF C = 1,TIME = 133
    STATEMENT 9: LEVEL 5 RETURN OF C = 10,TIME = 150
    STATEMENT 9: LEVEL 4 RETURN OF C = 55,TIME = 216
    STATEMENT 9: LEVEL 3 RETURN OF C = 220,TIME = 216
    STATEMENT 9: LEVEL 2 RETURN OF C = 715,TIME = 233
    STATEMENT 9: LEVEL 1 RETURN OF C = 2002,TIME = 233
    STATEMENT 9: LEVEL 0 RETURN OF C = 5005,TIME = 233
C(15,6)=5005
```

and with FUNCTION tracing the result is

```
    STATEMENT 6: LEVEL 0 CALL OF C('15','6'),TIME = 134
    STATEMENT 9: LEVEL 1 CALL OF C(14,5),TIME = 134
    STATEMENT 9: LEVEL 2 CALL OF C(13,4),TIME = 217
    STATEMENT 9: LEVEL 3 CALL OF C(12,3),TIME = 217
    STATEMENT 9: LEVEL 4 CALL OF C(11,2),TIME = 217
    STATEMENT 9: LEVEL 5 CALL OF C(10,1),TIME = 233
    STATEMENT 9: LEVEL 6 CALL OF C(9,0),TIME = 233
    STATEMENT 8: LEVEL 6 RETURN OF C = 1,TIME = 250
    STATEMENT 9: LEVEL 5 RETURN OF C = 10,TIME = 250
    STATEMENT 9: LEVEL 4 RETURN OF C = 55,TIME = 250
    STATEMENT 9: LEVEL 3 RETURN OF C = 220,TIME = 250
    STATEMENT 9: LEVEL 2 RETURN OF C = 715,TIME = 267
    STATEMENT 9: LEVEL 1 RETURN OF C = 2002,TIME = 267
    STATEMENT 9: LEVEL 0 RETURN OF C = 5005,TIME = 267
C(15,6)=5005
```

To facilitate the tracing of programmer-defined functions, the keyword &FTRACE is provided. When &FTRACE is on, <u>all</u> programmer-defined functions are traced on call and return. The value of &FTRACE is decremented by one each time a programmer-defined function is called or returns. &TRACE and &FTRACE are independent, and both may be used at the same time. The following program illustrates the use of &FTRACE.

```
          &FTRACE    =    1000                                                 1
*
*        THIS PROGRAM COMPUTES THE NUMBER OF SYMMETRIC BISECTIONS OF
*        A CHECKERBOARD OF EVEN ORDER.  THE PROBLEM IS DESCRIBED IN
*        MARTIN GARDNER'S "MATHEMATICAL GAMES" IN SCIENTIFIC AMERICAN
*        NOVEMBER, 1962.
*
          DEFINE('AXIS(X,Y)')                                                  2
          DEFINE('RIGHT(X,Y)')                                                 3
          DEFINE('LEFT(X,Y)')                                                  4
          DEFINE('UP(X,Y)')                                                    5
          DEFINE('DOWN(X,Y)')                                                  6
          DEFINE('COUNT(X)')                                                   7
*
READ      SUM    =    0                                                        8
          N    =    TRIM(INPUT)                           :F(END)              9
          BOARD    =    ARRAY(-N ':' N ',' -N ':' N)                          10
          BOARD<0,0>    =    ':'                                              11
          AXIS(0,0)                                                           12
          OUTPUT    =    'THERE ARE ' SUM ' SYMMETRIC BISECTIONS OF A ' 2 *   13
.         N ' BY ' 2 * N ' CHECKERBOARD'                  :(READ)             13
*
AXIS      X    =    X + 1                                                     14
          EQ(X,N)  COUNT()                                :S(RETURN)          15
          IDENT(BOARD<-X,-Y>)                             :F(FRETURN)         16
          IDENT(BOARD<X,Y>)                               :F(FRETURN)         17
          BOARD<X,Y>    =    ':'                                              18
          AXIS(X,Y)                                                           19
          UP(X,Y)                                                             20
          BOARD<X,Y>    =                                 :(RETURN)           21
*
RIGHT     X       =  X  +  1                                                  22
          EQ(X,N)      COUNT()                            :S(RETURN)          23
          IDENT( BOARD<-X,-Y>)                            :F(FRETURN)         24
          IDENT(BOARD<X,Y>)                               :F(FRETURN)         25
          BOARD<X,Y>    =  ':'                                                26
          RIGHT(X,Y)                                                          27
          UP(X,Y)                                                             28
          DOWN(X,Y)                                                           29
          BOARD<X,Y>     =                                :(RETURN)           30
*
UP        Y       =  Y  +  1                                                  31
          EQ(Y,N)      COUNT()                            :S(RETURN)          32
          IDENT(BOARD<-X,-Y>)                             :F(FRETURN)         33
          IDENT(BOARD<X,Y>)                               :F(FRETURN)         34
          BOARD<X,Y>    =  ':'                                                35
          RIGHT(X,Y)                                                          36
          UP(X,Y)                                                             37
          LEFT(X,Y)                                                           38
          BOARD<X,Y>     =                                :(RETURN)           39
*
LEFT      X       =  X  -  1                                                  40
          EQ(X,-N)     COUNT()                            :S(RETURN)          41
          IDENT(BOARD<-X,-Y>)                             :F(FRETURN)         42
          IDENT(BOARD<X,Y>)                                      :F(FRETURN)      43
          BOARD<X,Y>    =  ':'                                                    44
          LEFT(X,Y)                                                               45
          UP(X,Y)                                                                 46
          DOWN(X,Y)                                                               47
          BOARD<X,Y>    =                                        :(RETURN)        48
*
DOWN      Y        =  Y  -  1                                                     49
          EQ(Y,-N)                COUNT()                        :S(RETURN)       50
          IDENT(BOARD<-X,-Y>)                                    :F(FRETURN)      51
          IDENT(BOARD<X,Y>)                                      :F(FRETURN)      52
          BOARD<X,Y>    =  ':'                                                    53
          RIGHT(X,Y)                                                              54
          LEFT(X,Y)                                                               55
          DOWN(X,Y)                                                               56
          BOARD<X,Y>    =                                        :(RETURN)        57
*
COUNT     SUM      =  SUM  +  1                                  :(RETURN)        58
*
END                                                                               59
```

Given 2 as an input value, this program produces the following output.

```
    STATEMENT 12: LEVEL 0 CALL OF AXIS(0,0),TIME = 100
    STATEMENT 19: LEVEL 1 CALL OF AXIS(1,0),TIME = 117
    STATEMENT 15: LEVEL 2 CALL OF COUNT(''),TIME = 117
    STATEMENT 58: LEVEL 2 RETURN OF COUNT = '',TIME = 117
    STATEMENT 15: LEVEL 1 RETURN OF AXIS = '',TIME = 117
    STATEMENT 20: LEVEL 1 CALL OF UP(1,0),TIME = 133
    STATEMENT 36: LEVEL 2 CALL OF RIGHT(1,1),TIME = 133
    STATEMENT 23: LEVEL 3 CALL OF COUNT(''),TIME = 150
    STATEMENT 58: LEVEL 3 RETURN OF COUNT = '',TIME = 150
    STATEMENT 23: LEVEL 2 RETURN OF RIGHT = '',TIME = 150
    STATEMENT 37: LEVEL 2 CALL OF UP(1,1),TIME = 150
    STATEMENT 32: LEVEL 3 CALL OF COUNT(''),TIME = 166
    STATEMENT 58: LEVEL 3 RETURN OF COUNT = '',TIME = 166
    STATEMENT 32: LEVEL 2 RETURN OF UP = '',TIME = 166
    STATEMENT 38: LEVEL 2 CALL OF LEFT(1,1),TIME = 183
    STATEMENT 45: LEVEL 3 CALL OF LEFT(0,1),TIME = 183
    STATEMENT 45: LEVEL 4 CALL OF LEFT(-1,1),TIME = 200
    STATEMENT 41: LEVEL 5 CALL OF COUNT(''),TIME = 200
    STATEMENT 58: LEVEL 5 RETURN OF COUNT = '',TIME = 200
    STATEMENT 41: LEVEL 4 RETURN OF LEFT = '',TIME = 200
    STATEMENT 46: LEVEL 4 CALL OF UP(-1,1),TIME = 216
    STATEMENT 32: LEVEL 5 CALL OF COUNT(''),TIME = 216
    STATEMENT 58: LEVEL 5 RETURN OF COUNT = '',TIME = 216
    STATEMENT 32: LEVEL 4 RETURN OF UP = '',TIME = 233
    STATEMENT 47: LEVEL 4 CALL OF DOWN(-1,1),TIME = 233
    STATEMENT 51: LEVEL 4 FRETURN OF DOWN,TIME = 233
    STATEMENT 48: LEVEL 3 RETURN OF LEFT = '',TIME = 250
    STATEMENT 46: LEVEL 3 CALL OF UP(0,1),TIME = 250
    STATEMENT 32: LEVEL 4 CALL OF COUNT(''),TIME = 250
    STATEMENT 58: LEVEL 4 RETURN OF COUNT = '',TIME = 266
    STATEMENT 32: LEVEL 3 RETURN OF UP = '',TIME = 266
    STATEMENT 47: LEVEL 3 CALL OF DOWN(0,1),TIME = 283
    STATEMENT 51: LEVEL 3 FRETURN OF DOWN,TIME = 283
    STATEMENT 48: LEVEL 2 RETURN OF LEFT = '',TIME = 283
    STATEMENT 39: LEVEL 1 RETURN OF UP = '',TIME = 283
    STATEMENT 21: LEVEL 0 RETURN OF AXIS = '',TIME = 300
THERE ARE 6 SYMMETRIC BISECTIONS OF A 4 BY 4 CHECKERBOARD
```

3. <u>Label Tracing</u>

```
   TRACE(name,'LABEL')
```

causes trace printout whenever transfer is made to the name. No printout occurs if the statement labelled with the name is flowed into, or is entered as a function entry point.

The following program, which converts numbers from hexadecimal form to decimal form, illustrates label tracing.

```
             &TRACE   =   1000                                                 1
             TRACE('DEHEX1','LABEL')                                           2
*
             DEFINE('DEHEX(STR)NO')                          :(DEHEX.END)      3
*
*
DEHEX        STR  POS(0) SPAN('0')   =                                         4
DEHEX1       STR   LEN(1) . NO    =                          :F(RETURN)        5
             DEHEX  =  INTEGER(NO)   16 * DEHEX + NO         :S(DEHEX1)        6
             'ABCDEF'    BREAK(NO) . NO                      :F(FRETURN)       7
             DEHEX = 16 * DEHEX + 10 + SIZE(NO)              :(DEHEX1)         8
DEHEX.END                                                                      9
*
*
READ         NUMBER  =  TRIM(INPUT)                          :F(END)          10
             OUTPUT                                                           11
.            =  'DEHEX(' NUMBER ') = ' DEHEX(NUMBER)                          11
.                                                            :S(READ)         11
             OUTPUT  =  'UNABLE TO CONVERT ' NUMBER          :(READ)          12
END                                                                           13
```

Typical printout from this program is

```
     STATEMENT 6: TRANSFER TO DEHEX1,TIME = 17
     STATEMENT 6: TRANSFER TO DEHEX1,TIME = 17
     STATEMENT 6: TRANSFER TO DEHEX1,TIME = 34
DEHEX(100) = 256
     STATEMENT 6: TRANSFER TO DEHEX1,TIME = 50
DEHEX(000001) = 1
     STATEMENT 6: TRANSFER TO DEHEX1,TIME = 67
     STATEMENT 6: TRANSFER TO DEHEX1,TIME = 83
DEHEX(00011) = 17
     STATEMENT 8: TRANSFER TO DEHEX1,TIME = 100
DEHEX(000F) = 15
     STATEMENT 8: TRANSFER TO DEHEX1,TIME = 117
DEHEX(E) = 14
     STATEMENT 6: TRANSFER TO DEHEX1,TIME = 133
     STATEMENT 8: TRANSFER TO DEHEX1,TIME = 150
     STATEMENT 8: TRANSFER TO DEHEX1,TIME = 167
     STATEMENT 8: TRANSFER TO DEHEX1,TIME = 167
UNABLE TO CONVERT 1ABCG
     STATEMENT 6: TRANSFER TO DEHEX1,TIME = 183
     STATEMENT 8: TRANSFER TO DEHEX1,TIME = 200
     STATEMENT 8: TRANSFER TO DEHEX1,TIME = 200
     STATEMENT 8: TRANSFER TO DEHEX1,TIME = 217
```

```
DEHEX(1ABC) = 6844
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 233
DEHEX(000E) = 14
    STATEMENT 6: TRANSFER TO DEHEX1,TIME = 250
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 266
DEHEX(001E) = 30
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 283
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 300
DEHEX(00EC) = 236
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 316
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 333
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 333
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 350
DEHEX(000FACE) = 64206
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 450
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 466
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 466
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 483
    STATEMENT 8: TRANSFER TO DEHEX1,TIME = 483
DEHEX(AAAAA) = 699050
```

4. Keyword Tracing

```
       TRACE(name,'KEYWORD')
```

causes trace printout when the value of the named keyword is changed. Only three keywords can be traced: STCOUNT, STFCOUNT, and FNCLEVEL. The following program, which converts numbers from decimal to hexadecimal form, illustrates keyword tracing.

```
              &TRACE   =    1000                                              1
              TRACE('STFCOUNT','KEYWORD')                                     2
*
              DEFINE('HEXER(N)Q,R')                                           3
              HEGITS = '0123456789ABCDEF'                                     4
*
*
READ          NUM = TRIM(INPUT)                             :F(END)           5
              OUTPUT = 'HEXER(' NUM ') = ' HEXER(NUM)       :S(READ)          6
              OUTPUT = 'UNABLE TO CONVERT ' NUM             :(READ)           7
*
*
HEXER         INTEGER(N)                                    :F(FRETURN)       8
              Q  = GT(N,15)  N / 16                         :F(HEX.END)       9
              R = N - Q * 16                                                  10
              N = Q                                                           11
              HEGITS   LEN(R) LEN(1) . R                    :F(FRETURN)       12
              HEXER = R HEXER                               :(HEXER)          13
HEX.END       HEGITS  LEN(N) LEN(1) . R                     :F(FRETURN)       14
              HEXER = R HEXER                               :(RETURN)         15
END                                                                           16
```

Typical printout from this program is

```
    STATEMENT 9: &STFCOUNT = 1,TIME = 0
HEXER(1) = 1
    STATEMENT 9: &STFCOUNT = 2,TIME = 33
```

```
HEXER(17) = 11
    STATEMENT 9: &STFCOUNT = 3,TIME = 49
HEXER(15) = F
    STATEMENT 9: &STFCOUNT = 4,TIME = 66
HEXER(14) = E
    STATEMENT 9: &STFCOUNT = 5,TIME = 99
HEXER(6844) = 1ABC
    STATEMENT 8: &STFCOUNT = 6,TIME = 116
    STATEMENT 6: &STFCOUNT = 7,TIME = 116
UNABLE TO CONVERT 1239.0003
    STATEMENT 9: &STFCOUNT = 8,TIME = 166
HEXER(30) = 1E
    STATEMENT 9: &STFCOUNT = 9,TIME = 183
HEXER(236) = EC
    STATEMENT 9: &STFCOUNT = 10,TIME = 216
HEXER(64206) = FACE
    STATEMENT 9: &STFCOUNT = 11,TIME = 249
HEXER(699050) = AAAAA
    STATEMENT 9: &STFCOUNT = 12,TIME = 266
HEXER(13) = D
    STATEMENT 9: &STFCOUNT = 13,TIME = 299
HEXER(0) = 0
    STATEMENT 9: &STFCOUNT = 14,TIME = 316
HEXER(000) = 0
    STATEMENT 9: &STFCOUNT = 15,TIME = 349
HEXER(128) = 80
    STATEMENT 9: &STFCOUNT = 16,TIME = 366
HEXER(256) = 100
    STATEMENT 9: &STFCOUNT = 17,TIME = 416
HEXER(123456789) = 75BCD15
    STATEMENT 5: &STFCOUNT = 18,TIME = 648
```

5. Discontinuation of Tracing

Tracing is a global condition that depends on the value of &TRACE. Regardless of trace requests made through the TRACE function, there is no trace output if &TRACE is off. The value of &TRACE may be set to zero explicitly, or may reach zero as it is decremented as the result of tracing. Individual trace associations may be cancelled, however, by executing

```
STOPTR(name,type)
```

which cancels a single trace association for the name and type. Thus the tracing of statement failure is stopped by executing

```
STOPTR('STFCOUNT','KEYWORD')
```

B. Programmer-Defined Trace Functions

The TRACE function has an optional fourth argument that permits the programmer to supply procedures for tracing. The form of the function is

```
TRACE(name,type,tag,function)
```

where the function is a programmer-defined function.

When the traced action occurs, the function is called with the name as its first argument and the tag as its second argument. Thus the programmer may define trace procedures to supplement the standard ones. The keyword &TRACE is turned off on entry to a programmer-defined trace procedure and restored on return. This prevents accidental tracing of a trace procedure. The programmer may turn the &TRACE keyword on while in a trace procedure.

1. Invoking Programmer-Defined Trace Procedures

The exact time at which a programmer-defined trace procedure is called depends on the type of trace.

VALUE: just after assignment of the new value

CALL: just after evaluation of the arguments, but before execution of the first statement in the function

RETURN: just before the return is made

FUNCTION: as for CALL and RETURN

LABEL: just before transfer to the label

KEYWORD: just after the keyword is changed

2. Tools for Writing Programmer-Defined Trace Procedures

Special information is required for writing more elaborate programmer-defined trace procedures. Three keywords and three functions are provided expressly for this purpose.

1. &STNO is a protected keyword whose value is the statement number of the statement currently being executed.

2. &LASTNO is a protected keyword whose value is the statement number of the last statement executed.

3. &RTNTYPE is a protected keyword whose value is the type of return (RETURN, FRETURN, or NRETURN) made by the last defined function to return.

4. ARG(function,n) is a function whose value is the name of the nth argument of the programmer-defined function. ARG is useful in writing programmer-defined trace procedures that trace several functions and need to determine the names of the formal arguments of the functions being traced.

5. LOCAL(function,n) is a function whose value is the name of the nth local variable of the defined function.

6. FIELD(data type,n) is a function whose value is the name of the nth field of the programmer-defined data type.

The following example illustrates a programmer-defined function, VALTR, that prints a trace output only when a traced variable is assigned a specified value. KEY is a global variable. Trace output only occurs when a traced variable is assigned the value of KEY. If the variable being traced is not a string, the tag is used in the printed output. Use of this function is illustrated in the following program which produces trace output when certain variables are assigned the value 25.

```
          POWER    =   ARRAY('25,5')                                               1
     *
          KEY   =   25                                                             2
     *
          DEFINE('VALTR(VAR,TAG)ST,TIME')                                          3
          &TRACE    =   1000                                                       4
          TRACE('I','VALUE',,'VALTR')                                              5
          TRACE(.POWER<5,2>,'VALUE',' 5 ** 2','VALTR')                             6
          TRACE(.POWER<25,1>,'VALUE',' 25 ** 1','VALTR')                           7
     *
     *    SET UP MATRIX OF INTEGER POWERS
     *
          J   =   1                                                                8
     NEXTI I   =   0                                                               9
     NEXTP I   =   I + 1                                                          10
          POWER<I,J>   =   I ** J                          :S(NEXTP)              11
          J   =   LT(J,5) J + 1                            :S(NEXTI)F(END)        12
     *
     VALTR ST   =   &LASTNO                                                       13
          TIME   =   TIME()                                                       14
          IDENT($VAR,KEY)                                  :F(RETURN)             15
          TAG   =   IDENT(DATATYPE(VAR),'STRING') VAR                             16
          OUTPUT   =   'KEY VALUE "' KEY '" ASSIGNED TO ' TAG                     17
     +                 ' IN STATEMENT ' ST ' AT TIME ' TIME                       17
     +                                                     :(RETURN)              17
     END                                                                          18
```

The printed output is

```
KEY VALUE "25" ASSIGNED TO I IN STATEMENT 10 AT TIME 166
KEY VALUE "25" ASSIGNED TO  25 ** 1 IN STATEMENT 11 AT TIME 166
KEY VALUE "25" ASSIGNED TO  5 ** 2 IN STATEMENT 11 AT TIME 199
KEY VALUE "25" ASSIGNED TO I IN STATEMENT 10 AT TIME 266
KEY VALUE "25" ASSIGNED TO I IN STATEMENT 10 AT TIME 332
KEY VALUE "25" ASSIGNED TO I IN STATEMENT 10 AT TIME 416
KEY VALUE "25" ASSIGNED TO I IN STATEMENT 10 AT TIME 499
```

Chapter 8. Input and Output

Input and output are accomplished by associating variables with data sets (files). In the case of a variable associated in the output sense, each time the variable is assigned a value, a copy of the value is put out onto the associated data set. In the case of a variable associated in the input sense, each time the value of the variable is used, a new value is read from the associated data set and becomes the new value of the variable. Thus input and output go on during program execution without any explicit I/O statements, as a result of I/O associations. Variables having standard associations are described in the following sections.

A. Printed Output

The variable OUTPUT is associated with the standard print data set (usually the printer). Consequently, whenever OUTPUT is assigned a value, printout is generated. For example,

```
      OUTPUT = 'THE SELECTED VALUES ARE'
```

produces the output

```
THE SELECTED VALUES ARE
```

Output may also result from value assignment specified in patterns. For example,

```
      PEXP = BAL . EXP1 . OUTPUT '+' BAL . EXP2 . OUTPUT
           .
           .
           .
      EXP PEXP
```

prints the two terms in EXP, and assigns their values to EXP1 and EXP2. This type of output is often useful for diagnostic purposes, and does not affect the pattern matching or the assignments made to EXP1 and EXP2.

Ordinary printout is printed 131 characters per line, with as many lines as necessary being generated. The null string is treated as a blank character and a blank line is printed for it. Strings are usually assigned to output variables. Integers and real numbers assigned to an output variable are automatically converted to strings. If an array is assigned to an output variable, the printed output is ARRAY with the prototype of the array enclosed in parentheses. For example, the statements

```
        MATRIX    =    ARRAY('-2:2,-3:3',0)
        OUTPUT    =    MATRIX
```

print

```
ARRAY('-2:2,-3:3')
```

If the prototype is longer than twenty characters, only the string ARRAY is printed. If an object with any other data type is assigned to an output variable, the formal identification of its data type is printed. For example,

```
        OUTPUT    =    LEN(7)
```

prints

```
PATTERN
```

B. Punched Output

The variable PUNCH is associated with the standard punch data set. Consequently, whenever PUNCH is assigned a value, a punched card is generated. For example,

```
        PUNCH    =    0
```

produces a card with zero punched in column one.

All the remarks about print output apply to punch output, except that 80 characters are punched per card, with additional cards punched as necessary for longer strings. The cards have no sequence numbering or identification unless provided in the strings which are punched.

C. Input

The variable INPUT is associated with the standard input data set. Whenever the value of INPUT is used, a card image is read from the input stream and becomes the new value of INPUT. For example,

```
        OUTPUT    =    INPUT
```

reads a card image and prints it. Similarly,

```
        TRIM(INPUT)    BAL . EXP
```

reads a card image and matches for a balanced string. All eighty columns of the card images are read, and the value of INPUT is an eighty character string.

Since each use of INPUT reads a card image, previous values of INPUT are lost unless they are assigned to other variables.

If the end of the input data set is encountered when a value of INPUT is requested, failure results. This failure can be used to detect the end of an input data set. For example,

```
       I   =   1
READ   DATA<I>   =   INPUT                              :F(OUT)
       I   =   I + 1                                    :(READ)
OUT
```

reads card images into the array DATA until the input data stream is exhausted (or I exceeds the range of DATA). Control is then transferred to OUT.

D. The I/O System

All input/output is handled by FORTRAN IV I/O routines. That is, SNOBOL4 I/O is done by the same system that does I/O for FORTRAN IV object programs. Consequently, the conventions and I/O concepts specified for the FORTRAN IV language also apply to SNOBOL4. In addition, the version of the language described here operates under OS/360. It is necessary to understand both the fundamentals of FORTRAN IV I/O [7,8] and job control language (JCL) [9] in order to use the I/O facilities of SNOBOL4 effectively.

In FORTRAN, data sets (files) have numbers (data set reference numbers). These numbers are referred to in source-language programs and are associated with specific data sets by JCL at run time. There are three standard data sets:

normal input stream (5)

normal print output (6)

normal punch output (7)

DDNAMEs in JCL are used to associate the data set reference numbers with actual data sets. DDNAMEs for FORTRAN have the form FTxxFyyy, where xx corresponds to the data set reference number and yyy is a file sequence number for multifile data sets. The typical DD cards used in SNOBOL4 associate the standard data set reference numbers 5, 6, and 7 as follows:

```
//FT06F001 DD SYSOUT=A
//FT07F001 DD UNIT=SYSCP
//FT05F001 DD *
```

This JCL, or its equivalent, is contained in the SNOBOL4 cataloged procedure, and is supplied automatically when the cataloged procedure is used.

A wide range of devices and record structures can be specified on DD cards. By changing the DD cards, the data streams can be assigned to different data sets at run time. Thus,

```
//FT05F001 DD DSNAME=PROG1,VOLUME=SER=BTLXX1,                        X
//             UNIT=DISK,DISP=OLD
```

specifies an input stream from a data set PROG1 on a disk file. Similarly,

```
//FT07F001 DD DSNAME=PUNCHER,VOLUME=SER=MYSAV1,UNIT=TAPE,              C
//            LABEL=(1,SL),DISP=(NEW,PASS),                            C
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
```

causes punched output to go onto a 9-track tape with a blocking factor of 10.

A complete discussion of DD statements is beyond the scope of this manual, and is a very involved and difficult subject. The important fact is that JCL permits the specification of a wide variety of devices and record structures. This specification is made when the program is run and requires no alteration of the program.

The FORTRAN I/O used in SNOBOL4 only handles sequential data sets. In particular, it cannot handle members of partitioned data sets.

FORTRAN supports multifile data sets. The last three characters in the DDNAME specify the file number. When FORTRAN comes to the end of a file, it automatically opens the next file of the same data set reference number.

Thus, for example, input may come from several files:

```
//FT05F002 DD DSNAME=DATA2,UNIT=DISK,VOLUME=SER=BTLH04,DISP=OLD
//FT05F001 DD *
       .
       .
       .
/*
```

With these DD cards, after the in-line data stream is exhausted, records are read from DATA2. The failure which occurs when an end of a data set is reached must be taken into account in programming.

E. Output Associations

The variables OUTPUT and PUNCH have predefined output associations. Programmer-defined associations may be made using the function OUTPUT. The form of the function is

```
OUTPUT(name,number,format)
```

OUTPUT associates the name with the data set reference number according to the given format. The format is a string specifying a FORTRAN IV format. The following statements correspond to the associations for the variables OUTPUT and PUNCH:

```
OUTPUT('OUTPUT',6,'(1X,131A1)')
OUTPUT('PUNCH',7,'(80A1)')
```

Using the OUTPUT function, any variable can be associated with any data set reference number. For example,

```
        PRFORM    =    '(1X,131A1)'
        TEST    =    ARRAY('8,8')
        OUTPUT(.TEST<1,1>,6,PRFORM)
        OUTPUT(.TEST<8,8>,6,PRFORM)
```

associate the array elements TEST<1,1> and TEST<8,8> with the ordinary print data set and with the standard print format. As a result, whenever either TEST<1,1> or TEST<8,8> is assigned a value, the new value is printed.

Data set reference numbers are not restricted to 5, 6, and 7, but can range from 1 through 99. Associations can be made with data set reference numbers other than the standard ones. In this case, a DD statement for that number must be provided when the program is run.

```
        OUTPUT('TEXT',7,'(80A1)')
```

associates TEXT with the punch data set. On the other hand,

```
        OUTPUT('TEXT',20,'(80A1)')
```

and the DD statement

```
//FT20F001 DD DSNAME=NEWF,UNIT=TAPE,VOLUME=SER=MYSAV1,                 X
//             LABEL=(2,SL),DISP=(NEW,PASS),DCB=(RECFM=FB,             X
//             BLKSIZE=800,LRECL=80)
```

allow the program to put card images onto the second file of a tape. The LRECL parameter of 80 and the format (80A1) relate the record size of the file to the record size in the format.

Formats used in output association must specify the conversion of at least one element by A-conversion. (Normally nA1-conversion is used.) Integers are converted into strings and I-conversion must not be used. In addition to A-conversion, quoted literals, X-, H-, T-, and Z-conversion may be specified [7,8]. Carriage control must be provided for printing; otherwise the first character of the string is consumed for this purpose. Consider

```
        OUTPUT('TITLE',6,'(1H1,131A1/(1X,131A1))')
```

When a value is assigned to TITLE, a page is ejected and the value titles the next page of output. The use of literals is illustrated by

```
        OUTPUT('SUM',6,"(' SUM=',127A1/(1X,131A1))")
```

which includes identifying information with the format. Subsequently,

```
        SUM    =    300
```

causes the printout

```
SUM=300
```

The predefined associations can be changed. Thus,

```
        OUTPUT('OUTPUT',6,'(1X,120A1)')
```

shortens the line length for OUTPUT to 120 characters.

F. Input Associations

Programmer-defined input associations can be made using the function INPUT. The form of this function is

```
        INPUT(name,number,length)
```

INPUT associates the name with the data set reference number, and specifies that the resulting string is to have the given length. (Notice in particular that no format is specified.) INPUT has a predefined association equivalent to

```
        INPUT('INPUT',5,80)
```

The specified length has some special properties. If the length is less than the record size on the data set being read, the last part of the record is lost. Hence,

```
        INPUT('INPUT',5,72)
```

changes the association for INPUT so that only 72 columns are read. Columns 73 through 80 are lost if data set reference number 5 is associated with ordinary card input. A length longer than the record size should not be specified.

G. Other I/O Functions

Several other functions are provided for I/O-related operations [7,8]. All of these functions return the null string as value.

1. DETACH

DETACH(name) removes any input and output association which the name may have. For example,

```
        DETACH('OUTPUT')
```

terminates normal print output.

2. <u>ENDFILE</u>

ENDFILE(number) writes an end of file on (closes) the data set specified by the number. For example,

```
ENDFILE(20)
```

closes the data set associated with data set reference number 20.

3. <u>REWIND</u>

REWIND(number) repositions the data set associated with the number to the first file. For example,

```
REWIND(10)
```

rewinds the data set associated with data set reference number 10. Subsequently, reference to 10 refers to the beginning of the data set specified by FT10F001 (even if 10 is a multifile data set).

4. <u>BACKSPACE</u>

BACKSPACE(number) backspaces one record on the data set associated with the number.

Chapter 9. Structure of a SNOBOL4 Run

A SNOBOL4 run consists of three distinguishable parts:

1) compilation,
2) execution, and
3) termination.

A. Compilation

During compilation, the SNOBOL4 system is initialized and the source program is compiled into an intermediate object code in a form suitable for interpretation during program execution. Compilation uses the same processes as conversion of a string to object code using the CODE function. Additional processes are involved in the reading of lines to be compiled from the input data set, printing of a source listing on an output data set, and noting errors in the source program.

1. Source Program Input

Input to the compiler comes from the standard input stream associated with data set reference number 5. The compiler begins reading program from the data set associated with FT05F001. Only 72 characters per line are read, so that columns 73-80 of card-image input may be used for sequential numbering. The compiler continues to read until it encounters the end statement. If an end of file is encountered before the end statement is found, the compiler goes to the next file for reference number 5. The input program therefore may be in several sections given by FT05F001, FT05F002, etc.

2. Source Listing

The listing of the program with sequential statement numbers goes on the standard print output. When the end statement is encountered, the compilation process stops. A listing of the compilation and placement of statement numbers can be controlled by control lines. A minus sign at the beginning of a line identifies a control line. Program listing is suppressed by the control line

```
-UNLIST
```

Program listing is restored by the control line

```
-LIST
```

The normal positioning of statement numbers is at the right side of the source listing. Statement numbers optionally may be placed at the left side of the listing. The control line

```
-LIST LEFT
```

changes statement numbering to the left. Right positioning of the statement numbers is restored by

```
-LIST RIGHT
```

or simply

```
-LIST
```

Blanks may appear between the minus sign and LIST or UNLIST. One or more blanks must appear between the LIST and the LEFT or RIGHT. Any characters other than LEFT following blanks on the LIST control line cause the same action as RIGHT. An erroneous control line is ignored.

3. Errors Detected during Compilation

Certain kinds of errors in the source program are detected during compilation. When an error is detected in a statement, compilation of that statement is terminated and an error message is printed below the statement, describing the nature of the error. A list of compilation error messages is given in Appendix B. A marker pointing to the vicinity of the error is also printed. This marker may be somewhat before or after the error, depending on the nature of the error. Since compilation of a statement stops when an error is encountered, only the first error in any one statement is detected. Compilation continues in spite of erroneous statements. However, if more than fifty erroneous statements are found, error termination occurs and the program is not executed.

B. Execution

Execution of the compiled object code begins when compilation is complete. Ordinarily, program execution begins with the first statement of the program. Program execution may be started at any labelled statement by specifying that label in the end statement. The label of the first statement to be executed is placed in the position of the subject. For example,

```
END   INIT
```

causes program execution to begin with the statement labelled INIT .

Data read from the standard input source begins with the first line after the end statement. Data printed during execution follows the source listing.

C. Termination

Upon termination, a statistics summary is printed to provide timing information and counts of certain program operations. If the keyword &DUMP is on at program termination, a dump of natural variables and unprotected keywords is also provided. Only natural variables with nonnull values are included. If the value of a variable is not a string, the same representation of the value is given as would be given if the value were printed as the result of an output association.

There are four kinds of termination:

1) normal,
2) error,
3) intervention, and
4) catastrophic.

1. Normal Termination

Normal termination occurs when the program transfers to END or flows into the end statement. The number of the last statement executed and the function level are printed. The following program illustrates the printout produced by a program that terminates normally.

```
SNOBOL4 (VERSION 2.0, OCT. 7, 1968)
BELL TELEPHONE LABORATORIES, INCORPORATED


          &DUMP    =    1                                                  1
    *          THIS PROGRAM IS THE ALGORITHM BY HAO WANG (CF. 'TOWARD
    *     MECHANICAL MATHEMATICS', IBM JOURNAL OF RESEARCH AND
    *     DEVELOPMENT 4(1) JAN 1960 PP.2-22.)  FOR A PROOF-DECISION
    *     PROCEDURE FOR THE PROPOSITIONAL CALCULUS.  IT PRINTS OUT A
    *     PROOF OR DISPROOF ACCORDING AS A GIVEN FORMULA IS A THEOREM
    *     OR NOT.  THE ALGORITHM USES SEQUENTS WHICH CONSIST OF TWO
    *     LISTS OF FORMULAS SEPARATED BY AN ARROW (--*).  INITIALLY, FOR
    *     A GIVEN FORMULA F THE SEQUENT
    *
    *               --* F
    *
    *     IS FORMED.  WANG HAS DEFINED RULES FOR SIMPLIFYING A FORMULA
    *     IN A SEQUENT BY REMOVING THE MAIN CONNECTIVE AND THEN
    *     GENERATING A NEW SEQUENT OR SEQUENTS.  THERE IS A TERMINAL
    *     TEST FOR A SEQUENT CONSISTING OF ONLY ATOMIC FORMULAS:
    *
    *          A SEQUENT CONSISTING OF ONLY ATOMIC FORMULAS IS VALID IF
    *          THE TWO LISTS OF FORMULAS HAVE A FORMULA IN COMMON.
    *
    *     BY REPEATED APPLICATION OF THE RULES, ONE IS LED TO A SET OF
    *     SEQUENTS CONSISTING OF ATOMIC FORMULAS.  IF EACH ONE OF THESE
    *     SEQUENTS IS VALID THEN SO IS THE ORIGINAL FORMULA.
    *
    *
    *
         UNOP      =  'NOT'                                                 2
         BINOP     =  'AND' | 'IMP' | 'OR' | 'EQU'                          3
         FORMULA  =  ' ' UNOP . OP '(' BAL . PHI ')'  |                     4
                     ' ' BINOP . OP '(' BAL . PHI ',' BAL . PSI ')'         4
         ATOM       =   ( ' ' BAL ' ' ) . A                                    5
*
         DEFINE('WANG(ANTE,CONSEQ)PHI,PSI')                                    6
*
READ     EXP        =   TRIM(INPUT)                   :F(END)                  7
         OUTPUT     =                                                          8
         OUTPUT     =   'FORMULA: ' EXP                                        9
         OUTPUT     =                                                         10
*
         WANG(,' ' EXP)                               :F(INVALID)             11
         OUTPUT     =   'VALID'                       :(READ)                 12
INVALID  OUTPUT     =   'NOT VALID'                   :(READ)                 13
*
WANG     OUTPUT     =  ANTE ' --* ' CONSEQ                                    14
         ANTE       FORMULA  =                        :S( $('A' OP) )         15
         CONSEQ     FORMULA  =                        :S( $('C' OP) )         16
         ANTE       =  ANTE  ' '                                              17
         CONSEQ     =  ' '  CONSEQ  ' '                                       18
TEST     ANTE       ATOM  =                           :F(FRETURN)             19
         CONSEQ     A                                 :S(RETURN) F(TEST)      20
*
*
ANOT     WANG(ANTE,CONSEQ ' ' PHI)                    :S(RETURN) F(FRETURN)   21
*
AAND     WANG(ANTE ' ' PHI ' ' PSI,CONSEQ)            :S(RETURN) F(FRETURN)   22
*
AOR      WANG(ANTE ' ' PHI,CONSEQ)                    :F(FRETURN)             23
         WANG(ANTE ' ' PSI,CONSEQ)                    :S(RETURN) F(FRETURN)   24
*
*
*
*
*
AIMP     WANG(ANTE ' ' PSI,CONSEQ)                    :F(FRETURN)             25
         WANG(ANTE,CONSEQ ' ' PHI)                    :S(RETURN) F(FRETURN)   26
*
AEQU     WANG(ANTE ' ' PHI ' ' PSI,CONSEQ)            :F(FRETURN)             27
         WANG(ANTE,CONSEQ ' ' PHI ' ' PSI)            :S(RETURN) F(FRETURN)   28
*
CNOT     WANG(ANTE ' ' PHI,CONSEQ)                    :S(RETURN) F(FRETURN)   29
*
CAND     WANG(ANTE,CONSEQ ' ' PHI)                    :F(FRETURN)             30
         WANG(ANTE,CONSEQ ' ' PSI)                    :S(RETURN) F(FRETURN)   31
*
COR      WANG(ANTE,CONSEQ ' ' PHI ' ' PSI)            :S(RETURN) F(FRETURN)   32
*
CIMP     WANG(ANTE  ' ' PHI,CONSEQ ' ' PSI)           :S(RETURN) F(FRETURN)   33
*
CEQU     WANG(ANTE ' ' PHI,CONSEQ ' ' PSI)            :F(FRETURN)             34
         WANG(ANTE ' ' PSI,CONSEQ ' ' PHI)            :S(RETURN) F(FRETURN)   35
END                                                                           36

NO ERRORS DETECTED DURING COMPILATION
```

```
FORMULA:  IMP(NOT(OR(P,Q)),NOT(P))

 --*  IMP(NOT(OR(P,Q)),NOT(P))
 NOT(OR(P,Q)) --*  NOT(P)
 --*  NOT(P) OR(P,Q)
 P --*  OR(P,Q)
 P --*  P Q
VALID

FORMULA:  NOT(IMP(NOT(OR(P,Q)),NOT(P)))

 --*  NOT(IMP(NOT(OR(P,Q)),NOT(P)))
 IMP(NOT(OR(P,Q)),NOT(P)) --*
 NOT(P) --*
 --*  P
NOT VALID

FORMULA:  IMP(AND(NOT(P),NOT(Q)),EQU(P,Q))

 --*  IMP(AND(NOT(P),NOT(Q)),EQU(P,Q))
 AND(NOT(P),NOT(Q)) --*  EQU(P,Q)
 NOT(P) NOT(Q) --*  EQU(P,Q)
 NOT(Q) --*  EQU(P,Q) P
 --*  EQU(P,Q) P Q
 P --*  P Q Q
 Q --*  P Q P
VALID

FORMULA:  IMP(IMP(OR(P,Q),OR(P,R)),OR(P,IMP(Q,R)))

 --*  IMP(IMP(OR(P,Q),OR(P,R)),OR(P,IMP(Q,R)))
 IMP(OR(P,Q),OR(P,R)) --*  OR(P,IMP(Q,R))
 OR(P,R) --*  OR(P,IMP(Q,R))
 P --*  OR(P,IMP(Q,R))
 P --*  P IMP(Q,R)
 P Q --*  P R
 R --*  OR(P,IMP(Q,R))
 R --*  P IMP(Q,R)
 R Q --*  P R
 --*  OR(P,IMP(Q,R)) OR(P,Q)
 --*  OR(P,Q) P IMP(Q,R)
 --*  P IMP(Q,R) P Q
 Q --*  P P Q R
VALID


NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS    7


DUMP OF VARIABLES AT TERMINATION

NATURAL VARIABLES

 A = ' Q '
 ABORT = PATTERN
 ARB = PATTERN
 ATOM = PATTERN
 BAL = PATTERN
 BINOP = PATTERN
 EXP = 'IMP(IMP(OR(P,Q),OR(P,R)),OR(P,IMP(Q,R)))'
 FAIL = PATTERN
 FENCE = PATTERN
 FORMULA = PATTERN
 INPUT = 'IMP(IMP(OR(P,Q),OR(P,R)),OR(P,IMP(Q,R)))
 OP = 'IMP'
 OUTPUT = 'VALID'
 REM = PATTERN
 SUCCEED = PATTERN
 UNOP = 'NOT'


UNPROTECTED KEYWORDS

 &ABEND = 0
 &ANCHOR = 0
 &DUMP = 1
 &FTRACE = 0
 &FULLSCAN = 0
 &MAXLNGTH = 5000
 &STLIMIT = 50000
 &TRACE = 0


SNOBOL4 STATISTICS SUMMARY-

           1331 MS. COMPILATION TIME
            550 MS. EXECUTION TIME
            162 STATEMENTS EXECUTED,      34 FAILED
              0 ARITHMETIC OPERATIONS PERFORMED
             63 PATTERN MATCHES PERFORMED
              0 REGENERATIONS OF DYNAMIC STORAGE
           3.40 MS. AVERAGE PER STATEMENT EXECUTED
```

2. Error Termination

Error termination occurs in case of a programming error or internal condition sufficiently serious to prevent continued execution. The statement number in which execution terminated and the function level are printed. An error message is printed indicating the cause of the termination. A listing of termination messages is given in Appendix B. Dumps and statistics are then printed as for normal termination.

The following program, from which the input data was removed, illustrates a typical listing resulting from error termination. Because input data is lacking, statement 1 fails and the array A is not formed. Subsequent reference to A as an array in statement 4 is erroneous.

```
SNOBOL4 (VERSION 2.0, OCT. 7, 1968)
BELL TELEPHONE LABORATORIES, INCORPORATED


     *   LET THE FIRST DATA CARD SPECIFY THE MAXIMUM NUMBER OF
     *   CARDS TO BE SORTED.  GENERATE AN ARRAY.
     *
              A  =  ARRAY(TRIM(INPUT))                             1
     *
     *   DEFINE THE FUNCTION INSERT.
     *
              DEFINE('INSERT(J)TEMP')                              2
     *
     *   READ THE CARDS INTO THE ARRAY.
     *
     INIT     I  =  I + 1                                          3
              A<I>   =   TRIM(INPUT)            :F(SORT)           4
              OUTPUT  =  A<I>                   :(INIT)            5
     *
     *   LET N BE THE NUMBER OF CARDS.  INITIALIZE THE INDEX AND
     *   THEN SORT.
     *
     SORT     N  =  I - 1                                          6
              I  =  1                                              7
     *
     *   COMPARE TWO SUCCESSIVE CARDS.
     *
     SORTA    LGT(A<I>,A<I + 1>)                :S(SORTC)          8
     *
     *   IF THEY ARE IN THE PROPER ORDER,INCREMENT THE INDEX
     *   (UNLESS SORTING IS FINISHED) AND CONTINUE.
     *
     SORTB    I  =  LT(I,N - 1)  I + 1          :S(SORTA)F(DONE)   9
     *
     *   OTHERWISE, INSERT THE CARD IN ITS PROPER PLACE.
     *
     SORTC    INSERT(I + 1)                     :(SORTB)           10
     *
     *   PUNCH SORTED CARDS.
     *
     DONE     I  =  1                                              11
              OUTPUT  =                                            12
     PUNCH    OUTPUT  =  A<I>                                      13
              PUNCH  =  OUTPUT                                     14
              I  =  LT(I,N)  I + 1              :S(PUNCH)F(END)    15
```

```
*
*   FUNCTION DEFINITION
*
INSERT    TEMP  =  A<J - 1>                                                     16
          A<J - 1>  =  A<J>                                                     17
          A<J>  =  TEMP                                                         18
          J  =  GT(J,2)  J - 1                       :F(RETURN)                 19
          LGT(A<J - 1>,A<J>)                         :S(INSERT)F(RETURN)        20
END                                                                             21
```

```
NO ERRORS DETECTED DURING COMPILATION

ERROR TERMINATION IN STATEMENT   4 AT LEVEL  0
ERRONEOUS ARRAY REFERENCE




SNOBOL4 STATISTICS SUMMARY-


           782 MS. COMPILATION TIME
           166 MS. EXECUTION TIME
             4 STATEMENTS EXECUTED,       1 FAILED
             1 ARITHMETIC OPERATIONS PERFORMED
             0 PATTERN MATCHES
             0 REGENERATIONS OF DYNAMIC STORAGE
         41.50 MS. AVERAGE PER STATEMENT
```

3. Intervention Termination

Intervention termination occurs when operator or system action terminates the run. This may occur if the run exceeds specified limits. If the SNOBOL4 system is able to regain control after intervention, the message "CUT BY SYSTEM IN STATEMENT n AT LEVEL m" is printed. Dumps and statistics are then printed as for normal termination.

The following program illustrates intervention termination resulting from failure to sense an end-of-file condition. On the IBM System/360, when the data on FT05F001 is exhausted, statement 3 fails. A subsequent read attempt results in an attempt to open FT05F002, the next file for data set reference number 5 (see Chapter 8). A second file is not intended or provided and the error message (IHC219I) is printed by the FORTRAN I/O routines. Control then returns to the SNOBOL4 system and run statistics are printed.

```
SNOBOL4 (VERSION 2.0, OCT. 7, 1968)
BELL TELEPHONE LABORATORIES, INCORPORATED


          NM    =  BREAK(',') . N ',' BREAK(' ') . M                          1
          DEFINE('C(N,M)')                                                    2
READ      INPUT   NM                                                          3
          OUTPUT   =   'C(' N ',' M ')=' C(N,M)     :(READ)                   4
*
C         M  =  LT(N - M,M)   N - M                                           5
          C  =  EQ(M,0)   1              :S(RETURN)                           6
          C  =  N * C(N - 1,M - 1) / M   :(RETURN)                            7
END                                                                           8


NO ERRORS DETECTED DURING COMPILATION



C(15,6)=5005
C(17,10)=19448
C(20,2)=190
C(25,24)=25
C(25,24)=25

IHC219I

TRACEBACK FOLLOWS-   ROUTINE    ISN   REG. 14

                     IBCOM            A60356BC

CUT BY SYSTEM IN STATEMENT   3 AT LEVEL  0


             233 MS. COMPILATION TIME
             483 MS. EXECUTION TIME
              74 STATEMENTS EXECUTED,       37 FAILED
             110 ARITHMETIC OPERATIONS PERFORMED
               4 PATTERN MATCHES
               0 REGENERATIONS OF DYNAMIC STORAGE
            6.53 MS. AVERAGE PER STATEMENT
```

On the IBM System/360, cancellation prevents the SNOBOL4 system from regaining control. A system completion code is given but no further output is printed. The following program illustrates such intervention termination. The program loops, since the function DELETE is called with a null string for CHAR. The system completion code 222 indicates cancellation which results, in this case, from exceeding a specified time limit.

```
    *
              DEFINE('DELETE(STRING,CHAR)')
    *
    READ      STRING   =   TRIM(INPUT)     :F(END)
              CHAR   =   TRIM(INPUT)       :F(ERR)
              OUTPUT   =   STRING
              OUTPUT   =   CHAR
              OUTPUT   =   DELETE(STRING,CHAR)
              OUTPUT   =   :(READ)
    *
    *       THIS FUNCTION DELETES OCCURRENCES OF A CHARACTER FROM A STRING
    *
    DELETE    STRING   CHAR   =      :S(DELETE)
              DELETE   =   STRING      :(RETURN)
    END
```

```
NO ERRORS DETECTED DURING COMPILATION


THE RATIO OF ATOMIC WEIGHTS OF THE TWO COMPOUNDS SUGGESTS  A RELATIONSHIP
I
THE RATO OF ATOMC WEGHTS OF THE TWO COMPOUNDS SUGGESTS  A RELATONSHP

ONE OF THE MORE COMMON OCCURRENCES IN EVERYDAY COMMUNICATION IS
O
NE F THE MRE CMMN CCURRENCES IN EVERYDAY CMMUNICATIN IS

THE FIRST OF THREE TUTORIAL LECTURES ON THE PRESENT STATE OF ART



COMPLETION CODE - SYSTEM=222  USER=000
```

4. <u>Catastrophic Termination</u>

Catastrophic termination occurs when system or machine malfunction causes a situation so serious that intervention termination is impossible. In the case of a catastrophic termination, there may be no indication of the source or cause of the termination. Print and punch output may be incomplete or lacking altogether.

Chapter 10. Programming Details

The preceding sections have presented, in varying degrees of detail, the language features of SNOBOL4. There remains a collection of odds and ends that may be of more or less interest and utility to the programmer. This section includes a number of such items, a potpourri whose ingredients may interest various individuals.

A. Efficiency and Good Programming Practices

When efficiency is considered, the basic criterion is the total amount of time required to execute the program. Execution time is most affected by the algorithm used and the structure of the program; both are beyond the scope of discussion here. A less significant, but often more tangible, measure of efficiency is the average amount of time required to execute program statements. If the algorithm and program structure are fixed, two reasonable goals are:

1) reducing the number of statements which have to be executed, and
2) reducing the average execution time per statement.

These goals generally conflict. The number of executed statements may be reduced by increasing their complexity, but the average execution time is increased. More can be said about the techniques for improving the efficiency of statement execution. Some considerations listed below suggest good practices for program organization and data representation.

Comparative timing figures are given in some cases. These figures are approximate; precise figures depend on the machine and program environment.

1. Efficiency in Pattern Matching

Many considerations involved in using patterns efficiently were discussed earlier. A few points deserve special emphasis.

Use anchored pattern matching if possible.

Many patterns can or should match only beginning at the first character of the subject string. This is often true of an entire program, in which the anchored mode can be set using &ANCHOR. While the anchored mode can be turned on and off, it is also possible to anchor a pattern by beginning it with FENCE or POS(0).

```
     Q  =  FENCE P
```

creates a pattern Q that is an anchored version of P.

It is worth remembering that pattern matching usually takes longest when the pattern fails to match. This is particularly true when the pattern is not anchored. Consider the two examples

```
        &ANCHOR   =   1
        ')))))))))))))))))))))' BAL
```

and

```
        &ANCHOR   =   0
        ')))))))))))))))))))))' BAL
```

The pattern match in the second example takes 9.96 times as long as in the first.

<u>Use BREAK instead of ARB if possible.</u>

The pattern resulting from BREAK(CS) is designed to stream quickly through a string looking for any character in CS. ARB, on the other hand, operates without any knowledge of what is expected to follow it. ARB first matches the null string. Then if the component beyond it fails, ARB matches one character, then two characters, and so on. As an example, consider the two patterns

```
        P1    =   BREAK(',') LEN(1)
        P2    =   ARB ','
```

In most cases, these two patterns match the same set of strings. Consider the two cases

```
        'ABCDEFGHIJKLMNOPQRSTUVWXYZ,' P1
```

and

```
        'ABCDEFGHIJKLMNOPQRSTUVWXYZ,' P2
```

The pattern match in the second case takes 9.88 times as long as in the first.

ARB has many legitimate uses and is essential in many cases. BREAK provides a more efficient way of performing commonly used matching operations.

<u>Use ANY instead of alternation if possible.</u>

The pattern resulting from ANY(CS) matches any character at a speed independent of the order of the characters in CS. In an explicit alternation of characters, alternatives are matched in order, and the time it takes to find a match depends on the order of the alternatives. Consider the patterns

```
        P1    =    ANY('ABCDEFGHIJKLM')
```

and

```
        P2    =    'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I'
+                  | 'J' | 'K' | 'L' | 'M'
```

Applied to several different characters, the statements

```
        C    P1
```

and

```
        C    P2
```

give the following results:

1. For C = 'A', P1 is 1.08 times as fast as P2.
2. For C = 'G', P1 is 2.44 times as fast as P2.
3. For C = 'M', P1 is 3.79 times as fast as P2.

The formation of the pattern for alternation also takes more time and space than that for ANY.

Avoid repetitious construction of patterns.

If possible, a pattern structure should be constructed once and assigned to a variable. An expression which appears as a pattern in a statement must be evaluated each time the statement is executed. This form of evaluation, which constructs the pattern over and over, consumes both time and space.

If

```
        P1    =    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

then for the statements

```
        N    P1
```

and

```
     N   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

the second statement takes 7.59 times as long as the first if N is 5. This comparison includes the effects of the additional space consumed when the second example is used repeatedly. Space consumed by patterns must eventually be reclaimed by storage regeneration, which adds to the running time of the program.

Do not use the fullscan mode unless necessary.

The fullscan mode, established by turning on the keyword &FULLSCAN, is useful in some more complicated and esoteric applications. Since the fullscan mode bypasses all heuristics, pattern matching may take much longer. Consider the pattern

```
     P   =   ARB ','
```

This pattern is usually inefficient, but serves particularly well to illustrate the effect of heuristics. In the case of

```
     &FULLSCAN    =    0
     'ABCDEFGHIJKLMNOPQRSTUVWXYZ'   P
```

and

```
     &FULLSCAN    =    1
     'ABCDEFGHIJKLMNOPQRSTUVWXYZ'   P
```

both examples fail to match, but the second takes 13.42 times as long to do so. This is an extreme example, and the great difference in timing is due to all the combinations that ARB goes through. In the first case, the heuristics make the pattern act as if it were anchored.

In general, the fullscan mode does not produce such marked effects. On the other hand, since statements similar to the examples above are likely to occur in the average program, it is well not to turn the fullscan mode on except for statements in which it is required.

Use conditional rather than immediate value assignment if possible.

Immediate value assignment forms a substring and generates a variable for every intermediate successful match for the pattern component with which it is associated. This is a time and space consuming process. For example, if

```
     EXP    =    'D*A/(B*C)+(D-B)*C'
     P1     =    BAL $ B1 '+' BAL $ B2
     P2     =    BAL . B1 '+' BAL . B2
```

then in

```
       EXP   P1
```

and

```
       EXP   P2
```

P2 is 1.37 times faster than P1.

2. Structuring Data

The most serious inefficiencies in SNOBOL4 programs are usually the result of awkward or cumbersome representation of data. Encoding data as long strings of symbols may be very inefficient. Furthermore, every modification of a string by concatenation or decomposition produces a new string which consumes storage. Matching may be quite slow. On the other hand, arrays and programmer-defined data objects permit a considerable range of data structures, and operations on such structures are usually relatively efficient.

Consider two representations of a list: one as an array of elements, and the other as a string of items separated by commas. Suppose the list elements are of the form

```
ACOMP
ACOMPC
ADREAL
AEQLC
AEQLIC
   .
   .
   .
```

Then the list represented by an array has the form

```
       LIST<1>   =   'ACOMP'
       LIST<2>   =   'ACOMPC'
       LIST<3>   =   'ADREAL'
       LIST<4>   =   'AEQLC'
       LIST<5>   =   'AEQLIC'
                 .
                 .
                 .
```

and the list represented by a string has the form

```
       LIST    =   'ACOMP,ACOMPC,ADREAL,AEQLC,AEQLIC,...'
```

The speed of operating on the list depends, of course, on the operations to be performed and the number of items on the list. Consider the problem of creating another list from LIST with the items in reverse order. This may be done for the two data representations by the following program segments.

```
        J    =    PROTOTYPE(LIST)
        I    =    1
        NLIST    =    COPY(LIST)
REV     NLIST<J>    =    LIST<I>                        :F(OUT)
        J    =    J - 1
        I    =    I + 1                                 :(REV)


        ITEMP    =    BREAK(',') . ITEM LEN(1)
        CLIST    =    LIST
        NLIST    =
REV     CLIST    ITEMP    =                             :F(OUT)
        NLIST    =    ITEM ',' NLIST                    :(REV)
```

For a typical list of ten items of the type shown, the reversal of the string representation takes 2.3 times as long as for the array representation. For a list of 100 items, the factor is 3.7.

There are pros and cons for both representations. Lists of varying or unknown length are easier to handle as strings. Pattern matching can also be used directly on the string representation to perform operations like finding duplicate elements. On the other hand, access to individual items in the array representation is much simpler. Consider the problem of isolating the 73rd of 100 items in the string representation.

B. Storage Management

Dynamic storage is continually allocated during program compilation and execution. All forms of programmer data reside in dynamic storage and compete for available space. This includes compiled program, strings, patterns, and so forth. Some data, depending on its use, is transient and may be discarded. Other data is always accessible to the program and must be kept. When dynamic storage is exhausted, storage is regenerated, collecting all needed data and deleting all data inaccessible to the program. This process occurs automatically, and ordinarily does not concern the programmer directly. Run statistics indicating a large number of storage regenerations suggest potential trouble, however. Continual reconstruction of patterns and manipulation of very long strings are the most common causes of frequent storage regeneration. Storage regeneration, although it may degrade execution speed, should not be a factor in program efficiency unless it occurs frequently.

1. Forcing Storage Regeneration

In special circumstances, a programmer may want to force storage regeneration. This is done with the function COLLECT which forces storage regeneration. COLLECT returns as value the amount of storage (in bytes on the IBM System/360) remaining free after regeneration. COLLECT(N) fails if less than N bytes remain after regeneration.

2. Clearing Variable Values

Some programs are organized to process several sections of data in order, necessitating removal of residual data between sections. The function CLEAR assists in this matter.

```
       CLEAR()
```

sets the values of all natural variables, including ARB, BAL, etc., to the null string. CLEAR does not affect the value of keywords, I/O associations, function definitions, the value of array elements, or the value of fields of defined data objects. Furthermore, variables are cleared only at the level at which CLEAR is called. This permits the values of selected variables to be saved at a lower level and then restored. For example, the selected variables can be made formal arguments to a function which calls CLEAR. If the values of X, Y, Z, and PAT are to be saved, a function RESET could have the defining statement

```
       DEFINE('RESET(X,Y,Z,PAT)')
```

with the procedure

```
RESET  CLEAR()  :(RETURN)
```

The values of X, Y, Z, PAT, and RESET are saved when RESET is called, and restored when it returns. All other natural variables are cleared. The values of primitive patterns can be restored using the values of the corresponding keywords, which are not affected by CLEAR. For example,

```
       ARB   =   &ARB
```

restores the orignal value of ARB .

References

1. Farber, D. J., R. E. Griswold, and I. P. Polonsky. "SNOBOL, A String Manipulation Language," Journal of the Association for Computing Machinery, Vol. 11, No. 1 (January, 1964), pp. 21-30.

2. Farber, D. J., R. E. Griswold, and I. P. Polonsky. "The SNOBOL3 Programming Language." Bell System Technical Journal, Vol XLV, No. 6 (July-August, 1966), pp. 895-944.

3. Forte, Allen. SNOBOL3 Primer, The MIT Press, Cambridge, Massachusetts, 1967.

4. Hammersley, J. M. and D. C. Handscomb. Monte Carlo Methods, Methuen & Co. Ltd., London, 1965, pp. 27-29.

5. Lukasiewicz, Jan. Aristotle's Syllogistic from the Standpoint of Modern Formal Logic, Clarendon Press, Oxford, England, 1951, p.78.

6. Burks, A. W., D. W. Warren, and J. B. Wright. "An Analysis of a Logical Machine Using Parenthesis-free Notation," Mathematical Tables and Other Aids to Computation, Vol. VIII, 1954, pp. 53-57.

7. IBM System/360 Operating System. FORTRAN IV [G] Programmer's Guide. Form C28-6639-1, International Business Machines Corporation, 1968.

8. IBM System/360 FORTRAN IV Language. Form C28-6515-5, International Business Machines Corporation, 1968.

9. IBM System/360 Operating System. Job Control Language. Form C28-6539-5, International Business Machines Corporation, 1968.

10. IBM System/360 PL/I Reference Manual. Form C28-8201-1, International Business Machines Corporation, 1968, pp. 197-198.

Appendix A. Syntax of SNOBOL4

This formal description of the syntax of SNOBOL4 is given in a syntax notation used in many IBM manuals [10]. Rules explaining this notation follow.

1) A class of elements is denoted by a notation variable, which consists of lower case letters and periods and must begin with a letter.

2) Literal characters are denoted by capital letters or special characters. Lower case letters and syntactic symbols are underlined when they represent literals. A lone underscore stands for itself.

3) A syntactic unit is defined as one of the following:

   a. a notation variable,

   b. literal characters, or

   c. any collection of variables, literals, and syntax notation surrounded by braces or brackets.

4) Braces { } denote a grouping.

5) Square brackets [ ] denote an option. Anything enclosed within brackets may appear or be omitted.

6) Vertical stacking of syntactic units and the vertical stroke | denote alternatives.

7) Three dots ... denote optional repetition of the immediately preceding syntactic unit one or more times.

8) Footnotes are used where restrictions apply to notation variables.

1. Syntax of SNOBOL4 Statements

The following notation variables define the components of a statement, leading to the definition of a statement itself.

```
digit:  0|1|2|3|4|5|6|7|8|9

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|
         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

alphanumeric:  letter|digit

identifier:  letter [alphanumeric|.|_]...

blanks:  one or more blank characters

integer:  digit [digit]...

real:  integer . [integer]

unary:  +|-|*|.|&|$|?|¬|@|#|%

binary:  blanks [[+|-|*|/|**|.|$|||@|#|%] blanks]

string:  zero or more EBCDIC characters

sliteral:  'string[1]'

dliteral:  "string[2]"

literal:  sliteral|dliteral|integer|real

element:  [unary]... ⎧identifier      ⎫
                     ⎪literal         ⎪
                     ⎨function.call   ⎬
                     ⎪array.ref       ⎪
                     ⎩( expression )  ⎭

operation:  element binary {element|expression}

expression: [blanks] [element|operation] [blanks]

arg.list:  expression [, expression]...

function.call:  identifier ( arg.list )

array.ref:  identifier < arg.list >

label:  {alphanumeric string[3]}[4]

subject.field:  blanks element

pattern.field:  blanks expression
```

----

[1]not including a single quote '
[2]not including a double quote "
[3]not including a blank or semicolon
[4]but not END

```
object.field:  blanks expression

equal:  blanks =

goto:  {( expression )|< expression >}

goto.field:  blanks : [blanks] ⎧goto                          ⎫
                               ⎨S goto [blanks] [F goto]⎬
                               ⎩F goto [blanks] [S goto]⎭

eol:  end of line

eos:  [blanks] {;|eol}

assign.statement:  [label] subject.field equal [object.field] [goto.field] eos

match.statement:  [label] subject.field pattern.field [goto.field] eos

repl.statement:  [label] subject.field pattern.field equal [object.field]
                 [goto.field] eos

degen.statement:  [label] [subject.field] [goto.field] eos

end.statement:  END [blanks [label|END]] eos

statement:  assign.statement|match.statement|repl.statement|degen.statement|
            end.statement
```

2. Syntax of SNOBOL4 Programs

A SNOBOL4 program consists of a sequence of statements terminating with an end statement. Interspersed among these statements may be comment lines and control lines.

```
comment.line:  * string eol

control.line:  - [blanks] ⎧LIST blanks [LEFT|RIGHT]⎫ [blanks] eol
                          ⎩UNLIST                  ⎭
```

A statement begins immediately following the preceding statement, i. e. at the beginning of a line or following a semicolon. A statement may be continued on the next line by using a continue line.

```
continue.line:  {+|.} remainder of statement
```

Comment, control, and continue lines must begin at the beginning of a line. They may not start in the interior of a line following a semicolon. A statement may be broken over a line boundary anywhere a blank is mandatory. If a statement has the form

```
part1 blanks part2
```

it may be continued as

```
part1 [blanks] eol {+|.}  [blanks] part2
```

where the + or . begins a new line, and takes the place of the mandatory blank.

3. Syntax of SNOBOL4 prototypes

Prototypes for arrays, programmer-defined functions, and programmer-defined data types are evaluated during program execution, not during compilation. These prototypes may be given explicitly as literals or may be computed in a variety of ways. When ARRAY, DEFINE, or DATA is executed, the corresponding prototype is then analyzed. The syntax of these prototypes follows.

```
identifier.list:  identifier [, identifier]...

data.prototype:  identifier ( identifier.list )

function.prototype:  identifier ( [identifier.list] ) [identifier.list]

signed.integer:  [[+|-] integer]

dimension:  signed.integer [: signed.integer]

array.prototype:  dimension [, dimension]...
```

Appendix B. Error Messages

1. Compilation Error Messages

1. BINARY OPERATOR MISSING OR IN ERROR

A binary operator is erroneous or a blank between expressions is missing. Some examples are

```
X    =   F(X)*** 2
TEXT   =   '('TEXT ')'
M    =   (A B)N
```

2. ERRONEOUS INTEGER

An integer which is too large appears in the source program. On the IBM System/360, the maximum integer is $2^{31}-1$.

3. ERRONEOUS LABEL

A label does not begin with a digit or letter.

4. ERRONEOUS OR MISSING BREAK CHARACTER

A break character appears in an erroneous context, or a nested expression is not closed. Some examples are

```
X    =   (A,B)
A<1,2)   =   5
F(G(X)                :S(L)
```

5. ERRONEOUS REAL NUMBER

A real number which is too large or too small appears in the source program. On the IBM System/360, the range of real numbers is on the order of 10^{-78} to 10^{75}.

6. ERRONEOUS SUBJECT

An erroneous construction occurs in the subject. An example is

```
,    =    2
```

7. ERROR IN GOTO

A syntactic error occurs in the goto field. Some examples are

```
:S(L1)S(L2)
:S<A)
:S(L1) L2
```

8. ILLEGAL CHARACTER IN ELEMENT

An illegal character occurs in a element. Some examples are

```
       Z    =    A+B
       X    =    3:
       E    =    3.25P
```

9. IMPROPERLY TERMINATED STATEMENT

A statement terminates before a construction is complete. An example is

```
       N    =    M +
```

10. PREVIOUSLY DEFINED LABEL

A label has occurred previously in the program. The first occurrence of the label holds, and subsequent occurrences are erroneous.

11. UNCLOSED LITERAL

The closing quote on a literal is missing. Some examples are

```
       LETTER   =   'A
       TEXT    =   'HE YELLED STOP"
```

2. <u>Error Termination Messages</u>

1. ARGUMENT NOT DEFINED FUNCTION.

The function argument to ARG or LOCAL was not the name of a programmer-defined function.

2. CALL OF UNDEFINED FUNCTION.

A call was made to a function or operation for which no definition exists.

3. ERRONEOUS ARRAY REFERENCE.

An array reference was made to an object that does not have data type ARRAY. That is, A<I> is erroneous if the value of A is not an array.

4. ERRONEOUS END STATEMENT.

The starting label given in the end statement did not occur in the program, or there was a syntactic error in the end statement.

5. ERRONEOUS PROTOTYPE.

A prototype in a call to ARRAY, DATA, or DEFINE had a syntax error.

6. ERROR IN ARITHMETIC OPERATION.

Overflow or an illegal operation (such as division by zero) occurred in an arithmetic operation.

7. ERROR IN COMPILER.

An error occurred in the SNOBOL4 compiler. The program listing should be sent to the authors.

8. ERROR IN INTERPRETER.

An error occurred in the SNOBOL4 interpreter. The program listing should be sent to the authors.

9. ERROR IN PATTERN MATCHING.

An error occurred in the SNOBOL4 pattern matching program. The program listing should be sent to the authors.

10. ERROR IN STORAGE REGENERATION.

A programming error occurred in the SNOBOL4 storage regeneration program. The program listing should be sent to the authors.

11. ERROR IN SUBROUTINE.

A programming error occurred in one of the SNOBOL4 subroutines. The program listing should be sent to the authors.

12. EXCEEDED LIMIT ON STATEMENT EXECUTION.

Too many statements were executed. See the keyword &STLIMIT.

13. EXCESSIVE COMPILATION ERRORS.

The number of compilation errors exceeded fifty. An excessive number of compilation errors is assumed to indicate a situation so serious that further processing should be discontinued.

14. EXECUTION OF STATEMENT WITH COMPILATION ERROR.

The program encountered a statement that contains a compilation error. Such statements may be reached either by a transfer or normal program flow.

15. FAILURE IN GOTO EVALUATION.

A function or operation called in the evaluation of a goto failed.

16. ILLEGAL DATA TYPE.

The data type of an object was incorrect for the operation that was to be performed on it. For example, this error termination results if an attempt is made to multiply an integer by a pattern.

17. ILLEGAL TRACE TYPE.

The second argument to TRACE or STOPTR is not one of the trace types VALUE, CALL, RETURN, FUNCTION, LABEL, or KEYWORD.

18. ILLEGAL UNIT DESIGNATION.

A negative number was given as a data set reference number in an input or output association.

19. IMPROPER STATEMENT TERMINATION.

During conversion from STRING to CODE, the string was exhausted without proper statement termination.

20. INCORRECT NUMBER OF ARGUMENTS.

A primitive function was called with too many arguments or an array reference has been made with too many indices. This error may also occur if too few arguments are supplied for a primitive function, but only if the primitive function is invoked by APPLY or through a synonym for the function.

21. INSUFFICIENT STORAGE TO CONTINUE.

Storage available to the SNOBOL4 system was inadequate for program execution to continue.

22. NEGATIVE NUMBER IN ILLEGAL CONTEXT.

A negative number was given as an argument to LEN, POS, INPUT, RPOS, RTAB, or TAB.

23. NULL STRING IN ILLEGAL CONTEXT.

The indirectness operator was applied to the null string, as

```
      Z   =   $()
```

24. OBJECT EXCEEDS SIZE LIMIT.

An attempt was made to form a data object which exceeds the internal limit on the size of structures. This limit is 65535 bytes on the IBM System/360.

25. OVERFLOW IN PATTERN MATCHING.

Internal storage used by the pattern matching program was exceeded. This condition is usually the result of excessive recursion in a pattern.

26. READING ERROR.

An error return occurred from an input operation.

27. RETURN FROM ZERO LEVEL.

A transfer to RETURN, FRETURN, or NRETURN occurred outside the call of a defined function.

28. STACK OVERFLOW.

The stack used by the SNOBOL4 system was exhausted. This condition is usually the result of excessive recursion in programmer-defined functions. Stack overflow may also occur during storage regeneration.

29. STRING OVERFLOW.

A string exceeded the maximum length set for strings. See the keyword &MAXLNGTH.

30. TOO MANY DATA TYPES.

The limit on the number of programmer-defined data types was exceeded. This limit is 899.

31. UNDEFINED OR ERRONEOUS GOTO.

An attempt was made to transfer to a label which does not occur in the program, or the result of evaluating a goto was not a natural variable.

32. UNKNOWN KEYWORD.

Reference was made to a nonexistent keyword.

33. VARIABLE NOT GIVEN WHERE REQUIRED.

An object with only a value, not a name, has occurred where a name is required. Examples are

```
        SIZE(A)    =    3
```

and

```
                        :(TRIM(END))
```

3. Print Request Messages

Trace printout and the dump of natural variables following termination require the construction of strings whose lengths depend on the values involved. A fixed amount of space is available for such messages and in some cases this space may not be large enough to form the required string. In these cases, the message

```
***PRINT REQUEST TOO LONG***
```

is printed in lieu of the long string. Execution then continues normally. On the IBM System/360, the space available for the formation of such strings is about 3800 characters. There is no limit to the length of a string that can be printed as a result of an output association, except the limit on the length of strings.

Appendix C. Examples

1. Syntax Recognizer for SNOBOL4

```
SNOBOL4 (VERSION 2.0, OCT. 7, 1968)
BELL TELEPHONE LABORATORIES, INCORPORATED


     *
     *      THIS PROGRAM IS A SYNTACTIC RECOGNIZER FOR SNOBOL4 STATEMENTS.
     *
     *      FIRST A SERIES OF PATTERNS IS BUILD CULMINATING IN A PATTERN
     *      WHICH MATCHES ONLY SYNTACTICALLY CORRECT STATEMENTS.  CARD IMAGES
     *      ARE THEN READ IN AND PROCESSED.  INCORRECT STATEMENTS ARE
     *      IDENTIFIED BY AN ERROR MESSAGE.
     *
     *      THE FUNCTION OPT FORMS A PATTERN THAT MATCHES EITHER NULL OR ITS
     *      ARGUMENT.
     *
              DEFINE('OPT(PATTERN)')                                          1
     *
              LETTERS    =    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'                      2
     *
     *      ON THE IBM SYSTEM/360 LETTERS INCLUDE LOWER CASE AS WELL.
     *
              DIGITS    =    '0123456789'                                     3
              ALPHANUMERICS   =   LETTERS DIGITS                              4
              BLANKS    =   SPAN(' ')                                         5
              INTEGER    =   SPAN(DIGITS)                                     6
              REAL    =   SPAN(DIGITS) '.' OPT(SPAN(DIGITS))                  7
              IDENTIFIER    =   ANY(LETTERS) OPT(SPAN(ALPHANUMERICS '_.'))    8
              UNARY    =   ANY('+-&.$*?¬%#')                                  9
              BINARY    =   ANY('-+.$*|/*#') | '**'                           10
              BINARYOP    =   BLANKS OPT(BINARY BLANKS)                       11
              UNQALPHABET    =   &ALPHABET                                    12
              UNQALPHABET   '"'    =                                          13
              UNQALPHABET   "'"    =                                          14
              DLITERAL    =    '"' SPAN(UNQALPHABET "'") '"'                  15
              SLITERAL    =    "'" SPAN(UNQALPHABET '"') "'"                  16
              LITERAL    =   SLITERAL | DLITERAL | INTEGER | REAL             17
              ELEMENT    =   OPT(UNARY) (IDENTIFIER | LITERAL | *FUNCTION_CALL 18
     .                       | '(' *EXPRESSION | OPT(BLANKS) ')' | *ARRAY_REF) 18
              OPERATION    =   *ELEMENT BINARYOP (*ELEMENT | *EXPRESSION)     19
              EXPRESSION    =   OPT(BLANKS) (*ELEMENT | *OPERATION | NULL)    20
     .                          OPT(BLANKS)                                   20
              ARG_LIST    =   *EXPRESSION OPT(',' *ARG_LIST)                  21
              FUNCTION_CALL    =   IDENTIFIER '(' *ARG_LIST ')'               22
              ARRAY_REF    =   IDENTIFIER '<' *ARG_LIST '>'                   23
              LABEL    =   ANY(ALPHANUMERICS) (BREAK(' ;') | REM)             24
              LABEL_FIELD    =   OPT(LABEL)                                   25
              GOTO    =   '(' EXPRESSION ')' | '<' EXPRESSION '>'             26
              GOTO_FIELD    =   OPT(BLANKS ':' FENCE OPT(BLANKS) (GOTO | 'S'  27
     .                          GOTO | 'F' GOTO | 'S' GOTO OPT(BLANKS) 'F'    27
     .                          GOTO | 'F' GOTO OPT(BLANKS) 'S' GOTO)         27
     .                          OPT(BLANKS))                                  27
              RULE    =   OPT(BLANKS ELEMENT (BLANKS '=' OPT(BLANKS EXPRESSION 28
     .                 ) | OPT(BLANKS EXPRESSION OPT(BLANKS '=' OPT(BLANKS    28
     .                 EXPRESSION)))))                                        28
              EOS    =   RPOS(0) | ';'                                        29
              STATEMENT    =   LABEL_FIELD RULE GOTO_FIELD EOS                30
     *
     *      THE PATTERN FOR RECOGNIZING STATEMENTS IS NOW FORMED.  THE
     *      PROGRAM TO ANALYZE INPUT CARDS FOLLOWS.
     *
              COMMENT    =   ANY('*-')                                        31
              CONTINUE    =   ANY('.+') . CC                                  32
              INPUT('INPUT',5,72)                                             33
              &ANCHOR    =   1                                                34
              &FULLSCAN    =   1                                              35
              EOF    =                                                        36
     *
     *      INITIALIZE PROCESS FROM FIRST CARD.
     *
     READI    IMAGE    =   TRIM(INPUT)                     :F(END)            37
              OUTPUT    =   '      ' IMAGE                                    38
     *
     *        DO NOT PROCESS COMMENT OR CONTINUE CARDS.
     *
              IMAGE   COMMENT                              :F(READC) S(READI) 39
     NEXTST   IDENT(EOF)                                   :F(END)            40
              OUTPUT    =   '      ' LINE                                     41
              IMAGE    =   LINE                                               42
     READC    LINE    =   TRIM(INPUT)                      :F(ENDGAME)        43
              LINE    COMMENT                              :S(PRINT)          44
              LINE    CONTINUE    =                        :F(ANALYZE)        45
              OUTPUT    =   '      ' CC LINE                                  46
              IMAGE    =   IMAGE LINE                      :(READC)           47
     ANALYZE  IMAGE    STATEMENT    =                      :F(ERROR)          48
              DIFFER(IMAGE)                                :S(ANALYZE)        49
              OUTPUT    =   '<<< NO SYNTACTIC ERROR >>>'                      50
     SKIP     OUTPUT    =                                  :(NEXTST)          51
     *
     *      IF AN ERRONEOUS STATEMENT IS  ENCOUNTERED IN A STRING OF
     *      STATEMENTS SEPARATED BY SEMICOLONS, SUBSEQUENT STATEMENTS ARE
     *      NOT PROCESSED.
     *
     ERROR    OUTPUT    =   '<<< SYNTACTIC ERROR >>>'          :(SKIP)            52
     *
     PRINT    OUTPUT    =   '      ' LINE                      :(READC)           53
     ENDGAME  EOF    =   1                                     :(ANALYZE)         54
     *
     *
     OPT      OPT    =   NULL | PATTERN                        :(RETURN)          55
     END                                                                          56

NO ERRORS DETECTED DURING COMPILATION
```

```
     *
     *      A VARIETY OF CORRECT AND INCORRECT SNOBOL4 STATEMENTS FOLLOW
     *
     -LIST
     COMPUTE  X   =  Y + 3 ** -'2'
<<< NO SYNTACTIC ERROR >>>

            X   =   Y+Z
<<< SYNTACTIC ERROR >>>

              ELEMENT<I,J>= ELEMENT<I,-J> + ELEMENT<-I,J>
<<< SYNTACTIC ERROR >>>

            A<X,Y,Z + 1>   =   F(X,STRUCTURE_BUILD(TYPE,LENGTH + 1))
<<< NO SYNTACTIC ERROR >>>

     SETUP    PAT1    =    (BREAK(',:') $ FIRST | SPAN(' .') $ SECOND
     .                     . VALUE ARBNO(BAL | LEN(1))   :($SWITCH)
<<< NO SYNTACTIC ERROR >>>

            DEFINE('F(X,Y))
<<< SYNTACTIC ERROR >>>

            L   =   LT(N,B<J> L + 1
<<< SYNTACTIC ERROR >>>

     NEWONE_TRIAL   X   =   ¬COORD<1,K> X * X
<<< NO SYNTACTIC ERROR >>>

            TRIM(INPUT)   PAT1    :S(OK)   :F(BAD)
<<< SYNTACTIC ERROR >>>

         X   =   3.01; Y = 2.    ; Z   =   X * -Y
<<< NO SYNTACTIC ERROR >>>


NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS   40


SNOBOL4 STATISTICS SUMMARY-
           1464 MS. COMPILATION TIME
           2047 MS. EXECUTION TIME
            171 STATEMENTS EXECUTED,      32 FAILED
              0 ARITHMETIC OPERATIONS PERFORMED
             39 PATTERN MATCHES PERFORMED
              0 REGENERATIONS OF DYNAMIC STORAGE
          11.97 MS. AVERAGE PER STATEMENT EXECUTED
```

2. Topological Sort

```
SNOBOL4 (VERSION 2.0, OCT. 7, 1968)
BELL TELEPHONE LABORATORIES, INCORPORATED


      *
      *         TOPOLOGICAL SORT
      *
      *    MAPS A PARTIAL ORDERING OF OBJECTS INTO A LINEAR ORDERING
      *
      *         A(1), A(2), ..., A(N)
      *
      *    SUCH THAT IF   A(S) < A(T) IN THE PARTIAL ORDERING,THEN S < T.
      *    (CF. D.E.KNUTH, THE ART OF COMPUTER PROGRAMMING,VOLUME 1,
      *    ADDISON-WESLEY,MASS.,1968, P.262)
      *
                &DUMP      = 1
                OUTPUT('OUT',6,'(121A1)')
                PAIR       = BREAK('<') . MU LEN(1) BREAK(',') . NU LEN(1)
                DATA('ITEM(COUNT,TOP)')
                DATA('NODE(SUC,NEXT)')
                DEFINE('DECR(X)')
                DEFINE('INDEX(TAU)')
      *
      *      READ IN THE NUMBER OF ITEMS, N, AND GENERATE AN ARRAY OF ITEMS.
      *
      *    EACH ITEM HAS TWO FIELDS, (COUNT,TOP), WHERE
      *         COUNT = NO. OF ELEMENTS PRECEEDING IT.
                TOP = TOP OF LIST OF ITEMS SUCCEEDING IT.
      *
                N          = TRIM(INPUT)
                X          = ARRAY('0:' N)
      *
      *    INITIALIZE THE ITEMS TO (0,NULL).
      *
      T1        X<I>       =  ITEM(0,)               :F(T1A)
                I          =  I + 1                  :(T1)
      *
      *    READ IN RELATIONS.
      *
      T1A       OUT        = '1 THE RELATIONS ARE:'
      T2A       REL        = TRIM(INPUT) ','         :F(T3A)
                OUTPUT     = REL
      T2        REL        PAIR  =                   :F(T2A)
                J          = INDEX(MU)
                K          = INDEX(NU)
      *
      *    SINCE MU < NU, INCREASE THE COUNT OF THE KTH ITEM AND ADD A
      *    NODE TO THE LIST OF SUCCESSORS OF THE JTH ITEM.
      *
      T3        COUNT(X<K>) = COUNT(X<K>) + 1
                TOP(X<J>)  = NODE(K,TOP(X<J>))       :(T2)
      *
      *     A QUEUE IS MAINTAINED OF THOSE ITEMS WITH ZERO COUNT FIELD.
      *     THE LINKS FOR THE QUEUE, QLINK, ARE KEPT IN THE COUNT FIELD.
      *     THE VARIABLES F,R POINT TO THE FRONT AND REAR OF THE QUEUE.
      *
      T3A       OPSYN('QLINK','COUNT')
      *
      *     INITIALIZE THE QUEUE FOR OUTPUT.
      *
```

```
          R            = 0
          QLINK(X<0>)  = 0
          K            = 0
T4        K            = ?X<K + 1>  K + 1        :F(T4A)
          QLINK(X<R>)  = EQ(COUNT(X<K>),0)   K :F(T4)
          R            = K                     :(T4)
T4A       F            = QLINK(X<0>)
*
*     OUTPUT THE FRONT OF THE QUEUE.
*
          OUT          = '0 THE LINEAR ORDERING IS:'
T5        OUTPUT       = NE(F,0)   $(F ':')    :F(T8)
          N            = N - 1
          P            = TOP(X<F>)
*
*     ERASE RELATIONS.
*
T6        IDENT(P)                             :S(T7)
          DECR(.COUNT(X<SUC(P)>))              :S(T6A)
*
*     IF COUNT IS ZERO ADD  ITEM TO QUEUE.
*
          QLINK(X<R>)  = SUC(P)
          R            = SUC(P)
T6A       P            = NEXT(P)               :(T6)
*
*     REMOVE FROM QUEUE.
*
T7        F            = QLINK(X<F>)           :(T5)
*
*     FUNCTION DEFINITIONS.
*
DECR      $X           = GT($X,1)   $X - 1     :S(RETURN)
          $X           = 0                     :(FRETURN)
*
INDEX     INDEX        = DIFFER($(TAU ':'))   $(TAU ':')   :S(RETURN)
          TERMCT       = LT(TERMCT,N)   TERMCT + 1    :F(FRETURN)
          INDEX        = TERMCT
          $(TERMCT ':')    =  TAU
          $(TAU ':')    =  TERMCT              :(RETURN)
*
T8        OUTPUT       = NE(N,0) 'THE ORDERING CONTAINS A LOOP.'
END
```

```
NO ERRORS DETECTED DURING COMPILATION
```

```
 THE RELATIONS ARE:
LETTERS<ALPHANUM,NUMBERS<ALPHANUM,
BLANKS<OPTBLANKS,
NUMBERS<REAL,
NUMBERS<INTEGER,
LETTERS<VARIABLE,ALPHANUM<VARIABLE,
BINARY<BINARYOP,BLANKS<BINARYOP,
UNQALPHABET<DLITERAL,
UNQALPHABET<SLITERAL,
SLITERAL<LITERAL,DLITERAL<LITERAL,INTEGER<LITERAL,REAL<LITERAL,

 THE LINEAR ORDERING IS:
LETTERS
NUMBERS
BLANKS
BINARY
UNQALPHABET
INTEGER
REAL
ALPHANUM
OPTBLANKS
BINARYOP
SLITERAL
DLITERAL
VARIABLE
LITERAL


NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS   46


DUMP OF VARIABLES AT TERMINATION

NATURAL VARIABLES

 ABORT = PATTERN
 ALPHANUM: = 2
 ARB = PATTERN
 BAL = PATTERN
 BINARY: = 9
 BINARYOP: = 10
 BLANKS: = 4
 DLITERAL: = 12
 F = 0
 FAIL = PATTERN
 FENCE = PATTERN
 I = 15
 INPUT = 'SLITERAL<LITERAL,DLITERAL<LITERAL,INTEGER<LITERAL,REAL<LITERAL
 INTEGER: = 7
 J = 6
 K = 14
 LETTERS: = 1
 LITERAL: = 14
 MU = 'REAL'
 N = 0
 NU = 'LITERAL'
 NUMBERS: = 3
 OPTBLANKS: = 5
 OUT = '0 THE LINEAR ORDERING IS:'
 OUTPUT = 'LITERAL'
 PAIR = PATTERN
 R = 14
 REAL: = 6
 REM = PATTERN
 SLITERAL: = 13
 SUCCEED = PATTERN
 TERMCT = 14
 UNQALPHABET: = 11
 VARIABLE: = 8
 X = ARRAY('0:14')
 1: = 'LETTERS'
 10: = 'BINARYOP'
 11: = 'UNQALPHABET'
 12: = 'DLITERAL'
 13: = 'SLITERAL'
 14: = 'LITERAL'
 2: = 'ALPHANUM'
 3: = 'NUMBERS'
 4: = 'BLANKS'
 5: = 'OPTBLANKS'
 6: = 'REAL'
 7: = 'INTEGER'
 8: = 'VARIABLE'
 9: = 'BINARY'


UNPROTECTED KEYWORDS

 &ABEND = 0
 &ANCHOR = 0
 &DUMP = 1
 &FTRACE = 0
 &FULLSCAN = 0
 &MAXLNGTH = 5000
 &STLIMIT = 50000
 &TRACE = 0


SNOBOL4 STATISTICS SUMMARY-


          1431 MS. COMPILATION TIME
           632 MS. EXECUTION TIME
           430 STATEMENTS EXECUTED,      70 FAILED
            93 ARITHMETIC OPERATIONS PERFORMED
            24 PATTERN MATCHES PERFORMED
             0 REGENERATIONS OF DYNAMIC STORAGE
          1.47 MS. AVERAGE PER STATEMENT EXECUTED
```

3. ICEBOL - A Compressor of SNOBOL4 Programs

```
ICEBOL.VER.2                                                            ICEB   1
***********************************************************************ICEB   2
*                                                                     *ICEB   3
*                                                                     *ICEB   4
*                              ICEBOL                                 *ICEB   5
*                                                                     *ICEB   6
*     IS A PROGRAM TO COMPRESS SNOBOL4 SOURCE PROGRAMS.  IT DOES THIS *ICEB   7
*     BY REPLACING A SEQUENCE OF BLANKS BY A SINGLE BLANK AND IF NEC- *ICEB   8
*     ESSARY INDICATES AN END-OF-STATEMENT WITH A SEMI-COLON.         *ICEB   9
*     A TYPICAL COMPRESSION FACTOR IS THREE TO ONE.                   *ICEB  10
*                                                                     *ICEB  11
*                            USAGE                                    *ICEB  12
*     THE INPUT DATA TO ICEBOL CAN BE ANY SNOBOL4 PROGRAM OR SECTION OF*ICEB  13
*     PROGRAM PRECEDED BY ZERO OR MORE CONTROL CARDS.  CONTROL CARDS  *ICEB  14
*     START WITH A VERTICAL LINE IN COLUMN 1, AND MAY BE ANY OF THE   *ICEB  15
*     FOLLOWING (WHERE BLANKS ARE IRRELEVENT)                         *ICEB  16
*     |   DON'T CRUNCH COMMENTS                                       *ICEB  17
*     |   NO  COMMENTS                                                *ICEB  18
*     |   NO LIST                                                     *ICEB  19
*     COMMENTS ARE NORMALLY INCLUDED AS PART OF THE COMPRESSED DECK BUT*ICEB  20
*     KEEPING WITHIN THE SPIRIT OF ICEBOL SUCCESSIVE BLANKS ARE       *ICEB  21
*     NORMALLY REPLACED BY A SINGLE BLANK AND THEREBY MULTI-LINE      *ICEB  22
*     COMMENTS CAN BE COMPRESSED.  THE FIRST CONTROL CARD             *ICEB  23
*     ABOVE SUPPRESSES THE COMPRESSION OF COMMENTS.  THE SECOND CONT- *ICEB  24
*     ROL CARD ABOVE REMOVES COMMENTS ALTOGETHER.                     *ICEB  25
*     SNOBOL4 CONTROL CARDS (CARDS BEGINNING WITH A MINUS , -)  NORMAL-*ICEB  26
*     LY APPEAR BY THEMSELVES ON A SINGLE LINE.  THESE WILL BE REMOVED *ICEB  27
*     IF THE THIRD CONTROL CARD ABOVE IS INCLUDED.                    *ICEB  28
*                                                                     *ICEB  29
*                      LABELING AND CARD NUMBERING                    *ICEB  30
*                                                                     *ICEB  31
*     THE FIRST FOUR CHARACTERS OF THE FIRST CARD ARE USED TO LABEL THE*ICEB  32
*     DECK.  THE DECK IS SEQUENCE NUMBERED.                           *ICEB  33
*                                                                     *ICEB  34
*                                                                     *ICEB  35
*                                                     J. F. GIMPEL    *ICEB  36
*                                                     7/15/68         *ICEB  37
*                                                                     *ICEB  38
*                                                                     *ICEB  39
***********************************************************************ICEB  40
          &DUMP = 1                                                     ICEB  41
          INPUT('INPUT',5,72)                                           ICEB  42
          DEFINE( 'BLANK(N)')                     :(BLANK.END)          ICEB  43
BLANK     BLANK = DUP(' ',N)                      :(RETURN)             ICEB  44
BLANK.END DEFINE('DUP(S,N)')                      :(DUP.END)            ICEB  45
DUP       DUP = GT(N,0)  DUP S                    :F(RETURN)            ICEB  46
          N = N - 1                               :(DUP)                ICEB  47
DUP.END   DEFINE( 'RADJ(S,N)')                    :(RADJ.END)           ICEB  48
RADJ      (MANY.BLANKS S) RTAB(N) REM . RADJ      :S( RETURN)           ICEB  49
          MANY.BLANKS = MANY.BLANKS '          '  :(RADJ)               ICEB  50
RADJ.END  DEFINE( 'LADJ(S,N)')                    :(LADJ.END)           ICEB  51
LADJ      (S MANY.BLANKS) TAB(N)  . LADJ          :S(RETURN)            ICEB  52
          MANY.BLANKS = MANY.BLANKS '        '    :(LADJ)               ICEB  53
LADJ.END  DEFINE('TOSS(A)')                       :(TOSS.END)           ICEB  54
TOSS      IDENT(A)                                :S(RETURN)            ICEB  55
          CARD.NO = CARD.NO + 1                                         ICEB  56
          A = LADJ(A ,72) LABEL RADJ(CARD.NO,4)                         ICEB  57
          PUNCH = A                                                     ICEB  58
          OUTPUT =                                                      ICEB  59
          OUTPUT = A                                                    ICEB  60
          OUTPUT =                                     : (RETURN)             ICEB  61
TOSS.END  DEFINE('SPACE(N)')                           : (SPACE.END)          ICEB  62
SPACE     N GT(N,0) . OUTPUT REM = N - 1               :S(SPACE) F(RETURN)    ICEB  63
SPACE.END SPECIAL = FENCE ( '*' | '-')                                        ICEB  64
          COMMENT = FENCE '*'                                                 ICEB  65
          COMM.FLAG = 'ON'                                                    ICEB  66
          LIST.FLAG = 'ON'                                                    ICEB  67
          CRUNCH.FLAG = 'ON'                                                  ICEB  68
          IGNORE.CARD = FAIL                                                  ICEB  69
          INDENT = BLANK(30)                                                  ICEB  70
          OUTPUT( 'CONTROL' , 6 , '(132A1)')                                  ICEB  71
          CONTROL = 1                                                         ICEB  72
          SPACE(20)                                                           ICEB  73
          OUTPUT = INDENT INDENT 'ICEBOL2'                                    ICEB  74
          CONTROL = '+' INDENT INDENT '_______'                               ICEB  75
          CONTROL = 1                                                         ICEB  76
          DEFINE( 'CAT.IN(X)' , 'CAT.IN.1' )           : (CAT.IN.END)         ICEB  77
CAT.IN.1  LABEL = '    '                                                      ICEB  78
          CAT.NEXT = CAT.GET()                         :F(CAT.IN.1)           ICEB  79
          CAT.NEXT FENCE '|'                           :S(CI.7)               ICEB  80
          CAT.NEXT FENCE ' '                           :S(CI.1)               ICEB  81
          CAT.NEXT LEN(4) . LABEL                      : (CI.1)               ICEB  82
CI.7      CAT.NEXT "DON'T" ARBNO(' ') "CRUNCH" ARBNO(' ') "COMM" =            ICEB  83
.                                                      :F(CI.6)               ICEB  84
          CRUNCH.FLAG = 'OFF'                                                 ICEB  85
CI.6      CAT.NEXT 'NO' ARBNO(' ') 'COMM' =            :F(CI.8)               ICEB  86
          COMM.FLAG = 'OFF'                                                   ICEB  87
CI.8      CAT.NEXT 'COMM' =                            :F(CI.9)               ICEB  88
          COMM.FLAG = 'ON'                                                    ICEB  89
CI.9      CAT.NEXT 'NO' ARBNO(' ') 'LIST' =            :F(CI.10)              ICEB  90
          LIST.FLAG = 'OFF'                                                   ICEB  91
CI.10     CAT.NEXT 'CONT' =                            :F(CI.11)              ICEB  92
          LIST.FLAG = 'ON'                                                    ICEB  93
CI.11                                                  : (CAT.IN.1)           ICEB  94
CI.1      DEFINE( 'CAT.IN(X)' , 'CAT.IN.2')                                   ICEB  95
          IGNORE.CARD = (IGNORE.CARD | '*' ) IDENT(COMM.FLAG,'OFF')           ICEB  96
          IGNORE.CARD = (IGNORE.CARD | '-') IDENT( LIST.FLAG,'OFF')           ICEB  97
          IGNORE.FLAG = (IGNORE.FLAG | '*' RPOS(0)) IDENT( CRUNCH.FLAG        ICEB  98
.         , 'ON') IDENT(COMM.FLAG , 'ON')                                     ICEB  99
          IGNORE.CARD = FENCE (RPOS( 0) | IGNORE.CARD)                        ICEB 100
CAT.IN.2  CAT.IN = CAT.NEXT                                                   ICEB 101
CI.3      CAT.NEXT = CAT.GET()                         :F(CI.2)               ICEB 102
          CAT.NEXT SPECIAL                             :S(CI.SPECIAL)         ICEB 103
          CAT.IN SPECIAL                               :S(RETURN)             ICEB 104
          CAT.NEXT FENCE '.' = ' '                     :F(CI.5)               ICEB 105
          CAT.IN = CAT.IN CAT.NEXT                     : (CI.3)               ICEB 106
CI.5      CAT.IN ' '                                   :S(CI.61)              ICEB 107
          CAT.NEXT FENCE ' '                           :F(CI.61)              ICEB 108
          CAT.IN = CAT.IN CAT.NEXT                     : (CI.3)               ICEB 109
CI.61     CAT.IN ':'                                   :S(RETURN)             ICEB 110
          CAT.NEXT FENCE ' ' SPAN( ' ') ':' = ' :'                            ICEB 111
.                                                      :F(RETURN)             ICEB 112
          CAT.IN = CAT.IN CAT.NEXT                     : (CI.3)               ICEB 113
CI.SPECIAL                                                                    ICEB 114
          IDENT(CRUNCH.FLAG,'OFF')                     :S(RETURN)             ICEB 115
          CAT.IN COMMENT                               :F(RETURN)             ICEB 116
          CAT.NEXT COMMENT = ' '                       :F(RETURN)             ICEB 117
CI.COMMENT                                                                    ICEB 118
          CAT.NEXT ' ' SPAN(' ') = ' '                 :S(CI.COMMENT)         ICEB 119
          CAT.IN = CAT.IN CAT.NEXT                     : (CI.3)               ICEB 120
CI.2      DEFINE( 'CAT.IN(X)' , 'CAT.IN.3')            : (RETURN)             ICEB 121
CAT.IN.3                                               : (FRETURN)            ICEB 122
CAT.IN.END                                                                    ICEB 123
```

```
          DEFINE('CAT.GET(X)')                    :(CAT.GET.END)
CAT.GET   INPUT LEN(72) . CAT.GET                 :F( FRETURN)
          CAT.GET.CNT = CAT.GET.CNT + 1
          CAT.GET = TRIM(CAT.GET)
          OUTPUT = 'INPUT CARD' RADJ(CAT.GET.CNT , 5) ':' CAT.GET
          CAT.GET IGNORE.CARD                     :S(CAT.GET)F(RETURN)
CAT.GET.END
          DEFINE('SPEW(LINE)' )                   :( SPEW.END)
SPEW      BUFF = DIFFER(BUFF,NULL) BUFF ';'
          GT(SIZE(BUFF) + SIZE( LINE), 72)        :S(SPEW.2)
          BUFF = BUFF LINE
          LT(SIZE(BUFF),70)                       :S( RETURN)
          TOSS(BUFF)
          BUFF =                                  :(RETURN)
SPEW.2    NBUFF = BUFF
          BUFF = LINE
SPEW.3    A =
          BUFF CHUNK =
          IDENT(A,NULL)                           :S(SPEW.ERROR)
          LE(SIZE( NBUFF) + SIZE(A ),72)          :S(SPEW.6)
          BUFF = A BUFF
          NBUFF ';' RPOS(0) =                     :S( PERIOD.OUT)
          NBUFF '; ' RPOS(0) =                    :F(SPEW.5)
          BUFF = ' ' BUFF                         :( PERIOD.OUT)
SPEW.5    BUFF FENCE ARBNO(' ') ';' =             :S(PERIOD.OUT)
          BUFF = '.' BUFF
          BUFF FENCE '. ' = '.'
PERIOD.OUT
          TOSS(NBUFF)
          GT(SIZE(BUFF) ,72)                      :F(RETURN)
          NBUFF =                                 :(SPEW.3)
SPEW.6    NBUFF = NBUFF A                         :(SPEW.3)
SPEW.END  QT = '"' "'"
          DQ = '"'
          SQ = "'"
          QUOTED.LITERAL = SQ BREAK(SQ ) SQ | DQ BREAK(DQ) DQ
          OTHER = LEN(1) BREAK(QT ' ;(),' ) ( QUOTED.LITERAL | '(' |
.         NULL ) | RTAB(0)
          CHUNK = FENCE (QUOTED.LITERAL | ANY('( ;') | OTHER) . A
          PAT = ARB . B (' ' | ANY(QT)) . C
          NB72S.PATTERN = (TAB(60) ARB) . N ' ' ARBNO(NOTANY(' ')) . B
.         RPOS(0)
SQ.0      S = CAT.IN()                            :F(SQ.99)
          S SPECIAL                               :F(SQ.START)
          TOSS(BUFF)
          BUFF =
          IDENT(CRUNCH.FLAG , 'ON')               :S(SQ.4)
SQ.5      TOSS(S)                                 :(SQ.0)
SQ.4      S COMMENT                               :F(SQ.5)
SQ.7      GT(SIZE(S) , 72)                        :F( SQ.5)
          S LEN(72) . S REM . SS
          S NB72S.PATTERN =                       :F(SQ.8)
          TOSS(N)
          S = '* ' B SS                           :(SQ.7)
SQ.8      S = S SS
          S LEN(72) . N = '* '
          TOSS(N)                                 :(SQ.7)
SQ.START  S (BREAK(' ;') | REM) . N =
SQ.1      S PAT =                                 :F(SQ.2)
          IDENT('  ',C )                          :F( SQ.3)
          N = N B ' '
          S FENCE SPAN(' ') =                     :(SQ.1)
SQ.3      S BREAK(C) . D C =                      :F(SQ.ERR)
```

```
         N = N B C D C                            :(SQ.1)              ICEB 187
SQ.2     N = N S                                                       ICEB 188
         SPEW(N)                                  :(SQ.0)              ICEB 189
SQ.99    TOSS(BUFF)                                                    ICEB 190
         OUTPUT = 'END OF FILE REACHED BY ICEBOL'                      ICEB 191
         ENDFILE(7)                                                    ICEB 192
END                                                                    ICEB 193
```

The result of applying ICEBOL to itself follows.

```
ICEBOL.VER.2                                                           ICEB   1
***********************************************************************ICEB   2
*  * * ICEBOL * * IS A PROGRAM TO COMPRESS SNOBOL4 SOURCE PROGRAMS. IT ICEB   3
* DOES THIS * BY REPLACING A SEQUENCE OF BLANKS BY A SINGLE BLANK AND  ICEB   4
* IF NEC- * ESSARY INDICATES AN END-OF-STATEMENT WITH A SEMI-COLON. * A ICEB  5
* TYPICAL COMPRESSION FACTOR IS THREE TO ONE. * * USAGE * THE INPUT    ICEB   6
* DATA TO ICEBOL CAN BE ANY SNOBOL4 PROGRAM OR SECTION OF* PROGRAM     ICEB   7
* PRECEDED BY ZERO OR MORE CONTROL CARDS. CONTROL CARDS * START WITH A ICEB   8
* VERTICAL LINE IN COLUMN 1, AND MAY BE ANY OF THE * FOLLOWING (WHERE  ICEB   9
* BLANKS ARE IRRELEVENT) * | DON'T CRUNCH COMMENTS * | NO COMMENTS * | ICEB  10
* NO LIST * COMMENTS ARE NORMALLY INCLUDED AS PART OF THE COMPRESSED   ICEB  11
* DECK BUT* KEEPING WITHIN THE SPIRIT OF ICEBOL SUCCESSIVE BLANKS ARE * ICEB 12
* NORMALLY REPLACED BY A SINGLE BLANK AND THEREBY MULTI-LINE * COMMENTS ICEB 13
* CAN BE COMPRESSED. THE FIRST CONTROL CARD * ABOVE SUPPRESSES THE     ICEB  14
* COMPRESSION OF COMMENTS. THE SECOND CONT- * ROL CARD ABOVE REMOVES   ICEB  15
* COMMENTS ALTOGETHER. * SNOBOL4 CONTROL CARDS (CARDS BEGINNING WITH A ICEB  16
* MINUS , -) NORMAL-* LY APPEAR BY THEMSELVES ON A SINGLE LINE. THESE  ICEB  17
* WILL BE REMOVED * IF THE THIRD CONTROL CARD ABOVE IS INCLUDED. * *   ICEB  18
* LABELING AND CARD NUMBERING * * THE FIRST FOUR CHARACTERS OF THE     ICEB  19
* FIRST CARD ARE USED TO LABEL THE* DECK. THE DECK IS SEQUENCE         ICEB  20
* NUMBERED. * * * J. F. GIMPEL * 7/15/68 * * * ************************ICEB  21
* ********************************************                         ICEB  22
 &DUMP = 1; INPUT('INPUT',5,72); DEFINE( 'BLANK(N)') :(BLANK.END);BLANK ICEB  23
.BLANK = DUP(' ',N)  :(RETURN);BLANK.END DEFINE('DUP(S,N)') :(DUP.END) ICEB  24
DUP DUP = GT(N,0) DUP S :F(RETURN); N = N - 1 :(DUP);DUP.END DEFINE(   ICEB  25
.'RADJ(S,N)') :(RADJ.END);RADJ (MANY.BLANKS S) RTAB(N) REM . RADJ :S(  ICEB  26
.RETURN); MANY.BLANKS = MANY.BLANKS '          ' :(RADJ);RADJ.END DEFINE(ICEB 27
.'LADJ(S,N)') :(LADJ.END);LADJ (S MANY.BLANKS) TAB(N) . LADJ :S(RETURN) ICEB 28
 MANY.BLANKS = MANY.BLANKS '        ' :(LADJ);LADJ.END DEFINE('TOSS(A)') ICEB 29
.:(TOSS.END);TOSS IDENT(A)  :S(RETURN); CARD.NO = CARD.NO + 1; A = LADJ(AICEB 30
.,72) LABEL RADJ(CARD.NO,4); PUNCH = A; OUTPUT =; OUTPUT = A; OUTPUT =  ICEB  31
.:(RETURN);TOSS.END DEFINE('SPACE(N)') :(SPACE.END);SPACE N GT(N,0) .  ICEB  32
.OUTPUT REM = N - 1 :S(SPACE) F(RETURN);SPACE.END SPECIAL = FENCE ( '*' ICEB 33
.| '-'); COMMENT = FENCE '*'; COMM.FLAG = 'ON'; LIST.FLAG = 'ON'       ICEB  34
 CRUNCH.FLAG = 'ON'; IGNORE.CARD = FAIL; INDENT = BLANK(30); OUTPUT(   ICEB  35
.'CONTROL' , 6 , '(132A1)'); CONTROL = 1; SPACE(20); OUTPUT = INDENT   ICEB  36
.INDENT 'ICEBOL2'; CONTROL = '+' INDENT INDENT '_______'; CONTROL = 1  ICEB  37
 DEFINE( 'CAT.IN(X)' , 'CAT.IN.1' ) :(CAT.IN.END);CAT.IN.1 LABEL =     ICEB  38
.'    '; CAT.NEXT = CAT.GET() :F(CAT.IN.1); CAT.NEXT FENCE '|' :S(CI.7) ICEB 39
 CAT.NEXT FENCE ' ' :S(CI.1); CAT.NEXT LEN(4) . LABEL :(CI.1);CI.7     ICEB  40
.CAT.NEXT "DON'T" ARBNO(' ') "CRUNCH" ARBNO(' ') "COMM" = :F(CI.6)     ICEB  41
 CRUNCH.FLAG = 'OFF';CI.6 CAT.NEXT 'NO' ARBNO(' ') 'COMM' = :F(CI.8)   ICEB  42
 COMM.FLAG = 'OFF';CI.8 CAT.NEXT 'COMM' = :F(CI.9); COMM.FLAG = 'ON'   ICEB  43
CI.9 CAT.NEXT 'NO' ARBNO(' ') 'LIST' = :F(CI.10); LIST.FLAG = 'OFF'    ICEB  44
CI.10 CAT.NEXT 'CONT' = :F(CI.11); LIST.FLAG = 'ON';CI.11 :(CAT.IN.1)  ICEB  45
CI.1 DEFINE( 'CAT.IN(X)' , 'CAT.IN.2'); IGNORE.CARD = (IGNORE.CARD | '*'ICEB 46
.) IDENT(COMM.FLAG,'OFF'); IGNORE.CARD = (IGNORE.CARD | '-') IDENT(    ICEB  47
.LIST.FLAG,'OFF'); IGNORE.FLAG = (IGNORE.FLAG | '*' RPOS(0)) IDENT(    ICEB  48
.CRUNCH.FLAG , 'ON') IDENT(COMM.FLAG , 'ON'); IGNORE.CARD = FENCE (RPOS(ICEB 49
.0) | IGNORE.CARD);CAT.IN.2 CAT.IN = CAT.NEXT;CI.3 CAT.NEXT = CAT.GET() ICEB 50
.:F(CI.2); CAT.NEXT SPECIAL :S(CI.SPECIAL); CAT.IN SPECIAL :S(RETURN)  ICEB  51
 CAT.NEXT FENCE '.' = ' ' :F(CI.5); CAT.IN = CAT.IN CAT.NEXT :(CI.3)   ICEB  52
CI.5 CAT.IN ' ' :S(CI.61); CAT.NEXT FENCE ' ' :F(CI.61); CAT.IN = CAT.INICEB 53
.CAT.NEXT :(CI.3);CI.61 CAT.IN ':' :S(RETURN); CAT.NEXT FENCE ' ' SPAN( ICEB 54
.' ') ':' = ' :' :F(RETURN); CAT.IN = CAT.IN CAT.NEXT :(CI.3);CI.SPECIALICEB 55
.IDENT(CRUNCH.FLAG,'OFF') :S(RETURN); CAT.IN COMMENT :F(RETURN)        ICEB  56
 CAT.NEXT COMMENT = ' ' :F(RETURN);CI.COMMENT CAT.NEXT ' ' SPAN(' ') =  ICEB  57
.' ' :S(CI.COMMENT); CAT.IN = CAT.IN CAT.NEXT :(CI.3);CI.2 DEFINE(     ICEB  58
.'CAT.IN(X)' , 'CAT.IN.3') :(RETURN);CAT.IN.3 :(FRETURN);CAT.IN.END    ICEB  59
.DEFINE('CAT.GET(X)') :(CAT.GET.END);CAT.GET INPUT LEN(72) . CAT.GET :F(ICEB 60
.FRETURN); CAT.GET.CNT = CAT.GET.CNT + 1; CAT.GET = TRIM(CAT.GET)       ICEB  61
 OUTPUT = 'INPUT CARD' RADJ(CAT.GET.CNT , 5) ':' CAT.GET; CAT.GET       ICEB  62
.IGNORE.CARD :S(CAT.GET)F(RETURN);CAT.GET.END DEFINE('SPEW(LINE)' ) :(  ICEB  63
.SPEW.END);SPEW BUFF = DIFFER(BUFF,NULL) BUFF ';'; GT(SIZE(BUFF) + SIZE(ICEB  64
.LINE), 72) :S(SPEW.2); BUFF = BUFF LINE; LT(SIZE(BUFF),70) :S( RETURN) ICEB  65
 TOSS(BUFF); BUFF = :(RETURN);SPEW.2 NBUFF = BUFF; BUFF = LINE;SPEW.3 A ICEB  66
.=; BUFF CHUNK =; IDENT(A,NULL) :S(SPEW.ERROR); LE(SIZE( NBUFF) + SIZE(AICEB  67
.),72) :S(SPEW.6); BUFF = A BUFF; NBUFF ';' RPOS(0) = :S( PERIOD.OUT)   ICEB  68
 NBUFF '; ' RPOS(0) = :F(SPEW.5); BUFF = ' ' BUFF :( PERIOD.OUT);SPEW.5 ICEB  69
.BUFF FENCE ARBNO(' ') ';' = :S(PERIOD.OUT); BUFF = '.' BUFF; BUFF FENCEICEB  70
.'. ' = '.';PERIOD.OUT TOSS(NBUFF); GT(SIZE(BUFF) ,72) :F(RETURN); NBUFFICEB  71
.= :(SPEW.3);SPEW.6 NBUFF = NBUFF A :(SPEW.3);SPEW.END QT = '"' "'"; DQ ICEB  72
.= '"'; SQ = "'"; QUOTED.LITERAL = SQ BREAK(SQ ) SQ | DQ BREAK(DQ) DQ   ICEB  73
 OTHER = LEN(1) BREAK(QT ' ;(),' ) ( QUOTED.LITERAL | '(' | NULL ) |    ICEB  74
.RTAB(0); CHUNK = FENCE (QUOTED.LITERAL | ANY('( ;') | OTHER) . A; PAT =ICEB  75
.ARB . B (' ' | ANY(QT)) . C; NB72S.PATTERN = (TAB(60) ARB) . N ' '     ICEB  76
.ARBNO(NOTANY(' ')) . B RPOS(0);SQ.0 S = CAT.IN() :F(SQ.99); S SPECIAL  ICEB  77
.:F(SQ.START); TOSS(BUFF); BUFF =; IDENT(CRUNCH.FLAG , 'ON') :S(SQ.4)   ICEB  78
SQ.5 TOSS(S) :(SQ.0);SQ.4 S COMMENT :F(SQ.5);SQ.7 GT(SIZE(S) , 72) :F(  ICEB  79
.SQ.5); S LEN(72) . S REM . SS; S NB72S.PATTERN = :F(SQ.8); TOSS(N); S =ICEB  80
.'* ' B SS :(SQ.7);SQ.8 S = S SS; S LEN(72) . N = '* '; TOSS(N) :(SQ.7) ICEB  81
SQ.START S (BREAK(' ;') | REM) . N =;SQ.1 S PAT = :F(SQ.2); IDENT(' ',CICEB  82
.) :F( SQ.3); N = N B ' '; S FENCE SPAN(' ') = :(SQ.1);SQ.3 S BREAK(C) .ICEB  83
.D C = :F(SQ.ERR); N = N B C D C :(SQ.1);SQ.2 N = N S; SPEW(N) :(SQ.0)  ICEB  84
SQ.99 TOSS(BUFF); OUTPUT = 'END OF FILE REACHED BY ICEBOL'; ENDFILE(7)  ICEB  85
END                                                                     ICEB  86
```

4. Factorial Table Generator

```
SNOBOL4 (VERSION 2.0, OCT. 7, 1968)
BELL TELEPHONE LABORATORIES, INCORPORATED


    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *                                                                     *
    *       THIS PROGRAM COMPUTES AND PRINTS A TABLE OF N FACTORIAL       *
    *       FOR VALUES OF N FROM 1 THROUGH AN UPPER LIMIT "NX".           *
    *                                                                     *
    *       IT DEMONSTRATES A METHOD OF MANIPULATING NUMBERS WHICH ARE    *
    *       TOO LARGE FOR THE COMPUTER, AS STRINGS OF CHARACTERS.  THE    *
    *       COMMAS IN THE PRINTED VALUES ARE OPTIONAL, ADDED FOR READING *
    *       EASE.                                                         *
    *                                                                     *
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *
    *       INITIALIZATION.
    *
            NX = 45
    *
            N = 1
            NSET = 1
            NUM = ARRAY(1000)
            NUM<1> = 1
            FILL = ARRAY('0:3')
            FILL<0> = '000'
            FILL<1> = '00'
            FILL<2> = '0'
    *
            OUTPUT = '         TABLE OF FACTORIALS FOR 1 THROUGH ' NX
            OUTPUT =
    *
    *       COMPUTE THE NEXT VALUE FROM THE PREVIOUS ONE.
    *
    L1      I = 1
    L2      NUM<I> = NUM<I> * N                              :F(ERR)
            I = LT(I,NSET) I + 1                             :S(L2)
            I = 1
    L3      LT(NUM<I>,1000)                                  :S(L4)
            NUMX = NUM<I> / 1000                             :F(ERR)
            NUM<I + 1> = NUM<I + 1> + NUMX                   :F(ERR)
            NUM<I> = NUM<I> - 1000 * NUMX                    :F(ERR)
    L4      I = LT(I,NSET) I + 1                             :S(L3)
    *
    *       FORM A STRING REPRESENTING THE FACTORIAL.
    *
    L5      NSET = DIFFER(NUM<NSET + 1>) NSET + 1
            NUMBER = NUM<NSET>                               :F(ERR)
            I = GT(NSET,1) NSET - 1                          :F(L7)
    L6      NUMBER = NUMBER ',' FILL<SIZE(NUM<I>)> NUM<I>
            I = GT(I,1) I - 1                                :S(L6)
    *
    *       OUTPUT A LINE OF THE TABLE.
    *
    L7      OUTPUT = N '!=' NUMBER
            N = LT(N,NX) N + 1                               :S(L1)F(END)
    *
    *       ERROR TERMINATION.
    *
    ERR     OUTPUT = N '! CANNOT BE COMPUTED BECAUSE OF TABLE OVERFLOW.'
```

```
                OUTPUT = '      INCREASE THE SIZE OF ARRAY "NUM".'                29
     *
     END                                                                          30

NO ERRORS DETECTED DURING COMPILATION
```

```
          TABLE OF FACTORIALS FOR 1 THROUGH 45

1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5,040
8!=40,320
9!=362,880
10!=3,628,800
11!=39,916,800
12!=479,001,600
13!=6,227,020,800
14!=87,178,291,200
15!=1,307,674,368,000
16!=20,922,789,888,000
17!=355,687,428,096,000
18!=6,402,373,705,728,000
19!=121,645,100,408,832,000
20!=2,432,902,008,176,640,000
21!=51,090,942,171,709,440,000
22!=1,124,000,727,777,607,680,000
23!=25,852,016,738,884,976,640,000
24!=620,448,401,733,239,439,360,000
25!=15,511,210,043,330,985,984,000,000
26!=403,291,461,126,605,635,584,000,000
27!=10,888,869,450,418,352,160,768,000,000
28!=304,888,344,611,713,860,501,504,000,000
29!=8,841,761,993,739,701,954,543,616,000,000
30!=265,252,859,812,191,058,636,308,480,000,000
31!=8,222,838,654,177,922,817,725,562,880,000,000
32!=263,130,836,933,693,530,167,218,012,160,000,000
33!=8,683,317,618,811,886,495,518,194,401,280,000,000
34!=295,232,799,039,604,140,847,618,609,643,520,000,000
35!=10,333,147,966,386,144,929,666,651,337,523,200,000,000
36!=371,993,326,789,901,217,467,999,448,150,835,200,000,000
37!=13,763,753,091,226,345,046,315,979,581,580,902,400,000,000
38!=523,022,617,466,601,111,760,007,224,100,074,291,200,000,000
39!=20,397,882,081,197,443,358,640,281,739,902,897,356,800,000,000
40!=815,915,283,247,897,734,345,611,269,596,115,894,272,000,000,000
41!=33,452,526,613,163,807,108,170,062,053,440,751,665,152,000,000,000
42!=1,405,006,117,752,879,898,543,142,606,244,511,569,936,384,000,000,000
43!=60,415,263,063,373,835,637,355,132,068,513,997,507,264,512,000,000,000
44!=2,658,271,574,788,448,768,043,625,811,014,615,890,319,638,528,000,000,000
45!=119,622,220,865,480,194,561,963,161,495,657,715,064,383,733,760,000,000,000

NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS   27

SNOBOL4 STATISTICS SUMMARY-


           1048 MS. COMPILATION TIME
           2962 MS. EXECUTION TIME
           3296 STATEMENTS EXECUTED,      437 FAILED
           3376 ARITHMETIC OPERATIONS PERFORMED
              0 PATTERN MATCHES PERFORMED
              0 REGENERATIONS OF DYNAMIC STORAGE
           0.90 MS. AVERAGE PER STATEMENT EXECUTED
```

5. Bridge Dealing Program

The following program uses arrays, programmer-defined functions, and a variety of output formats to produces sets of bridge hands. Execution of the statements beginning at the label NEWDEAL produces one set of hands. Cards are dealt from the array DECK to the four arrays, NORTH, EAST, SOUTH, and WEST by the function DEAL. The hands are sorted by the function ARRANGE. The function DISPLAY prints the hands, one set per page.

```
SNOBOL4 (VERSION 2.0, OCT. 7, 1968)
BELL TELEPHONE LABORATORIES, INCORPORATED


                OUTPUT('TITLE',6,'(14H1THIS IS HAND ,110A1)')
                OUTPUT('DEALER',6,'(11H DEALER IS ,110A1)')
                OUTPUT('SKIP',6,'(A1)')
      *
      *
      *                   FUNCTIONS
      *
      *
                DEFINE('ARRANGE()')
                DEFINE('DEAL()')
                DEFINE('DISPLAY()')
                DEFINE('LINE(STR1,COL1,STR2,COL2)BL1,BL2')
                DEFINE('RANDOM(N)')
                DEFINE('SORT(HAND)I,J')
                DEFINE('SUITL(HAND,SUIT)N')                    :(CONSTANT)
      *
      *
      ARRANGE   SORT(NORTH)  SORT(EAST)  SORT(SOUTH)  SORT(WEST)  :(RETURN)
      *
      *
      DEAL      DEALSEQ  DEALHAND
                DECK  =  COPY(NEWDECK)
                N  =  51
      NLOOP     DEALSEQ  NXTHAND
                CARD  =  RANDOM(N + 1)
                ITEM($HAND,N / 4)  =  DECK<CARD>
                DECK<CARD>  =  NE(CARD,N)  DECK<N>
                N  =  GT(N,0)   N - 1                      :S(NLOOP)F(RETURN)
      *
      *
      DISPLAY   TITLE  =  NTHDEAL
                DEALER  =  DEALR
                SKIP = '        '
                OUTPUT  =  LINE('NORTH',40)
                OUTPUT  =
                OUTPUT  =  LINE(SUITL(NORTH,'S'),40)
                OUTPUT  =  LINE(SUITL(NORTH,'H'),40)
                OUTPUT  =  LINE(SUITL(NORTH,'D'),40)
                OUTPUT  =  LINE(SUITL(NORTH,'C'),40)
                SKIP = '        '
                OUTPUT  =  LINE('WEST',20,'EAST',60)
                OUTPUT  =
                OUTPUT  =  LINE(SUITL(WEST,'S'),20,
      +                         SUITL(EAST,'S'),60)
                OUTPUT  =  LINE(SUITL(WEST,'H'),20,
      +                         SUITL(EAST,'H'),60)
                OUTPUT  =  LINE(SUITL(WEST,'D'),20,
+                        SUITL(EAST,'D'),60)                              34
         OUTPUT  =  LINE(SUITL(WEST,'C'),20,                              35
+                        SUITL(EAST,'C'),60)                              35
         SKIP = '         '                                               36
         OUTPUT  =  LINE('SOUTH',40)                                      37
         OUTPUT  =                                                        38
         OUTPUT  =  LINE(SUITL(SOUTH,'S'),40)                             39
         OUTPUT  =  LINE(SUITL(SOUTH,'H'),40)                             40
         OUTPUT  =  LINE(SUITL(SOUTH,'D'),40)                             41
         OUTPUT  =  LINE(SUITL(SOUTH,'C'),40)                             42
+                                                     :(RETURN)           42
*
*
LINE     BL  LEN(COL1 - 1) . BL1                                          43
         BL  DIFFER(STR2)  LEN(COL2 - (COL1 + SIZE(STR1))) . BL2          44
LINE1    LINE  =  BL1  STR1  BL2  STR2                :(RETURN)           45
*
*
RANDOM   RAN.VAR  =  RAN.VAR * 1061 + 3251                                46
         RAN.VAR  RTAB(5)  =                                              47
         RANDOM  =  (RAN.VAR * N) / 100000            :(RETURN)           48
*
*
SORT     J  =  13                                                         49
SORT1    J  =  GT(J,1)  J - 1                         :F(RETURN)          50
         I  =  0                                                          51
SORT2    I  =  LT(I,J)  I + 1                         :F(SORT1)           52
         TEMP  =  LT(HAND<I - 1>,HAND<I>)  HAND<I - 1> :F(SORT2)          53
         HAND<I - 1>  =  HAND<I>                                          54
         HAND<I>  =  TEMP                             :(SORT2)            55
*
*
SUITL    SUITL  =  SUIT  '  '                                             56
SUITL1   N  =  LT($SUIT + 13,HAND<N>)  N + 1          :S(SUITL1)          57
SUITL2   N  =  LT($SUIT,HAND<N>)  N + 1           :F(RETURN)S(SUITL3)     58
SUITL3   SUITL  =  SUITL  $(HAND<N - 1> - $SUIT)      :(SUITL2)           59
*
*
*                 CONSTANTS
*
CONSTANT BL  =  '                                        '                60
+               '                                        '                60
         S  =  39                                                         61
         H  =  26                                                         62
         D  =  13                                                         63
         C  =  0                                                          64
         $1  =  2                                                         65
         $2  =  3                                                         66
         $3  =  4                                                         67
         $4  =  5                                                         68
         $5  =  6                                                         69
         $6  =  7                                                         70
         $7  =  8                                                         71
         $8  =  9                                                         72
         $9  =  10                                                        73
         $10  =  'J'                                                      74
         $11  =  'Q'                                                      75
         $12  =  'K'                                                      76
         $13  =  'A'                                                      77
         DEALSEQ  =  'NORTH,EAST,SOUTH,WEST,NORTH,'                       78
         NXTHAND  =  *HAND  ','  BREAK(',') . HAND                        79
         DEALHAND  =  *DEALR  ','  BREAK(',') . HAND . DEALR              80
         NORTH  =  ARRAY('0:12')                                          81
```

```
            EAST  =  ARRAY('0:12')                                                  82
            SOUTH  =  ARRAY('0:12')                                                 83
            WEST  =  ARRAY('0:12')                                                  84
            NEWDECK  =  ARRAY('0:51')                                               85
            RAN.VAR  =  157                                                         86
            DEALMAX  =  3                                                           87
            NTHDEAL  =                                                              88
            DEALR  =  'WEST'                                                        89
            N  =  0                                                                 90
   BLDDEK   NEWDECK<N>  =  N + 1                                                    91
            N  =  LT(N,51)  N + 1                             :S(BLDDEK)            92
   NEWDEAL  NTHDEAL  =  LT(NTHDEAL,DEALMAX)  NTHDEAL + 1      :F(END)               93
   *
            DEAL()                                                                  94
   *
            ARRANGE()                                                               95
   *
            DISPLAY()                                         :(NEWDEAL)            96
   END                                                                              97

NO ERRORS DETECTED DURING COMPILATION
```

```
THIS IS HAND 1
DEALER IS NORTH



                         NORTH

                         S   84
                         H   K752
                         D   J3
                         C   Q9742



          WEST                                EAST

          S   AQ63                            S   1075
          H   A98                             H   Q103
          D   862                             D   AK974
          C   K85                             C   A10



                         SOUTH

                         S   KJ92
                         H   J64
                         D   Q105
                         C   J63
```

```
THIS IS HAND 2
DEALER IS EAST



                                   NORTH

                                   S  K82
                                   H  965
                                   D  J75432
                                   C  5



               WEST                                   EAST

               S  J95                                 S  107
               H  K8                                  H  AQ32
               D  AK96                                D  108
               C  AJ84                                C  Q9732



                                   SOUTH

                                   S  AQ643
                                   H  J1074
                                   D  Q
                                   C  K106
```

THIS IS HAND 3
DEALER IS SOUTH

```
                              NORTH

                              S  KJ1093
                              H  J872
                              D  4
                              C  965


             WEST                                 EAST

             S  65                                S  42
             H  KQ1093                            H  6
             D  Q108                              D  AJ765
             C  QJ7                               C  K10842


                              SOUTH

                              S  AQ87
                              H  A54
                              D  K932
                              C  A3
```

```
NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS   93
SNOBOL4 STATISTICS SUMMARY-
           2130 MS. COMPILATION TIME
           5541 MS. EXECUTION TIME
           5736 STATEMENTS EXECUTED,      686 FAILED
           5678 ARITHMETIC OPERATIONS PERFORMED
            378 PATTERN MATCHES PERFORMED
              0 REGENERATIONS OF DYNAMIC STORAGE
           0.97 MS. AVERAGE PER STATEMENT EXECUTED
```

6. A P I A P T --- A Christmastime Algorithm

```
SNOBOL4 (VERSION 2.0, OCT. 7, 1968)
BELL TELEPHONE LABORATORIES, INCORPORATED


     *        WHEN THE OUTPUT ASSOCIATION FOR "SING" IS CHANGED TO
     *        A DIGITAL-TO-ANALOG CONVERTER WITH THE PROPER MELODY
     *        SYNTHESIZER, THIS PROGRAM SINGS THAT OLD CHRISTMASTIME
     *        FAVORITE, "A PARTRIDGE IN A PEAR TREE."
     *
     *                                    M. D. SHAPIRO
     *
     *
              ACAPPELLA.CHOIR = 6 OR MORE PEOPLE SINGING IN TUNE              1
     *
              DAYS = 'FIRST,SECOND,THIRD,FOURTH,FIFTH,SIXTH,'                 2
     .        'SEVENTH,EIGHTH,NINTH,TENTH,ELEVENTH,TWELFTH,'                  2
              NEXT = BREAK(',') . WHICH LEN(1)                                3
     *
              TRACE('SING','VALUE',,'SONG')                                   4
              &TRACE   =   1000                                               5
     *
              DEFINE('SONG()')                              :(NEXT.DAY)       6
     SONG     PAUSE IDENT(SING) OUTPUT('SING', ACAPPELLA.CHOIR,               7
     .        "(' " PAUSE "',100A1)") = '   '               :(RETURN)         7
     *
     NEXT.DAY DAYS NEXT =                                   :F(CODA)          8
              SING = (TAKE A BREATH)                                          9
              SING = 'ON THE ' WHICH ' DAY OF CHRISTMAS,'                     10
              SING = 'MY TRUE LOVE GAVE TO ME,'             :($WHICH)         11
     TWELFTH  SING = 'TWELVE LORDS A-LEAPING,'                                12
     ELEVENTH SING = 'ELEVEN LADIES DANCING,'                                 13
     TENTH    SING = 'TEN PIPERS PIPING,'                                     14
     NINTH    SING = 'NINE DRUMMERS DRUMMING,'                                15
     EIGHTH   SING = 'EIGHT MAIDS A-MILKING,'                                 16
     SEVENTH  SING = 'SEVEN SWANS A-SWIMMING,'                                17
     SIXTH    SING = 'SIX GEESE A-LAYING,'                                    18
     FIFTH    SING = 'FIVE GOLD RINGS,'                                       19
     FOURTH   SING = 'FOUR COLLY BIRDS,'                                      20
     THIRD    SING = 'THREE FRENCH HENS,'                                     21
     SECOND   SING = 'TWO TURTLEDOVES,'                                       22
     FIRST    SING = AND 'A PARTRIDGE IN A PEAR TREE.'                        23
              AND = IDENT(AND) 'AND '                       :(NEXT.DAY)       24
     *
     CODA     SING = INPUT                                  :S(CODA)          25
     *
     END                                                                      26


NO ERRORS DETECTED DURING COMPILATION
```

ON THE FIRST DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
A PARTRIDGE IN A PEAR TREE.

ON THE SECOND DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE THIRD DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE FOURTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE FIFTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
FIVE GOLD RINGS,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE SIXTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
SIX GEESE A-LAYING,
FIVE GOLD RINGS,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE SEVENTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
SEVEN SWANS A-SWIMMING,
SIX GEESE A-LAYING,
FIVE GOLD RINGS,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE EIGHTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
EIGHT MAIDS A-MILKING,
SEVEN SWANS A-SWIMMING,
SIX GEESE A-LAYING,
FIVE GOLD RINGS,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE NINTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,

NINE DRUMMERS DRUMMING,
EIGHT MAIDS A-MILKING,
SEVEN SWANS A-SWIMMING,
SIX GEESE A-LAYING,
FIVE GOLD RINGS,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE TENTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
TEN PIPERS PIPING,
NINE DRUMMERS DRUMMING,
EIGHT MAIDS A-MILKING,
SEVEN SWANS A-SWIMMING,
SIX GEESE A-LAYING,
FIVE GOLD RINGS,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE ELEVENTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
ELEVEN LADIES DANCING,
TEN PIPERS PIPING,
NINE DRUMMERS DRUMMING,
EIGHT MAIDS A-MILKING,
SEVEN SWANS A-SWIMMING,
SIX GEESE A-LAYING,
FIVE GOLD RINGS,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

ON THE TWELFTH DAY OF CHRISTMAS,
MY TRUE LOVE GAVE TO ME,
TWELVE LORDS A-LEAPING,
ELEVEN LADIES DANCING,
TEN PIPERS PIPING,
NINE DRUMMERS DRUMMING,
EIGHT MAIDS A-MILKING,
SEVEN SWANS A-SWIMMING,
SIX GEESE A-LAYING,
FIVE GOLD RINGS,
FOUR COLLY BIRDS,
THREE FRENCH HENS,
TWO TURTLEDOVES,
AND A PARTRIDGE IN A PEAR TREE.

```
     *
    ***
   *****
  *******
 *********
***********
    |||
NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS   25


SNOBOL4 STATISTICS SUMMARY-
            732 MS. COMPILATION TIME
            749 MS. EXECUTION TIME
            276 STATEMENTS EXECUTED,      123 FAILED
              0 ARITHMETIC OPERATIONS PERFORMED
             25 PATTERN MATCHES PERFORMED
              0 REGENERATIONS OF DYNAMIC STORAGE
           2.71 MS. AVERAGE PER STATEMENT EXECUTED
```

13-815357-4