second edition

THE SNOBOL4 PROGRAMMING LANGUAGE

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

Bell Telephone Laboratories, Incorporated

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

Reprinted electronically in 2005 by Ron Stephens & Catspaw, Inc.
by permission of AT&T.

(Originally published by Prentice Hall, Inc., ISBN 13-815373-6)

Library of Congress Catalog Card Number: 70-131996

Printed in the United Stated of America

World Wide Web sites for additional SNOBOL4 material:

Catspaw, Inc.	www.SNOBOL4.com
Phil Budne's SNOBOL4 resources	www.SNOBOL4.org

Foreword

SNOBOL4 is a computer programming language containing many features not commonly found in other programming languages. It evolved from SNOBOL [1,2,3]*, 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, and since most numerical operations involve character counting, integers are more commonly used. Conversion among integers, real numbers, and strings representing integers or real numbers is performed automatically as required. The programmer may, in addition, define other data types, such as complex numbers, and provide operations for them.

Often it is desirable to associate a group of items with one variable name through numerical indexing or some other identifying property. The SNOBOL4 array and table provide these capabilities with more flexibility than most programming languages. An

* Numbers in brackets refer to references listed at the end of this manual.

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. An element of an array is referenced by using an integer index. A table is similar to an array, except that the reference value need not be an integer, but can be any of several other types. Conversion can be made between tables and arrays.

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.

This book 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.

M. D. Shapiro

Lafayette, Indiana
May, 1970

Preface

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 first edition of this book, published in May, 1969, described Version 2. The description in this second edition corresponds to Version 3, released in December, 1969. Version 3 contains a number of features not available in Version 2.

SNOBOL4 has been implemented on several different computers, including the IBM System/360, UNIVAC 1108, GE 635, CDC 3600, CDC 6000 series, PDP-10, Sigma 5/6/7, Atlas 2, and 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 book corresponds to the implementation of SNOBOL4 developed at Bell Telephone Laboratories, Incorporated on 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. Program examples in this book were run on an IBM 360 Model 65.

Acknowledgement

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. R. B. K. Dewar, 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 helpful suggestions for improving the system. Mr. H. J. Strauss designed and implemented the external function interface. Mr. L. C. Varian's assistance in preparing the initial implementation for the IBM System/360 was particularly valuable.

Mr. J. F. Gimpel has made an important contribution to the organization and presentation of descriptive material in this book. Several of the programs used in the examples are his.

The authors' special thanks go to Mrs. M. T. Hammer and Mrs. L. W. Noll for their help in editing and proofreading the second edition of this book. The authors also 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 language and its documentation.

This revised edition was phototypeset by Alphanumeric Incorporated using their TEXTRAN*-2 system. The book was designed by Mrs. M. T. Hammer and Mrs. L. W. Noll. Software developed by Mrs. Noll was instrumental in preparing the book in its present form.

* Servicemark of Alphanumeric Incorporated

Contents

Chapter 1 - Introduction to the SNOBOL4 Programming Language

Chapter 2 - Pattern Matching

Chapter 3 - Primitive Functions, Predicates, and Operations

Chapter 6 - Keywords, Names, and Code

Chapter 7 - Types of Data

Chapter 8 - Tracing

Chapter 9 - Input and Output

Chapter 10 - Running a SNOBOL4 Program

Chapter 11 - Programming Details and Storage Management

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.

1.1 Assignment Statements

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.

1.2 Arithmetic

1.2.1 Integers

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

```
M    =    4
N    =    5
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 (** or !), then multiplication, followed by division, and finally addition and subtraction. All arithmetic 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.

1.2.2 Real Numbers

Real operands are also permitted in arithmetic expressions. The statements

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

assign real values to `PI` and `CIRCUM`.

If real numbers are mixed with integers in arithmetic expressions, the result is a real number. For example, the value of

```
SUM    =    16.4 + 2
```

is `18.4`.

1.3 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    =    "'"
```

Single quotation marks are used in the examples given in this book where one type of quotation mark is sufficient.

1.3.1 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'
' '
```

1.3.2 Strings in Arithmetic Expressions

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

```
Z   =   '10'
X   =   5 * -Z + '10.6'
```

X has the value **-39.4**. Numeral strings representing integers can contain only digits and an optional preceding sign. Numeral strings representing real numbers must have at least one digit before the decimal point. Thus, the following strings cannot be used in arithmetic expressions:

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

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

1.3.3 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 reals and integers as in

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

which gives **SEAT** the value **K22**.

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

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

is equivalent to

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

or

```
SEAT   =   'K24'
```

1.3.4 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, integer or real 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.

Data cards to be read in occur immediately after the end statement that terminates the program.

1.4 Pattern Matching Statements

The operation of examining strings 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:

```
TENS UNITS 30

TENS (UNITS 30)
```

1.5 Replacement Statements

A replacement statement has the form

subject pattern = object

where the fields are separated by at least one blank. Pattern matching is performed as in the pattern matching statement. 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.

1.6 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'
```

Using KEYWORD in the pattern field, the statement

```
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**.

1.7 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

pattern **.** variable

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 **.** associates to the left, and has higher precedence than concatenation and alternation.

```
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.

1.8 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 an identifying 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 preceded by an **F** or **S** corresponding to failure or success. 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. Then the program flows into the end statement causing 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.

An unconditional transfer is indicated by the absence of an **F** or **S** before the enclosing parentheses. 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.

1.9 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')
```

is erroneous because the value of the operand of **$** is a pattern, not a string. Indirect reference in a goto is demonstrated by

```
        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**.

1.10 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.

1.10.1 Primitive Functions

The primitive function **SIZE** has a single string argument and returns as value an integer that 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')
```

is erroneous because the value of the argument is a pattern.

DUPL is another function that performs an operation that is frequently required. **DUPL(string,integer)** returns as value a string that consists of a number of duplications of the string argument. The value of

```
DUPL('/*',5)
```

is **/*/*/*/*/***. **DUPL** returns the null string if the second argument is zero, and fails if it is negative. The statement

```
OUTPUT    =    DUPL(' ',40 - SIZE(S))    S
```

prints the string **S** right justified to column 40 if its length is not greater than 40. Otherwise the statement fails, and **S** is not printed.

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

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

returns as value a string which is the same as **TEXT**, except that each occurrence of a character appearing in **CH1** is 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.

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 a card with the first 40 characters from a card that is read in.

```
INPUT   LEN(40) . PUNCH
```

1.10.2 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 numbers.

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

returns the null string as value if **N1** is a number less than or equal to **N2**. **N1** and **N2** may be either integer or real. 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,

```
        SUM   =   0
        N   =   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 data objects. 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. The predicate **IDENT** is the converse of **DIFFER**. **IDENT** fails if the values of its arguments are not identical.

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.

1.10.3 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(DELETE(INPUT,'.'),' ')
```

deletes all periods and 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(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