The SNOBOL3 Programming Language

By D. J. FARBER, R. E. GRISWOLD and I. P. POLONSKY

(Manuscript received March 4, 1966)

SNOBOL3 is a programming language designed for the manipulation of strings. Features of the language include symbolic naming of strings and pattern matching. In addition to a basic set of primitive string-valued functions, the system includes the facility for defining functions. These defined functions facilitate the programming of recursive procedures.

This paper presents an intuitive description of SNOBOL3 and at the same time incorporates complete reference material for the programmer. The implementation of SNOBOL3 for the IBM 7094 computer operating under BE-SYS-7 is the basis for this description, but most of the material is common to all implementations.

I. INTRODUCTION

In recent years a number of high-level programming languages have been developed to extend the usefulness of the computer in dealing with primarily nonnumerical problems. The most widely used languages have been IPL,[1] LISP,[2] and COMIT.[3] In 1962 SNOBOL[4] was developed for problems involving the manipulation of character strings. The basic operations of SNOBOL permit the formation, examination, and rearrangement of strings. SNOBOL3 is a generalization and extension of SNOBOL. New features include string-valued functions and input-output facilities integrated into the string structure of the language. There are two types of functions: primitive functions that are included in the system and defined functions that are defined by the programmer in the SNOBOL3 language.

This paper is a description of SNOBOL3 as a programming language. Emphasis is placed on the language as distinct from its implementation. In order to provide information for the potential programmer, however, some references to the implementation are necessary. There are several implementations of SNOBOL3 which differ in detail, particularly with regard to input-output. The implementation for the IBM 7094 computer operating under the BE-SYS-7 monitor is the basis for this paper.

Areas where other implementations are likely to differ are noted in the applicable sections.

Section II describes briefly and informally the essential features of the language. This section is designed as a survey to provide an understanding of the general nature and capabilities of the language. Section III is an elaboration of Section II completing the description of the language. Sections II and III together provide a reference source for the programmer. Section IV describes the environment in which the language operates, including information which the programmer will find useful in running programs. A list of primitive functions and sample programs are included in appendices.

II. INFORMAL DESCRIPTION

SNOBOL3 has just one type of basic data structure: a string of characters. The primitive operations of the language provide for the formation, examination and rearrangement of strings. Arithmetic is defined for operands that are integer strings. The operations to be performed are specified in statements that may also be labeled and may have go-to's specifying transfers. A SNOBOL3 program consists of a sequence of statements terminated by an END statement.

2.1 *Names*

A symbolic name can be assigned to a string and used as a means of referring to that string. There are several ways in which a name can be assigned a value. The simplest is the assignment statement. For example, the statement

```
VOWELS = "AEIOU"
```

assigns the string AEIOU as the value of the name VOWELS. The string consisting of a pair of quotation marks enclosing a string of characters is a *literal* specifying the string AEIOU. The string VOWELS appearing to the left of the equal sign is a *name*. A name that has been assigned a value can be used whenever it is necessary to refer to that value. Thus,

```
NON.CONST = VOWELS
```

causes NON.CONST to have the same value as VOWELS. The null string, having length zero, can be assigned explicitly as value as in the statement

```
ZIP =
```

2.2 *Concatenation*

The basic operation of concatenation of strings is indicated by listing the names successively. The names are separated by blanks. Thus, to concatenate two string names STRING1 and STRING2 and then assign the result to the name STRING3, the following assignment statement suffices:

```
STRING3 = STRING1 STRING2
```

Many strings can be concatenated in a string expression, with literals as well as names used to specify the strings. Thus, the following rules

```
ARGUMENT = "2X + 3"
EXPRESSION = "SIN(" ARGUMENT ")"
```

would assign the string SIN(2X + 3) to the name EXPRESSION.

2.3 *Integer Arithmetic*

Arithmetic operations can be performed on integer strings with the operators +, −, /,* having their usual meaning in integer arithmetic and ** indicating exponentiation. Blanks are used to separate the strings and operators. The statements

```
J = "5"
I = "3"
N = I + "2"
M = (I * "3") + J
```

assign the values 5 and 14 to the names N and M. All arithmetic operations are binary but more complex expressions can be constructed using parentheses as indicated in the last example. Arithmetic has precedence over concatenation, and both types of operations can be performed in one assignment statement. Hence, the statement

```
INDEX = "A." I + "1" "." J
```

assigns the value A.4.5 to the name INDEX.

2.4 *Pattern Matching*

String pattern matching consists of examining a string for a succession of substrings of specified form. A pattern-matching statement consists of the string to be examined followed by a pattern. In its simplest form the pattern may be simply a string. For example, the statement

```
NAME.1 "IS"
```

would examine the value of NAME.1 to determine whether it contains the literal substring IS. The success or failure of a pattern match can affect the flow of the program and has other consequences that will be described later. In the example above, NAME.1, which specifies the string to be examined, is called the *string reference* of the statement. The string reference can also be a literal as in the following pattern-matching statement.

```
"+-*/"  OPERATOR
```

There are a variety of types of patterns in SNOBOL3 enabling the programmer to make complex inquiries about a string. The pattern, for example, may be expressed as a concatenation of strings as in the statement

```
EXPRESSION  "X"  OPERATOR  "1"
```

Patterns of greater generality may be obtained by using *string variables*. As the name indicates, a string variable may have a string as value. There are several types of string variables and the strings which are acceptable values of a string variable depend on the type of the variable. The simplest type of string variable is the arbitrary string variable, so named because it can have any string as its value. An arbitrary string variable is designated by a name bounded by asterisks.

A typical example of the use of an arbitrary string variable would be in determining whether the value of NAME.1 contains the string THE and the string IS but not necessarily consecutively. The arbitrary string variable would be used to match the substring between THE and IS. The pattern-matching statement could be

```
NAME.1  "THE"  *SEPARATOR*  "IS"
```

If the value of NAME.1 were THERAPIST, then the pattern match would be successful with *SEPARATOR* matching RAP.

A consequence of the successful pattern match is the naming of substrings that match string variables. In the above example, SEPARATOR would be given the value RAP as if the assignment statement

```
SEPARATOR = "RAP"
```

had been executed.

In addition to arbitrary string variables, there are two other types of string variables: fixed-length and balanced.

A fixed-length string variable can match any string of a specified number of characters. The notation for a fixed-length variable is similar to

an arbitrary string variable, except the name is followed by a slash and then by a string specifying the length.

The first three characters of the string named TEXT could be named PART1 by the rule

```
TEXT      *PART1/"3" *
```

If N had the value 3, the statement could have been written

```
TEXT      *PART1/N*
```

As a second example, consider the statement

```
"+-*"     *PLUS/"1"*     *MINUS/"1"*     *STAR/"1"*
```

The pattern successfully matches the string, and PLUS, MINUS, and STAR are assigned values.

A balanced string variable can only match strings that are parenthesis balanced in the usual algebraic sense. Strings matched by balanced string variables do not have to contain parentheses but cannot be null. Such variables are therefore useful for pattern matching on strings that are mathematical expressions. The notation for a balanced string variable consists of a name enclosed within parentheses and surrounded by a pair of asterisks. For example, if EXPRESSION has the value SIN(A*(B + C)), then the pattern match in the statement

```
EXPRESSION "SIN(" *(ARG)* ")"
```

is successful and ARG is given the value A*(B + C). This use of the balanced string variable may be compared to the arbitrary string variable in the following example

```
EXPRESSION "SIN(" *ARG1* ")"
```

where the value A*(B + C would be assigned to ARG1.

2.5 *Rearranging Strings*

By combining the operations of scanning and assignment in the same rule, strings may be modified by replacement, deletion, or rearrangement. In particular, if a pattern is followed by an equal sign and then by a string expression, the substring matching the pattern will be replaced by the value of the expression if the pattern match succeeds. As an example of replacement, consider the following sequence of rules.

```
CARD = "KING OF HEARTS"
CARD   "HEART" = "DIAMOND"
```

The second statement causes HEART to be replaced by DIAMOND producing the string KING OF DIAMONDS. The following example illustrates how the naming of substrings by string variables may be used in the expression that specifies the rearrangement. The statements

```
SUM = "A1+A2"
SUM    *X* "+" *Y* = "+(" X "," Y ")"
```

change the value of SUM to +(A1,A2).

2.6 *Indirect Referencing*

A level of indirectness can be introduced in SNOBOL3 by prefixing a $ to a name. Thus, if DAY has the value TUESDAY, $DAY is equivalent to TUESDAY. An example of the usefulness of this facility is the ability to modify the naming done in a pattern match. Thus, in the following statements

```
DAY = "TUESDAY"
TEXT      "," *$DAY* ","
```

the name TUESDAY will be assigned to a substring of TEXT if the value of TEXT is such that the pattern match succeeds.

A $ can also be prefixed to a string expression that is enclosed in parentheses. For example, the following statements assign the value of WORD to one of the names LISTA, LISTB, ··· LISTZ according to the first character in the value.

```
WORD *CH/"1"*
$("LIST" CH) = WORD
```

Thus, if WORD has the value DALLAS, the first statement sets the value of CH equal to D. The parenthesized expression

```
("LIST" CH)
```

has the value LISTD. Hence, the effect of the $ is to make the second statement equivalent to

```
LISTD = WORD
```

2.7 *Labels and the Flow of Control*

A label may be assigned to a statement for reference when controlling the flow of the program. The label is merely appended to the beginning of the statement as in

```
HERE  LIST = "(A,B,C,D)"
```

A statement without a label must begin with a blank.

Statements in a SNOBOL3 program are executed in sequential order. This order of execution can be modified by means of a go-to appended to the end of a statement. Go-to's are separated from the rest of the statement by a slash. There are two types of go-to: unconditional and conditional. The unconditional go-to consists of a label enclosed within parentheses. Thus, after executing

```
HERE   LIST = "(A,B,C,D)"          /(THERE)
```

control is transferred to the statement with label THERE. By means of the conditional go-to, control can be transferred depending on whether success or failure has been signaled during the execution of the statement. The letters S and F are used to indicate the two conditions. For example, in the following rule the transfer to the statement with label L2 will occur only if the pattern match is successful.

```
L1     TEXT    "," *A* "."     /S(L2)
```

If the pattern match fails, the next statement in the program is executed.

Transfer on a failure signal can be similarly programmed. As an example, consider the following sequence of statements which will delete from the string named TEXT all occurrences of the characters in LIST:

```
L1     LIST    *CHAR/"1"* =        /F(DONE)
L2     TEXT     CHAR =             /S(L2)F(L1)
DONE
```

In statement L1, the first character in LIST is named CHAR and is deleted by replacing it with a null string. Statement L2 is executed repeatedly until all occurrences of CHAR have been deleted from TEXT. Then the process iterates using the next character in LIST. Finally, when there are no characters left in LIST, the pattern match in statement L1 fails. Thus, if TEXT had the value A+B*C/D+E and LIST the value +*/−, the resulting value of TEXT would be ABCDE.

A transfer may be computed by the use of indirectness in the go-to as illustrated in the following example: If PDL is assumed to have the value A1,B5,C3, then the statement

```
PDL    *RET* "," =      /S($RET)
```

causes deletion of A1, and transfer to the statement labeled A1.

2.8 *Functions*

There are two types of functions in SNOBOL3: primitive and defined. Some functions may signal failure. This failure terminates the execution

of the statement in which the function call occurs and may be used to control the flow of the program.

2.8.1 *Primitive Functions*

A basic set of primitive functions, programmed at the machine-language level, has been included in the SNOBOL3 system.

SIZE is an example of a primitive function. The value of SIZE(X) is the number of characters in the string named X. Thus, the statements

```
STR = "FOUL"
  Z = SIZE(STR)
```

assign the value 4 to Z. As a result of the statements

```
X = SIZE(TEXT) - "1"
TEXT     *FRONT/X*     *LAST*
```

LAST is defined to be the name of the last character in TEXT.

One use of functions is to conditionally signal failure and hence alter the flow of control. For example, EQUALS(X,Y) signals failure if X and Y do not have identical values. If the values of X and Y are identical, the function returns the null string as value. Thus, the statement

```
N = EQUALS(A,B) N + "1"
```

will increment N only if the values of A and B are equal.

Another type of primitive function is one that modifies the behavior of the SNOBOL3 system itself. The function call MODE("ANCHOR") is an example. It modifies the pattern matching processor and returns a null value. Subsequently a pattern match can succeed only if the matching substring is at the beginning of the string reference. Thus, if MODE("ANCHOR") has been executed before the statements

```
EXP = "SIN(A + B)"
EXP "(" *(ARG)* ")"
```

the pattern match fails.

2.8.2 *Defined Functions*

A section of SNOBOL3 program can be defined to be a function and certain names occurring in the section can be declared formal parameters. This function declaration is accomplished by a call of the primitive function DEFINE. For example,

```
DEFINE("REVERSE(X)","REV")
```

declares the section of program beginning at the statement with label REV to be a function named REVERSE with a formal parameter X. Suppose REVERSE(X) returns as value the string named X with the characters reversed. Then the portion of program defining REVERSE could be

```
REV     X               *CHAR/"1"*      =       /F(RETURN)
        REVERSE         = CHAR          REVERSE     /(REV)
```

The reserved label RETURN causes return to the place at which the function was called. The name of the function, in this case REVERSE, serves a special purpose. When the function is called, its value is saved and then set to the null string. When transfer to RETURN occurs, its value is the value returned by the function. Thus,

```
Z = REVERSE("ABCDE")
```

assigns the value EDCBA to Z.

2.9 *Statement Format*

A SNOBOL3 statement has a simple format consisting of several fields of arbitrary length separated by blanks. The fields are the label, string reference, pattern, equal sign, replacement expression, and the go-to. A statement may contain some or all of these fields. The replacement statement

```
HERE    TEXT    " "    *WORD*    " " = " "    /S(GOT)
```

has all of the fields.

If the label is omitted the statement must begin with a blank. If the next field is not a go-to, it is considered to be the string reference. Thus,

```
                        /(L7.3)
THERE                   /(HERE)
```

are statements that do not have a string reference. The statement

```
L5      EQUALS(OP,"END")        /S(END)
```

has the string reference EQUALS(OP,"END").

The field following the string reference up to an equal sign or a go-to is the pattern. In a statement without a go-to or equal sign, the pattern is the field following the string reference. Thus, in each of the following statements

```
      STR    A  *B* ","  = B     /S(GREAT)
      STR    A  *B* ","  =
L1    STR    A  *B* ","          /S(GREAT)
      STR    A  *B* ","
```

the pattern is

```
A  *B* ","
```

Note that the elements within the pattern are also separated by blanks.

A statement without a pattern, but containing an equal sign, is an assignment statement. Some examples are:

```
ANS = N + "5"
RES =                 /(READ)
```

The latter example assigns the null string as the value of RES.

No fields are permitted after the go-to field.

III. DETAILED DESCRIPTION

The previous section was an informal description of the basic parts of SNOBOL3. The following section completes this description in a more comprehensive and detailed manner.

3.1 *Names and String Expressions*

3.1.1 *Names*

Names are used to refer to string values symbolically. In addition, names are required for certain parts of statements:

(*i*) string variable names
(*ii*) string references in assignment statements
(*iii*) labels in go-to fields.

Names may be explicit or implicit. Explicit names can consist only of letters, numbers, periods and colons. Examples are:

```
N
STATEMENT.VARIABLE
X:1
37
```

Implicit names, constructed by indirect references, may consist of any nonnull string of characters. Any indirect reference is an implicit name.

For example:

```
$F
$SIZE(N)
$("M" K)
```

Consequently,

```
,,,, = "6"
```

is syntactically *incorrect*, but

```
INAME = ",,,,"
$INAME = "6"
```

is proper.

The particular characters comprising a name have no significance; a name is merely an identifier. A name may be the same as a label or the name of a function.

3.1.2 *String Expressions*

The basic string-valued elements are:

(*i*) literals
(*ii*) names
(*iii*) function calls
(*iv*) arithmetic operations
(*v*) parenthetical groupings.

Any string of characters (including the null string) not containing a quotation mark (see Section 3.1.3) may be included between the quotation marks of a literal. Function calls, parenthetical groupings, and names may be indirectly referenced. Parentheses are required between successive levels of indirect references.

A string expression is a string-valued element or the concatenation of several such elements. Some typical string expressions are:

```
"PARAGRAPH SUB-HEADINGS FOLLOW"
"N" ((A + "1") * INTERVAL)
$BASE + SIZE(N)
F(X,F(X,X))
$($($ROOT))
M "." P
$("N" I)
```

3.1.3 *Names and Values*

All names have null values at the beginning of program execution except for the string QUOTE. QUOTE has a preassigned value which is a quotation mark.

Names, including QUOTE, may subsequently be given other values by assignment statements or as a result of pattern matching. The resulting name-value relationship between strings forms the basic data structure in SNOBOL3. Structures can be built to arbitrary depths. For example, the statements

```
N1 = "N2"
N3 = "N2"
N2 = "N4"
N4 = "N6"
N5 = "N4"
N6 = "N3"
```

might be used to represent relationships between data as indicated in Fig. 1.

Indirect referencing can be used to refer to the relationships in the structure. The range of such structures is limited by the fact that a name can have at most one value at any time, while a string can be the value of any number of names simultaneously.

3.2 *Arithmetic*

3.2.1 *Integers*

Some strings have the property of being SNOBOL3 integers. Such strings are required in arithmetic operations and as arguments of certain primitive functions. In order for a string to be a SNOBOL3 integer

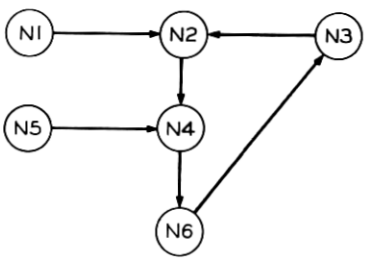

Fig. 1 — The name-value relationship among data.

(*i*) it must consist entirely of digits except for the first character which may be a sign, and
(*ii*) its absolute value considered as a decimal integer must be less than 10^{10}.

In numerical contexts
(*i*) unsigned numbers are taken to be positive,
(*ii*) leading zeros are ignored,
(*iii*) minus zero is equal to plus zero, and
(*iv*) the null string is taken to be zero.

The following strings are SNOBOL3 integers:

```
5
+10
0003976
-37
-000003
+0
```

The following strings are *not* SNOBOL3 integers:

```
-
+A
3.27E-2
3.7
876935476271
0-
10,000
```

The primitive function .NUM(X) succeeds, returning a null value, if the value of X is a SNOBOL3 integer and fails otherwise. Thus, .NUM("A") fails, while .NUM("100") returns a null value.

3.2.2 *Arithmetic Expressions*

Arithmetic operations must be separated from their operands by blanks. Consequently A+B is syntactically incorrect. Any expression whose value is a SNOBOL3 integer is an acceptable operand.

All arithmetic operations are binary. Thus,

```
N + "3"
```

is a legal operation, while

```
N + "3" * "2"
```

is a syntactic error. Parentheses may be used for grouping terms to create more complicated expressions:

N + ("3" * "2")

In expressions containing both concatenation and arithmetic, arithmetic has precedence over concatenation. Thus, the value of

"N" "5" + "7"

is N12 and the value of

"3" * "2" "10" / "2"

is 65. Parentheses may be used to group concatenations and arithmetic to obtain the desired result. Thus, the value of

"3" * ("2" "10" / "2")

is 75.

The following sequence of statements illustrates possible combinations:

```
ALPHA = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    N = SIZE(ALPHA) + "1"
    M = (N + SIZE(N)) * "2"
    K = ("-" N M) + "5"
```

As a result of executing these statements, N would have the value 27, M the value 58, and K the value −2753.

The result of an arithmetic expression is a *normalized* SNOBOL3 integer. Integers are normalized as follows:

(*i*) positive integers are unsigned,
(*ii*) leading zeros are removed, and
(*iii*) any value equal to zero is returned as an unsigned zero.

Thus,

"+0003" + "0"

has value 3, and

"" * "2"

has the value 0.

3.2.3 *The Evaluation of Arithmetic Expressions*

Two modes for evaluating arithmetic expressions are available. The normal mode is *truncation*. In the truncation mode any fractional part

resulting from division (or exponentiation) is discarded. Thus, the value of

"5" / "2"

is 2, and the value of

"3" ** "−1"

is 0.

An *integer* mode is available which causes an arithmetic operation to fail if a fractional part would result. The integer mode may be invoked by executing the function call MODE("INTEGER"). The normal mode may be restored by executing MODE("TRUNCATION").

3.2.4 *Error Conditions in Arithmetic Operations*

Error conditions in arithmetic operations occur if:

(*i*) a fractional part would occur in integer mode,
(*ii*) an operand is not a SNOBOL3 integer,
(*iii*) the result of an arithmetic operation is not a SNOBOL3 integer (because it is too large), or
(*iv*) division by zero is attempted.

In all cases, the arithmetic operation fails, terminating the execution of the rule in which it occurs. The failure may be utilized to change the flow of control by means of a conditional go-to.

3.2.5 *Numerical Functions*

There are six functions for comparing the magnitude of integers:

.EQ(X,Y)	(X = Y)
.NE(X,Y)	(X ≠ Y)
.LT(X,Y)	(X < Y)
.LE(X,Y)	(X ≦ Y)
.GT(X,Y)	(X > Y)
.GE(X,Y)	(X ≧ Y).

These functions succeed, returning a null value, if the condition indicated is satisfied and fail otherwise. The functions also fail if either argument is not a SNOBOL3 integer. A common use of the functions is to control loops. For example, the following program assigns the squares of the first 100 positive integers to the names SQ1 through SQ100, respectively.

```
          N = "1"
COMPUTE   $("SQ" N) = N * N
          N = .LT(N,"100") N + "1"   /S(COMPUTE)
```

The function .REMDR(X,Y) has as its value the remainder of X divided by Y. For example, the value of

.REMDR("5", "2")

is 1. The sign of the remainder is the same as the sign of the divisor and the value is normalized.

.REMDR fails if either argument is not a SNOBOL3 integer or if the value of Y is zero.

3.3 *Pattern Matching*

Pattern matching is a basic operation in SNOBOL3. The examination, rearrangement, and combination of data depend on pattern matching; and the success or failure of matching is often used for altering the flow of control.

3.3.1 *Pattern Elements*

A pattern consists of a succession of pattern elements separated by blanks. There are two basic categories of pattern elements: *string constants* and *string variables.*

Any string expression is a constant, except that arithmetic expressions must be enclosed in parentheses. The following expressions are examples of string constants:

```
K
"35R"
SIZE(Z)
.LE(N,M + SIZE(L))
(M + (N * "2"))
```

String variables may or may not have associated names. The following elements are examples of string variables:

```
**
*()*
*/"3"*
*VARIABLE1*
*$SIZE(N)*
*(EXP)*
```

The length of a fixed-length variable may be any string constant whose value is a nonnegative SNOBOL3 integer when evaluated. The following fixed-length variables illustrate possible forms the length may take:

```
*FL/N*
*V/(SIZE(N) + "1")*
*V/(M + (N * Z))*
```

The lengths of the following variables are syntactically *incorrect:*

HEAD/N + "1"	(Arithmetic expressions must be enclosed in parentheses.)
SPAN/"A"	(The value of the length must be an integer.)

3.3.2 *The Matching Process*

Pattern matching consists of three phases:

(*i*) evaluation of expressions in the pattern,
(*ii*) the actual matching, and
(*iii*) the assignment of values to names associated with string variables.

3.3.2.1 *Evaluation.* Before any matching, all expressions in the pattern are evaluated. Expressions may occur in string constants, the names of string variables, and in the length of fixed-length variables. Evaluation proceeds from left to right. Any failure in evaluation (such as the failure of a function call or arithmetic operation) terminates the execution of the rule without any matching or naming.

The value of all expressions is fixed by evaluation before matching. No evaluation is performed during matching. The only exception to this rule is back referencing described in a following section. Thus, in the pattern

```
*N*     *SPAN/N*
```

the length of the fixed-length variable is the value of N *before* matching and is not influenced by any subsequent match for the arbitrary string variable with the name N.

3.3.2.2 *Matching.* Pattern elements must match consecutive substrings in the value of the string reference. In most cases the match can easily be determined from the following rule:

Pattern matching proceeds from left to right, each pattern element matching the shortest possible substring according to the type of the element.

In some complicated cases, more precise definitions are necessary. The following definitions provide the details for resolving difficult cases.

(*i*) The pattern match proceeds element by element from left to right starting at the leftmost (first) element. The elements must match consecutive substrings in the value of the string reference.

(*ii*) An attempt is first made to match the first element starting at the first character in the value of the string reference. If this is not possible, an attempt is made starting at the second character, and so on.

(*iii*) When an element is successfully matched, a *forward* match is attempted for the next element.

(*iv*) If an element cannot be matched, *rematch* is attempted for the preceding element. Rematching is an attempt to extend the substring matching a pattern element and occurs because the pattern match cannot be successfully concluded with the previous match.

(*v*) Pattern matching terminates successfully when the rightmost (last) pattern element has been matched. Pattern matching terminates in failure if no match can be found for the first element.

The methods of forward matching and rematching depend on the type of the pattern element. In each case, the element must match a substring in the string reference starting at the character following the substring matching the preceding element. The details follow.

(a) *String Constants*

In forward matching, a string constant matches a substring identical to its value. If this is not possible, forward matching fails. A null constant always matches.

No rematch is possible, and rematching always fails.

See the special case of back referencing.

(b) *Arbitrary String Variables*

In forward matching, an arbitrary variable matches a null string.

In rematching, one character is added to the substring previously matched by the variable. If the string reference is not long enough for such a match, rematching fails.

As a special case, if the last element in the pattern is an arbitrary string variable, it matches the remainder of the string.

(c) *Balanced String Variables*

In forward matching, the string matched by a balanced variable depends on the first character of the substring where the variable is to match. If this first character is not a parenthesis, then the variable matches that character. If the first character is a right parenthesis, the match fails. If the first character is a left parenthesis, the string being examined is considered character by character until a matching right parenthesis is found. If there is no matching parenthesis, failure is indicated. Notice that a balanced string variable always matches at least one character.

In rematching, the previously matched substring is extended by the next shortest balanced string according to the rules for forward matching. If this is not possible, rematching fails.

(d) *Fixed-Length String Variables*

In forward matching, a fixed-length variable matches a substring of length specified by the variable. If the string being examined is not long enough, forward matching fails.

Rematching always fails.

(e) *Back Referencing*

Back referencing is a special case in pattern matching in which tentatively matched substrings can be referred to dynamically during the matching process. If a constant in the pattern has the same name as a name associated with a variable to the left of it in the pattern, the value of the constant is taken to be the substring currently matched by the variable. Thus, in the pattern

```
*N* ',' N
```

the constant N must match a substring identical to the substring matching *N*. Since matching is done left to right, a tentative match always exists for a back-referenced variable.

Back referencing only occurs when the name associated with a variable is to the left of a constant with the same name. Consequently the pattern

```
N ',' *N*
```

does not contain back referencing.

If there are several occurrences of the same name in a pattern, a

named constant back references the variable with its name which is nearest to it on the left. In the pattern

```
*N* "," N N *N* "," N
```

the first and second named constants refer to the first variable and the third named constant refers to the second variable.

Any type of variable may be back referenced and any number of named constants may back reference variables in an arbitrarily complicated way.

The determination of back referencing within a pattern is made after the evaluation of expressions in the pattern but before matching. In the statements

```
A = "C"
B *C* "," $A
```

the pattern is back referenced. However, in the statements

```
VARI = "SPAN"
X  *VARI*  $VARI
```

there is no back referencing.

Back referencing only applies to names which are pattern elements and not to any other name in the pattern. Specifically in the pattern

```
*N*  *INT/SIZE(N)*
```

the length of INT is determined by evaluation before matching and does not change during the matching process.

3.3.2.3 *Naming.* If the pattern match fails, no naming is done and the execution of the rule is terminated. If the pattern match succeeds, naming is performed from left to right for each name associated with a string variable. The substring matching the variable becomes the new value of the associated name. If a name is associated with more than one variable, the value is assigned corresponding to the rightmost variable with that associated name.

In the case that a name is computed as the result of an expression, the name is determined by the evaluation made before pattern matching. Thus, in the statements

```
A = "C"
Z  *A*  ","  *$A*
```

the name associated with the second string variable is C regardless of the value of Z.

3.3.3 *Pattern Matching Modes*

In the normal mode of pattern matching, the first element of the pattern may match starting anywhere in the value of the string reference. Thus, the simple match

"0123456789" "6"

succeeds. This mode is referred to as *unanchored.* The alternative mode, in which the first pattern element must match a substring beginning with the first character of the string reference, is called *anchored.* This mode may be invoked by executing the function call

MODE("ANCHOR")

Subsequently, all pattern matching will be in the anchored mode unless otherwise modified. The normal mode may be restored by

MODE("UNANCHOR")

The mode of matching may be changed for the duration of a single statement by means of the two functions ANCHOR and UNANCH. These functions, which have no arguments, must be called before matching (Refer to the Section 3.4.3). Both functions return null values. Thus, in

Z ANCHOR( ) "." *IDENT* "."

the pattern match is anchored regardless of the matching mode current in the program. Subsequent statements are not affected. The null value returned by ANCHOR does not affect the match since a null string may match anywhere.

ANCHOR and UNANCH supersede the MODE function even if the MODE function is executed subsequently in the evaluation of the pattern elements. Hence in the statement

STRING UNANCH( ) MODE("ANCHOR") "."

the pattern match is unanchored, although the anchored mode will subsequently prevail.

3.3.4 *Examples of Pattern Matching*

The following examples illustrate some of the situations which occur in pattern matching. String reference values are given as literals for clarity. Naming is indicated for those pattern matches which succeed. The normal unanchored matching mode is assumed.

Example 1:

"K)AK(A + B + C)ST" "K" *(A)* "ST"

The match succeeds with

A = "(A + B + C)"

Example 2:

"K)AK(A + B + C)ST" ANCHOR( ) "K" *(A)* "ST"

The match fails.

Example 3:

"S)(S + A*B(S" "S" *(A)* "S"

The match fails.

Example 4:

"ABCDEFGHIJKLMNO" *HV/"5"* *A* "K" *B*

The match succeeds with

HV = "ABCDE"
A = "FGHIJ"
B = "LMNO"

Notice that since the last pattern element is an arbitrary string variable it matches the remainder of the string reference.

Example 5:

"364:" *A* *SUM/"3"* ":"

The match succeeds with

A = " "
SUM = "364"

Example 6:

"ARMY" *A* *B* *C*

The match succeeds with

A = " "
B = " "
C = "ARMY"

Notice that the first two arbitrary string variables match null strings since this satisfies the requirement for matching the shortest possible substrings.

Example 7:

"ABC" *(BAL1)* *(BAL2)*

The match succeeds with

BAL1 = "A"
BAL2 = "B"

Example 8:

"AB" *(BAL1)* *(BAL2)* *(BAL3)*

The match fails since each balanced string variable must match at least one character.

Example 9:

"ABCD" *S/"2"* *T/"3"*

The match fails since the string being matched is not long enough.

Example 10:

"ABCDEFGHFGH" *A/"3"* A

The match succeeds with

A = "FGH"

This is a simple example of back referencing.

Example 11:

"ABCDEFGHFGH" ANCHOR( ) *A/"3"* A

The match fails.

Example 12:

"32579.97" *A* *B* "." B A

The match succeeds with

A = "7"
B = "9"

These values can be verified by carefully applying the matching rules. (The expected match might be a null value for both A and B.)

Example 13:

The following example illustrates the complexity which may occur with back referencing.

```
"BACCABACABABACACAB"  *A*  *(B)*  *(C)*
                  *D*  C  D  B  D  C  A  *E*  A  E
```

The match succeeds with

A = " "
B = "BAC"
C = "CAB"
D = "A"
E = " "

Example 14:

```
"A,A,B,B"   *X*   ","   X   ","   *X*   ","   X
```

The match succeeds with

X = "B"

3.4 *Program Structure and the Flow of Control*

A program consists of a succession of statements terminated by an END statement containing the reserved label END. The END statement may also contain a name which is the label of the first statement to be executed. If the END statement contains no name, execution begins with the first statement of the program.

Statements are subsequently executed one after another unless control is transferred by means of a go-to.

3.4.1 *Labels*

Labels are distinguished by beginning in Column 1. A statement with no label must have a blank in Column 1. The first character of a label must be a letter or a digit. Subsequent characters may be anything but blanks. Labels are program constants; the particular characters in a label have no significance even if they resemble some other structure such as a name or a function call. Thus F(X) is a legitimate label but has no further meaning.

3.4.2 *The Go-To Field*

Go-to's are used to alter the ordinary sequential execution of statements. In general, a statement may be successfully completed, or failure may be indicated for a number of causes. The success or failure may be sensed and used by corresponding conditional go-to's to alter the order in which statements are executed.

A statement with an unconditional go-to may not have conditional go-to's. Furthermore, a statement may not have more than one unconditional, success or failure go-to. In statements with both success and failure go-to's, the go-to's may occur in either order.

The labels given in the go-to's must be names and transfer is made to the name (not its value). The label in a go-to may be computed by the use of implicit names resulting from indirect references. For example, in the statement

```
X = "3"      /($("R" X))
```

transfer is made to the statement with label R3. Function calls occurring in go-to's must not fail.

3.4.3 *The Order of Execution Within a Statement*

The order of execution of operations within a statement may be important to the programmer for two reasons:

(*i*) Failure of an operation within a statement terminates execution of the statement at that point so that subsequent operations are not performed.

(*ii*) Calls of defined functions may change the values of names which appear subsequently in the same statement.

Consequently, a detailed knowledge of when various parts of a statement are evaluated may be required to determine how a program will function. The overall order of execution within a statement is as follows:

(*i*) The string reference (if any) is evaluated.

(*ii*) The elements of the pattern (if any) are evaluated from left to right (see Section 3.3.2).

(*iii*) The pattern match (if any) is performed.

(*iv*) Any naming as the result of a successful pattern match is performed.

(*v*) If a string expression is specified as a replacement, that string expression is evaluated.

(*vi*) Reformation (if specified) of the value of the string reference is made.

(*vii*) The go-to (if any) corresponding to the success or failure of the statement is evaluated.

(*viii*) Transfer is made to the next statement accordingly.

If failure is signaled in any of the steps (*i*) through (*vi*) above, execution of the statement terminates at that point and the appropriate go-to is evaluated. In particular note that *only* the appropriate go-to is evaluated. The order of evaluation within a string expression is as follows:

(*i*) Elements in a concatenation are evaluated left to right.

(*ii*) In a function or parenthetical grouping, the innermost expression in the nesting is evaluated first.

(*iii*) Arithmetic is performed before concatenation.

(*iv*) All arguments of a function are evaluated, left to right, before the function is called.

3.4.4 *Termination of Execution*

Program execution is usually terminated by a transfer to the label END or by flowing into the END statement.

Depending on the monitor system under which SNOBOL3 operates, the termination of a SNOBOL3 program may or may not terminate the job which initiated the SNOBOL3 program. Thus, two modes may be distinguished:

(*i*) *endjob*, in which job execution is terminated upon completion of the SNOBOL3 program, and

(*ii*) *system*, in which job execution may continue after completion of the SNOBOL3 program.

The normal mode in SNOBOL3 is endjob. The alternative mode may be invoked by the function call MODE("SYSTEM"). The normal mode may be restored by the function call MODE("ENDJOB")

Execution of a SNOBOL3 program may be interrupted by the function SYSTEM(FILE). A call of SYSTEM(FILE) suspends program execution, returning control to the monitor. The input source for the monitor is set to the value of FILE. If the value of FILE is null, the input source is set to the standard input source. The availability of SYSTEM depends on the monitor under which SNOBOL3 operates.

3.5 *Input-Output and File Manipulation*

3.5.1 *Implementation Differences*

Input-output is particularly subject to differences in machines and monitor systems. Consequently the input-output behavior of the

SNOBOL3 system may vary considerably in different implementations. Reference to files, record sizes and the handling of end-of-file differ most. The following sections should be read with this in mind.

3.5.2 *String-Oriented Input and Output*

Input and output is accomplished through string names associated with logical files.

For example, SYSPOT ("system peripheral output tape") is associated with the standard output file. Every time SYSPOT is given a value, a copy of the value is printed on the system output file. Thus, the statement

```
SYSPOT = "TABLE OF VALUES"
```

will cause the printing of TABLE OF VALUES on the output listing.

SYSPPT ("system peripheral punch tape") is associated with the standard punch file. Values given SYSPPT are punched rather than printed.

Similarly, SYSPIT ("system peripheral input tape") is associated with the standard input file. Every time the value of SYSPIT is required, a card image is read from the input file to become the value of SYSPIT. For example, the statement

```
SYSPIT    *FIELD1*    ","    *FIELD2*    " "
```

might be used to read and name data items on input cards with the format indicated by the pattern.

3.5.3 *The Association of String Names with Files*

The names SYSPOT, SYSPPT, and SYSPIT are automatically associated with standard files at the beginning of program execution. Any name (including these three) may be associated with any file during program execution by means of association functions. All the following functions return null values.

(*i*) PRINT(NAME,FILE) associates the value of NAME with the value of FILE in the print sense. Thus,

```
PRINT("X", "OUT")
```

associates the name X with logical file OUT. After execution of this function call, copies of all values assigned to X will be placed on the file OUT.

(*ii*) PUNCH(NAME,FILE) associates the value of NAME with

the value of FILE in the punch sense. The distinction between punch and print association is described in the next section.

(*iii*) READ(NAME,FILE) associates the value of NAME with the value of FILE in the read sense.

The execution of an association function detaches the name from any file with which it may be associated. Thus,

PUNCH("SYSPOT", "SCR")

is permissible.

Any nonnull name may be associated with any file. If the value of FILE is null, the name will be associated with the appropriate standard file. A name may be associated with only one file, but any number of names may be associated with a file.

3.5.4 *Output*

Output occurs whenever an output-associated name is given a value. Thus,

FIELD *SYSPOT* "," *SYSPOT/"5"*

results in two outputs if the pattern match is successful.

Print and punch association differ in carriage control. When output is performed on values whose names are associated in the print sense, six blanks are prefixed to provide carriage control. No carriage control is provided for output resulting from punch association. Consequently, punch should be used for intermediate files which are to be subsequently read.

Printing on the standard print file is 126 characters per line (not counting carriage control). Additional lines are generated as necessary for longer strings. Punching on the standard punch file is 72 characters per card with additional cards generated as necessary. Values punched or printed on other files are augmented with blanks to an even multiple of six characters. Resulting strings containing 84 characters or less are printed as single records of the length of the augmented string. Longer records are printed 84 characters per record. Any residual string over a multiple of 84 characters is printed as a record of the residual length.

Printing a null value always produces a record because of the six blanks prefixed for carriage control. Punching a null value does not produce a record.

Since output of strings may break one string into many records, care must be taken that the strings may be properly reconstructed if neces-

sary. This remark also applies to padding with blanks and handling of null strings as described above.

Names associated with output files retain their values like any other names. The output process does not destroy values.

3.5.5 *Input*

All strings read from the standard input file are 84 characters long. Blanks are used to fill out shorter records. Records read from other files are not extended.

It is particularly important to notice that any use of the value of a read-associated name results in the reading of a record and the loss of the previous value of the name. This is true regardless of context. For example,

```
SYSPIT      *Z*      ","
```

results in the reading of a record. The record is destroyed by any subsequent use of SYSPIT. Consequently, it is good programming practice to assign the result of reading to some other string. For example,

```
SYSPOT = SYSPIT
```

prints the next record, and the resulting string remains as the value of SYSPOT for further use.

An important aspect of a read-associated name is the indication of failure if the read operation fails (as the result of an end-of-file or binary record). Such a failure terminates execution of the statement in which it occurs and may be used by a conditional go-to. The ability of the SNOBOL3 system to regain control after an end-of-file depends on the monitor under which SNOBOL3 operates.

3.5.6 *Other Functions*

Several functions exist for performing standard input-output operations and file manipulation. All these functions return null values.

(*i*) DETACH(NAME) removes the value of NAME from any input-out association. For example,

```
DETACH("SYSPOT")
```

terminates normal print output. If the value of NAME is not associated with a file, no action is taken.

DETACH may be used to save the value of a name associated in the read sense. For example, the statements

```
SYSPIT    *A*    ","    *B*    ","    *C*    /S(PROG)
DETACH("SYSPIT")    /(ERROR)
```

might be used to go to an error routine in case an input record does not have the expected format. By detaching SYSPIT, the record in error may be examined without destroying it.

(*ii*) REWIND(NAME) rewinds the file associated with the value of NAME. Note that the argument is the name and not the file. An end-of-file is written on a file in output status before it is rewound.

(*iii*) BSREC(NAME) backspaces one record on the file associated with the value of NAME.

(*iv*) EJECT(NAME) writes an eject carriage control character on the file associated with the value of NAME.

(*v*) OPEN(KEY,FILE) opens the specified file in the key area which is the value of KEY. The applicability of this function depends on the monitor under which SNOBOL3 operates.

3.6 *Primitive Functions*

3.6.1 *Function Calls*

Function calls may occur anywhere in a statement where a string value is appropriate. An argument of a function may be any string expression, however complicated. Any argument may be explicitly null and trailing arguments that are omitted are given null values. Thus, the three function calls

```
EQUALS(X," ")
EQUALS(X,)
EQUALS(X)
```

are equivalent. A primitive function may be called with up to six arguments, regardless of the number specified by the function. Such additional arguments are evaluated but are ignored by the function.

All function calls return strings as value if they succeed. In the case of functions that have no natural value, a null string is returned.

It is important to notice that the function name and the left parenthesis may not be separated by blanks. Thus,

```
SIZE(N)
```

is a function call, while

```
SIZE (N)
```

is the concatenation of a name and a parenthetical grouping. Similarly, functions such as ANCHOR( ) which have no argument must be written with the parentheses. Otherwise they will be taken for string names rather than function calls.

3.6.2 *Functions Relating to Functions*

(*i*) OPSYN(NEW,OLD) OPSYN permits the programmer to associate a new name with a function. Thus, for example,

OPSYN("LENGTH","SIZE")

makes the name LENGTH a function with the same definition as SIZE. Either LENGTH or SIZE may now be used to call the function. New names may be associated with either primitive or defined functions. OPSYN returns a null value.

(*ii*) CALL(FNC) CALL permits the programmer to invoke a function implicitly by interpreting a string as a function call. The value of FNC must correspond syntactically to a function call.

For example, the function call

CALL("SIZE(M)")

is equivalent to the function call

SIZE(M).

The arguments in FNC are interpreted as explicit names. Thus, all arguments in a function invoked by CALL must be assigned names. For example, to invoke

.GT(SIZE(N),"5")

by the use of CALL, statements of the form

```
ARG1 = SIZE(N)
ARG2 = "5"
CALL(".GT(ARG1,ARG2)")
```

are required.

Any primitive or defined function may be invoked by CALL. Value is returned and success or failure indicated in the same manner as if the function call appeared explicitly.

3.6.3 *Miscellaneous Primitive Functions*

There are six primitive functions in addition to the functions described elsewhere in Section III. They are:

(*i*) EQUALS(X,Y) EQUALS returns a null value if the value of X is identical to the value of Y and fails otherwise. The values must be identical and not just numerically equal (compare Section 3.2.5).

(*ii*) UNEQL(X,Y) UNEQL returns a null value if the value of X is not identical to the value of Y and fails otherwise.

(*iii*) TRIM(S) TRIM returns as value the value of S with trailing blanks removed, thus, for example, TRIM(SYSPIT) is a convenient method of removing superfluous blanks from input cards.

(*iv*) TIME( ) TIME is a function of no arguments which returns as value the millisecond time from the beginning of program compilation. The value is returned as a 6-character number.

(*v*) DATE( ) DATE is a function of no arguments which returns as value the current date. The value is returned as a 6-character number. For example, April 1, 1966 would appear as

040166

(*vi*) SIZE(S) SIZE returns as value the number of characters in the value of S. For example, the value of SIZE ("0123456789") is 10.

3.6.4 *Addition of Primitive Functions to the SNOBOL3 System*

The SNOBOL3 system is designed so that separately-compiled primitive functions may be added easily. This facility has been used extensively to add a wide range of capabilities.[5,6,7] A discussion of the format and communication conventions required for primitive functions is beyond the scope of this paper. Ref. 8, which is available from the authors, describes these matters in detail.

3.7 *Defined Functions*

3.7.1 *The Definition of a Function*

A defined function is characterized by four items:

(*i*) a name, by which it is called and which is used for returning value,

(*ii*) a list of formal arguments, used for passing values to the function,

(*iii*) a label, indicating its entry point, and

(*iv*) a list of local names used by the function.

A function must be defined during program execution before it is used. This definition is accomplished by a call of the DEFINE function which

establishes the four items above. The form of the call is

DEFINE(FORM,LABEL,NAMES)

FORM is a prototype of the function call, giving the function name and the list of formal arguments. The value of LABEL is the entry point, and the value of NAMES is the list of local names separated by commas. For example,

DEFINE("FACT(N)","F")

defines a function FACT with one formal argument N. Execution of FACT is to begin at the label F. No local names are declared. Similarly,

DEFINE("MATRIXADD(A,B)","MA","I,J,K")

defines a function MATRIXADD with the two formal arguments A and B, the entry point MA and local names I, J, and K.

The total number of formal arguments and local names must not exceed ten. This limit is an assembly parameter.

3.7.2 *The Execution of Defined Functions*

The call of a defined function is identical to the call of a primitive function (see Section 3.6.1). Hence, trailing arguments which are omitted are given null values. A defined function, however, may not be called with more arguments than given in its definition.

When a defined function is called, values of the following names are saved:

(*i*) the name of the function.
(*ii*) all formal arguments.
(*iii*) all local names.

New values are assigned to these names as follows:

(*i*) the name of the function is given a null value.
(*ii*) the formal arguments are assigned values by evaluating the corresponding arguments in the function call.
(*iii*) all local names are assigned null values.

Saving of old values and assignment of new values is made from left to right as the names appear in the DEFINE call.

After these new values have been assigned, control is transferred to the entry point of the function and program execution continues in a normal fashion until transfer is made to one of the two reserved labels RETURN or FRETURN.

RETURN terminates execution of the function. By convention, the

value of the function call is the value of the function name when the return was made. For example, if FACT as defined above is designed to compute the factorial of a number, the corresponding program might be

```
F  FACT = .EQ(N,"0") "1"   /S(RETURN)
   FACT = FACT(N - "1") * N  /(RETURN)
```

Then the statements

```
SYSPOT = FACT("3")
SYSPOT = FACT("2") + FACT("4")
```

would print 6 and 26, respectively.

FRETURN terminates execution of the function and signals failure. Execution of the statement in which the call occurs terminates at that point in the same manner as in the failure of a primitive function.

When return is made from a function (by either RETURN or FRETURN) the saved values of all names are restored in the opposite order from which they were saved.

A function may have a formal argument which is the same as its name. This is useful when the value of a function is to be a simple modification of one of its arguments. A function whose value is its first argument with all occurrences of its second argument deleted might be defined as

```
DEFINE("DELETE(DELETE,CHAR)","DEL")
```

with the program

```
DEL  DELETE  CHAR = /S(DEL)F(RETURN)
```

Here DELETE can be operated on as desired and the value has the correct name (that is, the name of the function) when the deletion is completed.

3.7.3 *Local Names*

Local names may be declared when names used in a function have values which should not be destroyed by a function call. Consider the following function which intersperses commas between the characters in its argument

```
COMMA ARG     *CHAR/"1"* = /F(RETURN)
      COMMA = COMMA CHAR "," /(COMMA)
```

The definition would be

```
DEFINE ("COMMA(ARG)","COMMA","CHAR")
```

so that the use of CHAR during the function call would not change the value of CHAR outside the function.

Local names are particularly important when recursively called functions use names for intermediate computation. Appendix II contains a program in which such use of local names is necessary.

IV. OPERATING ENVIRONMENT

The SNOBOL3 system consists of a compiler and an interpreter. The compiler translates SNOBOL3 source programs into an internal language suitable for the interpreter.

4.1 *Compilation*

4.1.1 *Source Program Listing*

During compilation, the source program is read and compiled card by card. Only columns 1 through 72 are read by the compiler. Consecutive statement numbers are added to the listing for reference.

4.1.2 *Comments*

A card with an asterisk in column 1 is treated as a comment. Comments are printed but otherwise ignored by the compiler. Comments may be used freely throughout the program and may be placed anywhere before the END card.

4.1.3 *Continue Cards*

A statement may be broken over card boundaries by use of the continue card convention. A period in column 1 is interpreted by the compiler as an indication that the card is a continuation of the preceding statement. Statements may be broken over card boundaries anywhere a blank is permissible in the syntax. Literals cannot be broken over card boundaries. A very long literal must be represented as a concatenation of shorter literals. For example,

```
      SYSPOT = "THE MAXIMUM LENGTH OF "
.     "THE COMPUTATIONAL THREAD HAS BEEN "
.     "COMPUTED TO BE"
```

There is no limit to the number of continue cards which may be used for a statement.

4.1.4 *Compiler Control Cards*

The programmer can perform some operations during the compilation process by means of compiler control cards. Compiler control cards are identified by a minus sign in column 1. The control action is taken when the card is encountered. Following the minus sign, the first nonblank subfield is taken to be the control word for the card. Other subfields, if any, are separated internally by commas. The control cards and actions are:

(*i*) — TITLE
Eject to a new page in the listing of the source program. Title subsequent pages with the information on the remainder of the control card.

(*ii*) — EJECT
Eject to a new page in the listing of the source program.

(*iii*) — SPACE
Print a blank line in the source program listing.

(*iv*) — UNLIST
Stop listing the source program. (The source program is normally listed.)

(*v*) — LIST
Resume listing.

(*vi*) — PCC
Print control cards. (Control cards are normally not printed.) PCC is a binary switch. Successive uses turn printing of control cards on and off.

(*vii*) — OPEN KEY,FILE
Open the file in the specified key area.

(*viii*) — REWIND FILE
Rewind the specified file.

(*ix*) — SOURCE FILE
Change the input source for the SNOBOL compiler to the specified file.

(*x*) — SYSTEM
Return control to the monitor under which SNOBOL3 operates.

(*xi*) — NULLOP OP
Make the control card operation OP inoperative for the rest of the program.

Invalid control cards, i.e., control cards not in the list above or with a format error, are printed but otherwise ignored. Control cards may be

used anywhere in the program before the END card, including between continue cards.

4.1.5 *Diagnostic Messages from the Compiler*

At the end of compilation one of three comments appears:

(*i*) SUCCESSFUL COMPILATION, indicating the source program contains no syntactic errors,
(*ii*) ERROR IN COMPILATION, indicating syntactic errors in the source program, or
(*iii*) FATAL ERROR ENCOUNTERED DURING COMPILATION, indicating the occurrence of an error of sufficient severity to terminate the compilation process.

Syntactic errors are printed following the source deck listing with statement numbers referring to each type of syntactic error. Compilation of a statement ceases when a syntactic error is encountered. Consequently subsequent errors in the same statement will not be detected. The syntactic error messages are

(*i*) ILLEGAL CONSTRUCTION, usually indicating an illegal character in a name, arithmetic operators not surrounded by blanks, or arithmetic operations in the pattern not enclosed in parentheses.
(*ii*) ERROR IN GROUPING, usually indicating unbalanced parenthesization, e.g., (A B))
(*iii*) TOO MANY ELEMENTS IN FUNCTION OR GROUPING, indicating overflow of an internal buffer due to an excessively complicated parenthetical grouping or function call. The maximum number of elements in such structures is an assembly parameter of the SNOBOL3 system and is about 50.
(*iv*) VARIABLE WITH GROUPING OR FUNCTION NOT CLOSED, indicating that a parenthetical grouping or function call in a string variable is not followed by a terminating asterisk, e.g., *T/SIZE(N)
(*v*) ERROR IN LENGTH SPECIFIER, indicating a syntactic error in the length of a fixed-length string variable, e.g., *F/"A"*
(*vi*) "NAMELESS" STRING REFERENCE IN ASSIGNMENT STATEMENT, indicating an attempt to assign a value to a literal, function call, or parenthetical grouping, e.g., "3" = "2"

(*vii*) ARITHMETIC OPERATION WITHOUT SECOND OPERAND, e.g., A = B + /(L1)

(*viii*) ARITHMETIC OPERATION WITHOUT FIRST OPERAND, e.g., A = + B

(*ix*) TWO ARITHMETIC OPERATIONS IN A ROW, e.g., A = B * * C

(*x*) NONBINARY ARITHMETIC OPERATION, e.g., A = B * C + D

(*xi*) PRIOR STATEMENT NOT PROPERLY TERMINATED, indicating a missing continue card or incomplete construction. This error is not detected until the following card is read and determined not to be a continue card. Consequently a statement such as A = B + not followed by a continue card will cause this error message. Compare with (*vii*) above.

(*xii*) ERROR IN GO-TO FIELD, e.g., /S(L1)(L2)

(*xiii*) "NAMELESS" STRING VARIABLE, e.g., *SIZE(SYSPOT)*

(*xiv*) ILLEGAL LABEL, indicating a label which does not start with a number or letter.

(*xv*) MULTIDEFINED LABEL, indicating the same label has occurred more than once.

(*xvi*) CONTINUE CARD NOT PRECEDED BY STATEMENT, indicating the first card in the source deck is a continue card. (Comment cards and compiler control cards may be freely interspersed between continue cards.)

If a fatal error occurs during compilation, the nature of the fatal error is printed followed by a listing of any syntactic errors. The fatal error messages are

(*i*) PROGRAM BUFFER OVERFLOW, indicating the source program is too large for an internal buffer. The size of this buffer is an assembly parameter.

(*ii*) ERROR READING INPUT TAPE, indicating a binary record was encountered during compilation. This condition is almost always due to the omission of an END statement.

(*iii*) END TRANSFER ADDRESS IN ERROR, indicating the label specified on the END card does not start with a number or letter.

(*iv*) END TRANSFER SPECIFIES UNDEFINED LABEL, indicating the label specified on the END card does not occur as a label in the program.

(*v*) MORE THAN 50 NONFATAL ERRORS, indicating the occurrence of more than 50 syntactic errors in the source program. This limit on syntactic errors is an assembly parameter. Such an excess of syntactic errors usually indicates the source deck is not a SNOBOL3 program.

(*vi*) SYSTEM ERROR, indicating a programming error in the SNOBOL3 compiler, or a machine error.

4.2 *Program Execution*

If a fatal error does not occur during compilation, execution is entered. Execution continues until the program transfers to or flows into the END statement or until an error occurs.

4.2.1 *Error Diagnostics*

The possible errors are:

(*i*) ATTEMPT TO EXECUTE STATEMENT WITH COMPILATION ERROR. Execution is terminated if an attempt is made to execute a statement with a compilation error. In the case of multidefined labels, the first occurrence is considered the valid label, and all transfers to this label go to the first occurrence. Subsequent statements with the same label are considered to be erroneous and flowing into such a statement terminates execution.

(*ii*) ATTEMPT TO TRANSFER TO AN UNDEFINED LABEL.

(*iii*) STRING OVERFLOWED 5461 CHARACTERS. This maximum length of strings is an implementation constraint.

(*iv*) INTERNAL BUFFER OVERFLOW, indicating an internal buffer has been exceeded, usually the result of excessive depth of recursive function calls or an excessively long pattern. The buffer sizes are assembly parameters.

(*v*) ATTEMPT TO USE NEGATIVE LENGTH IN A VARIABLE, indicating the length of a fixed-length variable is negative.

(*vi*) FUNCTION FAILED IN GO-TO FIELD, indicating a function call failed while evaluating a go-to field.

(*vii*) ATTEMPT TO ASSOCIATE A NULL NAME WITH I/O FILE.

(*viii*) ILLEGAL FILE OR FILE OPERATION, such as performing an input or output operation on a name not associated with a file or attempting to rewind the standard input file.

(*ix*) ATTEMPT TO READ PAST EOF ON SYSTEM INPUT TAPE. The first attempt to read an end-of-file on the standard input file results in failure of the statement in which the attempt occurred. A second attempt is fatal.

(*x*) IMPROPER ATTEMPT TO OPSYN A FUNCTION, indicating an attempt to OPSYN a name to an undefined function.

(*xi*) ATTEMPT TO CALL AN UNDEFINED FUNCTION.

(*xii*) IMPROPER DEFINITION OF A FUNCTION, indicating an error in a call of DEFINE.

(*xiii*) UNDEFINED OR NULL LABEL USED IN DEFINE STATEMENT, indicating the label specified in a call of DEFINE is null or does not occur in the program.

(*xiv*) TOO MANY ARGUMENTS IN A FUNCTION DEFINITION, indicating that the number of arguments and local variables in a defined function exceeds ten.

(*xv*) IMPROPER CALL OF A DEFINED FUNCTION, indicating too many arguments in the call of a defined function, or an improper argument for the CALL function.

(*xvi*) FUNCTION ENTERED OTHER THAN BY CALL, indicating an attempt to return from a defined function which has not been called.

(*xvii*) INDIRECT REFERENCE THROUGH THE NULL STRING, indicating an attempt to use the null string as a name.

(*xviii*) OUT OF SPACE, indicating available storage has been exhausted.

(*xix*) SYSTEM ERROR, indicating a programming error in the SNOBOL3 interpreter, or a machine error.

4.2.2 *Post-Mortem Information*

On termination of program execution, information is printed for the programmer's use.

If execution was terminated as the result of an error, the number of the statement in which the error occurred and the current level of function call are printed in addition to the error message.

In either normal or error termination, statistics concerning execution are provided. The number of statements executed and the number of times the scanner was entered for pattern matching are tabulated. Storage allocation statistics are provided, and total millisecond times in the compiler and interpreter are given.

4.3 *Debugging Aids*

Several functions are available specifically for debugging.

4.3.1 *Function Tracing*

Function calls may be traced by use of TRACE(FLIST) where the value of FLIST is a list of function names for which a trace is desired. For example, the call

TRACE("SIZE,F,EQUALS")

results in the subsequent tracing of the three functions given. Both primitive and defined functions can be traced.

When a defined function being traced is called, a message is printed on the listing indicating the level from which the call was made, the name of the function and the value of all its arguments at the time of the call. When the function returns, a message is printed indicating the level to which the return is made, the name of the function, and the value returned. If a failure return is made, this is also indicated but no value is given.

When a primitive function being traced is called, a message is printed on the listing indicating the level at which the call was made, the name of the function, and the value of its arguments. If the function call returned successfully, the value is given. Otherwise failure is indicated.

The tracing of functions may be stopped by calling

STOPTR(FLIST)

where the value of FLIST is a list of functions for which tracing is to be stopped.

4.3.2 *String Tracing*

A name may be traced during execution by calling the function STRACE(NAME,FILE) which associates the value of NAME with the value of FILE in the trace sense. For example,

STRACE("Y","OUT")

would cause Y to be associated with logical file OUT. Subsequently every time a value is assigned to Y, a trace message will be printed on the associated file indicating the name being traced, its new value and the statement number where the value was assigned. STRACE is essentially an input-output association function and behaves like the other

association functions. Consequently STRACE detaches NAME from any other input or output association. Similarly, if the second argument is null, association is made with the standard output file. String tracing may be terminated by detaching NAME.

4.3.3 *Diagnostics Resulting from Tracing*

Two nonfatal errors may occur as a result of tracing. Advisory diagnostic messages are printed for these cases.

(*i*) (name) HAS NOT BEEN DEFINED AND WILL NOT BE TRACED, indicating a request to trace an undefined function of the indicated name.

(*ii*) ** THE FOLLOWING TRACE OUTPUT HAS BEEN TRUNCATED, indicating that the printing of a string or function trace would exceed internal storage limitations, an assembly parameter set to about 600 characters. In this case the trace printout is truncated.

4.3.4 *String Dumps*

An alphabetical listing of all strings with nonnull values may be obtained on termination of execution. The MODE function is used to request this string dump.

MODE("DUMPERR") causes a string dump if execution is terminated by an error during execution. MODE("DUMP") causes a string dump following either normal or error termination. These calls of the MODE function must of course be made before execution is terminated.

V. ACKNOWLEDGMENT

The authors wish to express their appreciation to the many people who have contributed their ideas to the design of SNOBOL3. D. L. Clark, G. F. Faulhaber, G. K. Manacher, M. D. McIlroy, J. F. Poage, A. Simon, L. C. Varian, and D. Walters have been particularly helpful.

The assistance of G. K. Manacher and L. C. Varian in the implementation of the system is most gratefully acknowledged.

APPENDIX A

Catalog of Primitive Functions

Primitive functions may be divided into categories according to the nature of their operation and area of applicability. Individual functions

are described in the appropriate sections. For reference purposes a complete list of primitive functions follow.

A. Numerical Functions (Section 3.2)
   1. .EQ(X,Y)
   2. .NE(X,Y)
   3. .LE(X,Y)
   4. .LT(X,Y)
   5. .GE(X,Y)
   6. .GT(X,Y)
   7. .NUM(X)
   8. .REMDR(X,Y)

B. Diagnostic Functions (Section 4.3)
   1. TRACE(FLIST)
   2. STOPTR(FLIST)
   3. STRACE(NAME,FILE)

C. Input-Output and File Manipulation Functions (Section 3.5)
   1. READ(NAME,FILE)
   2. PRINT(NAME,FILE)
   3. PUNCH(NAME,FILE)
   4. EJECT(NAME)
   5. REWIND(NAME)
   6. BSREC(NAME)
   7. DETACH(NAME)
   8. OPEN(KEY,FILE)

D. System Mode Functions
   1. MODE(X)
      a. "ANCHOR" — Sections 2.8.1 and 3.3.3
      b. "UNANCHOR" — Section 3.3.3
      c. "INTEGER" — Section 3.2.3
      d. "TRUNCATION" — Section 3.2.3
      e. "SYSTEM" — Section 3.4.4
      f. "ENDJOB" — Section 3.4.4
      g. "DUMP" — Section 4.3.4
      h. "DUMPERR" — Section 4.3.4
   2. ANCHOR( ) — Section 3.3.3
   3. UNANCH( ) — Section 3.3.3

E. Functions Relating to Functions
   1. DEFINE(FORM,LABEL,NAMES) — Sections 2.8.2 and 3.7

2. OPSYN(NEW,OLD)	Section 3.6.2
3. CALL(FNC)	Section 3.6.2
F. Miscellaneous Functions	
1. EQUALS(X,Y)	Sections 2.8.1 and 3.6.3
2. UNEQL(X,Y)	Section 3.6.3
3. TRIM(S)	Section 3.6.3
4. TIME( )	Section 3.6.3
5. DATE( )	Section 3.6.3
6. SIZE(S)	Sections 2.8.1 and 3.6.3
7. SYSTEM(FILE)	Section 3.4.4

APPENDIX B

This appendix contains three sample programs. These programs are designed to illustrate various uses and features of SNOBOL3.

The first two programs involve symbolic evaluations which are inherently recursive. The third is an example of text manipulation.

All three examples use pattern matching in various forms. Statement 20 in the first example illustrates the use of back referencing to determine whether two lists have an element in common. The second program illustrates function tracing.

```
*       EXAMPLE 1.  THE WANG ALGØRITHM
*
*            THIS PRØGRAM IS THE ALGØRITHM BY HAØ WANG (CF. "TØWARD
*       MECHANICAL MATHEMATICS", IBM JØURNAL ØF RESEARCH AND
*       DEVELØPMENT 4(1) JAN.1960 PP.2-22.)  FØR A PRØØF-DECISIØN
*       PRØCEDURE FØR THE PRØPØSITIØNAL CALCULUS.  IT PRINTS ØUT A
*       PRØØF ØR DISPRØØF ACCØRDING AS A GIVEN FØRMULA IS A THEØREM
*       ØR NØT.  THE ALGØRITHM USES SEQUENTS WHICH CØNSIST ØF TWØ
*       LISTS ØF FØRMULAS SEPARATED BY AN ARRØW (--*).  INITIALLY, FØR
*       A GIVEN FØRMULA F THE SEQUENT
*
*                    --* F
*
*       IS FØRMED.  WANG HAS DEFINED RULES FØR SIMPLIFYING A FØRMULA
*       IN A SEQUENT BY REMØVING THE MAIN CØNNECTIVE AND THEN
*       GENERATING A NEW SEQUENT ØR SEQUENTS.  THERE IS A TERMINAL
*       TEST FØR A SEQUENT CØNSISTING ØF ØNLY ATØMIC FØRMULAS:
*
*            A SEQUENT CØNSISTING ØF ØNLY ATØMIC FØRMULAS IS VALID IF
*            THE TWØ LISTS ØF FØRMULAS HAVE A FØRMULA IN CØMMØN.
*
*       BY REPEATED APPLICATIØN ØF THE RULES ØNE IS LED TØ A SET ØF
*       SEQUENTS CØNSISTING ØF ATØMIC FØRMULAS.  IF EACH ØNE ØF THESE
*       SEQUENTS IS VALID THEN SØ IS THE ØRIGINAL FØRMULA.



        DEFINE("WANG(ANTE,CØNSEQ)" , "WANG","PHI,PSI")
*       READ IN THE EXPRESSIØN

READ    EXP         =  TRIM(SYSPIT)                 /F(END)
        SYSPØT      =
        SYSPØT      =  "FØRMULA: "  EXP
        SYSPØT      =
        WANG(," "  EXP)                             /F(INVALID)
        SYSPØT      =  "VALID"                       /(READ)
INVALID SYSPØT      =  "NØT VALID"                   /(READ)
WANG    SYSPØT      =  ANTE  " --* "  CØNSEQ
        ANTE        " NØT("  *(PHI)*  ")"  =        /S(ANØT)
        ANTE        " AND(" *(PHI)* "," *(PSI)* ")" =    /S(AAND)
        ANTE        " IMP(" *(PHI)* "," *(PSI)* ")" =    /S(AIMP)
        ANTE        " ØR(" *(PHI)* "," *(PSI)* ")" =     /S(AØR)
        ANTE        " EQU(" *(PHI)* "," *(PSI)* ")" =    /S(AEQU)

        CØNSEQ      " NØT(" *(PHI)* ")" =                /S(CNØT)
        CØNSEQ      " AND(" *(PHI)* "," *(PSI)* ")" =    /S(CAND)
        CØNSEQ      " IMP(" *(PHI)* "," *(PSI)* ")" =    /S(CIMP)
        CØNSEQ      " ØR(" *(PHI)* "," *(PSI)* ")" =     /S(CØR)
        CØNSEQ      " EQU(" *(PHI)* "," *(PSI)* ")" =    /S(CEQU)

        (ANTE " :" CØNSEQ " ")   " " *(P)* " " ** *:" ** " " P " "
                                                     /S(RETURN)F(FRETURN)

ANØT    WANG(ANTE,CØNSEQ  " "  PHI)                  /S(RETURN)F(FRETURN)

AAND    WANG(ANTE  " "  PHI  " "  PSI, CØNSEQ)      /S(RETURN)F(FRETURN)

AØR     WANG(ANTE  " "  PHI,CØNSEQ)                  /F(FRETURN)
        WANG(ANTE  " " PSI,CØNSEQ)                   /S(RETURN)F(FRETURN)

AIMP    WANG(ANTE  " "  PSI,CØNSEQ)                  /F(FRETURN)
        WANG(ANTE,CØNSEQ  " " PHI)                   /S(RETURN)F(FRETURN)

AEQU    WANG(ANTE  " " PHI  " "  PSI,CØNSEQ)        /F(FRETURN)
        WANG(ANTE,CØNSEQ  " " PHI  " "  PSI)        /S(RETURN)F(FRETURN)

CNØT    WANG(ANTE  " "  PHI,CØNSEQ)                  /S(RETURN)F(FRETURN)

CAND    WANG(ANTE,CØNSEQ  " " PHI)                   /F(FRETURN)
        WANG(ANTE,CØNSEQ  " "  PSI)                  /S(RETURN)F(FRETURN)

CØR     WANG(ANTE,CØNSEQ  " " PHI  " "  PSI)        /S(RETURN)F(FRETURN)

CIMP    WANG(ANTE  " "  PHI,CØNSEQ  " "  PSI)       /S(RETURN)F(FRETURN)

CEQU    WANG(ANTE  " "  PHI,CØNSEQ  " "  PSI)       /F(FRETURN)
        WANG(ANTE  " "  PSI,CØNSEQ  " "  PHI)       /S(RETURN)F(FRETURN)
END

SUCCESSFUL CØMPILATIØN
```

```
FØRMULA: IMP(NØT(ØR(P,Q)),NØT(P))

 --*  IMP(NØT(ØR(P,Q)),NØT(P))
 NØT(ØR(P,Q)) --*  NØT(P)
 --*  NØT(P) ØR(P,Q)
 P --*  ØR(P,Q)
 P --*  P Q
VALID

FØRMULA: IMP(AND(NØT(P),NØT(Q)),EQU(P,Q))

 --*  IMP(AND(NØT(P),NØT(Q)),EQU(P,Q))
 AND(NØT(P),NØT(Q)) --*  EQU(P,Q)
 NØT(P) NØT(Q) --*  EQU(P,Q)
 NØT(Q) --*  EQU(P,Q) P
 --*  EQU(P,Q) P Q
 P --*  P Q Q
 Q --*  P Q P
VALID

FØRMULA: IMP(IMP(ØR(P,Q),ØR(P,R)),ØR(P,IMP(Q,R)))

 --*  IMP(IMP(ØR(P,Q),ØR(P,R)),ØR(P,IMP(Q,R)))
 IMP(ØR(P,Q),ØR(P,R)) --*  ØR(P,IMP(Q,R))
 ØR(P,R) --*  ØR(P,IMP(Q,R))
 P --*  ØR(P,IMP(Q,R))
 P --*  P IMP(Q,R)
 P Q --*  P R
 R --*  ØR(P,IMP(Q,R))
 R --*  P IMP(Q,R)
 R Q --*  P R
 --*  ØR(P,IMP(Q,R)) ØR(P,Q)
 --*  ØR(P,Q) P IMP(Q,R)
 Q --*  ØR(P,Q) P R
 Q --*  P R P Q
VALID
```

```
 NØRMAL EXIT FRØM SNØBØL AT LEVEL        0

SNØBØL RUN STATISTICS , NØ. ØF RULES EXECUTED =    254 NØ ØF SCANNER ENTRIES =    187

STØRAGE ALLØCATIØN STATISTICS  --      171  STRINGS STØRED      397  WØRDS FØR STØRED STRINGS
        600 REFERENCE ASSIGNMENT WØRDS ,       0 REFERENCE,        0   GARBAGE, AND
        0 HYPER-GARBAGE CØLLECTIØN(S)

ELAPSED TIMES-CØMPILER     620 , INTERPRETER        735 IN MS
```

```
*       EXAMPLE 2.  DIFFERENTIATIØN ØF ALGEBRAIC EXPRESSIØNS
*
*            THIS PRØGRAM DIFFERENTIATES A FULLY PARENTHESIZED
*       ALGEBRAIC EXPRESSIØN WITH RESPECT TØ X.  THE EXPØNENTIATIØN
*       ØPERATØR IS INDICATED BY A DØLLAR SIGN.

        DEFINE("D(D)","DO",,"U,V,N")                                        1
        DEFINE("SIMPLIFY(EXP)","SO",,"U")                                   2
        TRACE("D,SIMPLIFY")                                                 3

*       READ IN THE EXPRESSIØN.

READ    EXP     =  TRIM(SYSPIT)                         /F(END)             4

        MØDE("ANCHØR")                                                      5
        SYSPØT  =                                                           6
        SYSPØT  =  "THE DERIVATIVE ØF " EXP " IS " SIMPLIFY(D(EXP))         7
.                                                       /(END)              7

*       THE FUNCTIØN "D"

DO      D       "(" *(U)* "+" *(V)* ")" = "(" D(U) "+" D(V) ")"             8
.                                                       /S(RETURN)          8

        D       "(" *(U)* "-" *(V)* ")" = "(" D(U) "-" D(V) ")"             9
.                                                       /S(RETURN)          9

        D       "(" *(U)* "*" *(V)* ")" = "((" U "*" D(V) ")+("             10
.                       V  "*" D(U) "))"                /S(RETURN)          10

        D       "(" *(U)* "/" *(V)* ")" = "(((" V "*" D(U) ")-("            11
.                         U "*" D(V) "))/(" V "$2))"    /S(RETURN)          11

        D       "(" *(U)* "$" *(N)* ")" =                                   12
.                     "((" N "*(" U "$" N - "1" "))*" D(U) ")"              12
.                                                       /S(RETURN)          12

        D       =  EQUALS(D,"X")  "1"                   /S(RETURN)          13
        D       =  "0"                                  /(RETURN)           14

*       THE FUNCTIØN "SIMPLIFY"

SO      MØDE("UNANCH")                                                      15
S1      EXP     "(" *(U)* "*0)"   =  "0"                /S(S1)              16
        EXP     "(0*" *(U)* ")"   =  "0"                /S(S1)              17
S2      EXP     "(" *(U)* "$1)"   =  U                  /S(S2)              18
S3      EXP     "(" *(U)* "*1)"   =  U                  /S(S3)              19
S4      EXP     "(1*" *(U)* ")"   =  U                  /S(S4)              20
S5      EXP     "(0+" *(U)* ")"   =  U                  /S(S5)              21
S6      EXP     "(" *(U)* "+0)"   =  U                  /S(S6)              22
S7      EXP     "(" *(U)* "-0)"   =  U                  /S(S7)              23
        SIMPLIFY  =  EXP                                /(RETURN)           24
END                                                                         25
```

```
SUCCESSFUL CØMPILATIØN
```

```
0 LEVEL CALL ØF D("(((A*(X$2))+(B*X))+C)")

1 LEVEL CALL ØF D("((A*(X$2))+(B*X))")

2 LEVEL CALL ØF D("(A*(X$2))")

3 LEVEL CALL ØF D("(X$2)")

4 LEVEL CALL ØF D("X")

4 LEVEL RETURN ØF D = "1"

3 LEVEL RETURN ØF D = "((2*(X$1))*1)"
```

```
3 LEVEL CALL ØF D("A")

3 LEVEL RETURN ØF D = "0"

2 LEVEL RETURN ØF D = "((A*((2*(X$1))*1))+((X$2)*0))"

2 LEVEL CALL ØF D("(B*X)")

3 LEVEL CALL ØF D("X")

3 LEVEL RETURN ØF D = "1"

3 LEVEL CALL ØF D("B")

3 LEVEL RETURN ØF D = "0"

2 LEVEL RETURN ØF D = "((B*1)+(X*0))"

1 LEVEL RETURN ØF D = "(((A*((2*(X$1))*1))+((X$2)*0))+((B*1)+(X*0)))"

1 LEVEL CALL ØF D("C")

1 LEVEL RETURN ØF D = "0"

0 LEVEL RETURN ØF D = "((((A*((2*(X$1))*1)))+((X$2)*0))+((B*1)+(X*0)))+0)"

0 LEVEL CALL ØF SIMPLIFY("((((A*((2*(X$1))*1))+((X$2)*0))+((B*1)+(X*0)))+0)")

0 LEVEL RETURN ØF SIMPLIFY = "((A*(2*X))+B)"
 THE DERIVATIVE ØF (((A*(X$2))+(B*X))+C) IS ((A*(2*X))+B)

-------------------------------------------------------------------------------

 NØRMAL EXIT FRØM SNØBØL AT LEVEL        0
SNØBØL RUN STATISTICS , NØ. ØF RULES EXECUTED =     71 NØ ØF SCANNER ENTRIES =     54
STØRAGE ALLØCATIØN STATISTICS  --      125  STRINGS STØRED      216  WØRDS FØR STØRED STRINGS
        450 REFERENCE ASSIGNMENT WØRDS ,       0 REFERENCE,       0   GARBAGE, AND
        0 HYPER-GARBAGE CØLLECTIØN(S)
ELAPSED TIMES-CØMPILER     474 , INTERPRETER        380 IN MS
```

```
*       EXAMPLE 3.  EDITØR
*
*            THIS PRØGRAM READS IN TEXT AND PRINTS N CHARACTERS
*       PER LINE.  EXTRA BLANKS ARE INSERTED BETWEEN WØRDS
*       TØ FILL ØUT LINES.

        MØDE("ANCHØR")                                                      1
        DEFINE("INSERT(K,LINE)","IN","BLANK,WØRD")                          2

*       READ IN N
        TRIM(SYSPIT)          *N*                                           3

READ    TEXT   =  TEXT TRIM(SYSPIT) " "              /S(READ)               4
        SYSPØT    =                                                         5
        SYSPØT    =  "PRINT THE FØLLØWING STRING " N                        6
.                          " CHARACTERS PER LINE:"                          6
        SYSPØT    =                                                         7

*       PRINT THE INPUT TEXT

        TEXT1   =   TEXT                                                    8
PRINT   TEXT1   *SYSPØT/"90"*   =            /S(PRINT)                      9
        SYSPØT    =   TEXT1                                                10
        SYSPØT    =                                                        11
        SYSPØT    =                                                        12
TEST    .GT(SIZE(TEXT),N)                    /F(LASTLINE)                  13
        K         =    "0"                                                 14
SCAN    TEXT      *LINE/(N - K)*  " "  =     /F(BUMPK)                     15
        SYSPØT    =  INSERT(K,LINE)          /(TEST)                       16
BUMPK   K         =   .LT(K,N)  K + "1"      /S(SCAN)F(ERR)                17

*   FUNCTIØN TØ INTERSPERSE K BLANKS IN A LINE

IN      INSERT    =  .EQ(K,"0")  LINE        /S(RETURN)                    18
        LINE      **  " "                    /S(BLINK)                     19
        INSERT    =  LINE                    /(RETURN)                     20
BLINK   BLANK     =  BLANK  " "                                            21
LØØP    LINE    *WØRD*  BLANK   =            /F(MØRE)                      22
        INSERT    =   INSERT  WØRD BLANK  " "                              23
        K         =    .GT(K,"1")   K - "1"  /S(LØØP)                      24
        INSERT    =  INSERT LINE             /(RETURN)                     25
MØRE    LINE      =  INSERT LINE                                           26
        INSERT    =                          /(BLINK)                      27

LASTLINE  SYSPØT  =  TEXT                    /(END)                        28
ERR     SYSPØT    =                                                        29
        SYSPØT  = "** EDITØR CANNØT PRINT LINE BECAUSE N IS TØØ SMALL."     30
END                                                                         31
```

```
SUCCESSFUL CØMPILATIØN
```

```
PRINT THE FØLLØWING STRING 75 CHARACTERS PER LINE:

TWØ FUNCTIØNS HAVE BEEN ADDED TØ INCREASE THE FLEXIBILITY ØF DEALING WITH THE SYSTEM INPUT
 SØURCE. GETSRC RETURNS AS VALUE THE CURRENT SYSTEM SØURCE. SETSRC SETS THE CURRENT SYSTEM
 SØURCE TØ THE VALUE ØF FILE. A NULL VALUE IS RETURNED. THREE I/Ø FUNCTIØNS HAVE BEEN ADDE
D WHICH TAKE FILES AS ARGUMENTS. THESE FUNCTIØNS CØMPLEMENT THE CØRRESPØNDING SNØBØL3 FUNC
TIØNS WHICH REQUIRE NAMES ASSØCIATED WITH FILES AS ARGUMENTS.


TWØ  FUNCTIØNS  HAVE BEEN ADDED TØ INCREASE THE FLEXIBILITY ØF DEALING WITH
THE SYSTEM INPUT SØURCE. GETSRC RETURNS AS VALUE THE CURRENT SYSTEM SØURCE.
SETSRC SETS THE CURRENT SYSTEM SØURCE TØ THE VALUE ØF FILE. A NULL VALUE IS
RETURNED.  THREE  I/Ø  FUNCTIØNS  HAVE  BEEN  ADDED  WHICH  TAKE  FILES  AS
ARGUMENTS.  THESE  FUNCTIØNS CØMPLEMENT THE CØRRESPØNDING SNØBØL3 FUNCTIØNS
WHICH REQUIRE NAMES ASSØCIATED WITH FILES AS ARGUMENTS.
```

```
 NØRMAL EXIT FRØM SNØBØL AT LEVEL        0

SNØBØL RUN STATISTICS , NØ. ØF RULES EXECUTED =    130 NØ ØF SCANNER ENTRIES =     42

STØRAGE ALLØCATIØN STATISTICS  --      194  STRINGS STØRED     1296  WØRDS FØR STØRED STRINGS
        600 REFERENCE ASSIGNMENT WØRDS ,       0 REFERENCE,       0   GARBAGE, AND
        0 HYPER-GARBAGE CØLLECTIØN(S)

ELAPSED TIMES-CØMPILER     .410 , INTERPRETER        547 IN MS
```

REFERENCES

1. Newell, A., Ed., *Information Processing Language-V Manual,* Prentice-Hall, 1961.
2. McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Comm., *ACM 3* April, 1960, p. 184.
3. An Introduction to COMIT Programming, The Research Lab of Electronics and the Computation Center, M.I.T., 1961.
4. Farber, D. J., Griswold, R. E., and Polonsky, I. P., SNOBOL, a String Manipulation Language, J. ACM, *11,* No. 1, 1964.
5. Manacher, G. K., A Package of Subroutines for the SNOBOL Language. (Unpublished)
6. Griswold, R. E. and Polonsky, I. P., Tree Functions for SNOBOL3. (Unpublished)
7. Griswold, R. E., Linked-List Functions for SNOBOL3. (Unpublished)
8. Farber, D. J., Griswold, R. E., Manacher, G. K., Polonsky, I. P., and Varian, L. C., Programming Machine-Language Functions for SNOBOL3. (Unpublished)