Check for updates

SNOBOL, A String Manipulation Language*

D. J. Farber, R. E. Griswold, and I. P. Polonsky

Bell Telephone Laboratories, Inc., Holmdel, New Jersey

Abstract. SNOBOL is a programming language for the manipulation of strings of symbols. A statement in the SNOBOL language consists of a rule that operates on symbolically named strings. The basic operations are string formation, pattern matching and replacement. Facilities for integer arithmetic, indirect referencing, and input-output are included. In the design of the language, emphasis has been placed on a format that is simple and intuitive. SNOBOL has been implemented for the IBM 7090.

1. *Introduction*

The ability to manipulate symbolic rather than numeric data is becoming increasingly important in programming. As symbolic manipulations become more complex, programming in machine-oriented languages becomes increasingly tedious and cumbersome. A number of programming languages [1–4] have been developed to aid the programmer in such problems. As interest in language translation, program compilation and combinatorial problems has increased, many of these languages have been used for types of problems for which they were never intended. It is clear that more general symbol manipulation languages will materially expand the class of problems that can be programmed with reasonable time and effort.

The string-oriented symbolic language Snobol has been developed with these problems in mind. The choice of the string of symbols as the basic data structure in Snobol was made because most symbol manipulation problems of current interest may be naturally described in terms of string manipulations. Unfortunately, no standard notation or accepted system of operations exists for string manipulations. Three basic operations seem essential, however: (i) creation of strings, (ii) examination of the contents of strings, and (iii) alteration of strings depending on their contents.

A system for accomplishing these basic operations forms the nucleus of Snobol. In constructing the syntax and selecting the notation for Snobol, the potential programmer was given careful consideration. Emphasis has been placed on simplicity and intuitiveness while maintaining so far as possible the inherent power of a high-level programming language.

In developing Snobol the authors have been guided by experience with other languages, principally Comit [1] and SCL [4].

2. *Basic Concepts*

2.1. Strings and String Names. The basic data structure in Snobol is a string of symbols. Names are assigned to strings to provide an easy way of re-

* Received October, 1963.

ferring to particular strings. The name of a string may be any string of numerals and/or letters with medial periods permitted. Thus the string with name LINE.1 may have the contents[1]

AROUND, AROUND THE SUN WE GO:

2.2 STRING FORMATION. The most elementary type of string manipulation is the formation of strings. A string named LINE.1 with the contents given above is formed by the following rule

LINE.1 = "AROUND, AROUND THE SUN WE GO:"

The pair of quotation marks specifies the literal contents of a string. Any symbols (except quotation marks) can be placed within the quotation marks. Strings can also be formed by concatenation. Thus the rule

LINE.1 = "AROUND, AROUND" "THE SUN WE GO:"

produces the same result as the preceding example.

Strings which have been named previously can be used to form new strings. For example, the rule EXAMPLE = LINE.1 forms a string named EXAMPLE with the same contents as the string named LINE.1.

Both literals and named strings can be used in formation. The sequence of rules

LINE.1 = "AROUND, AROUND THE SUN WE GO:"

LINE.2 = "THE MOON GOES ROUND THE EARTH."

LINE.3 = "WE DO NOT DIE OF DEATH:"

LINE.4 = "WE DIE OF VERTIGO."

TEXT = LINE.1 "/" LINE.2 "/" LINE.3 "/" LINE.4

will form a composite string with slashes separating the lines in the conventional manner. Note that the spaces between string names and literals serve as break characters for distinguishing the elements to be concatenated. At least one space is required for separation, but more may be inserted.

In forming a string, the string itself may be used. Hence, after performing the two rules

NUMBER = "1"

NUMBER = NUMBER NUMBER "0"

the string NUMBER will contain the literals "110".

[1] This and the next few examples are taken from Archibald MacLeish, "Mother Goose's Garland," *Collected Poems, 1917–1952*, Houghton Mifflin Co., Boston, Mass. Quoted by permission of the publishers.

2.3 PATTERN MATCHING. The process of examining the contents of a string for a given substring is called *pattern matching*. For example, to determine whether the string named LINE.1 contains the literals "ROUND", the following rule would suffice:

```
LINE.1  "ROUND"
```

This rule is similar to a formation rule, but without the equal sign. The string LINE.1 is scanned from the left for an occurrence of the five literals "ROUND" in succession. A pattern matching rule may succeed or fail. Section 3 describes how this success or failure may be recognized and used. If LINE.1 is formed as above, the scan would be successful. The string being scanned is not altered in any way.

The pattern may be specified by the concatenation of a number of literals and string names just as the contents of a string to be formed were specified. For example,

```
TEXT  LINE.1  "/"  LINE.2
```

specifies a scan of the string named TEXT for an occurrence of the contents of the string LINE.1 immediately followed by the literal "/" and in turn immediately followed by the contents of the string LINE.2.

2.4 STRING VARIABLES. The type of scanning described in the Section 2.3 is clearly limited. One might, for example, want to know whether a string contains one substring followed by another, but with the second substring not necessarily immediately after the first. A string variable is introduced to permit this kind of scanning. The rule

```
LINE.1  "AROUND"  *FILLER*  "SUN"
```

is of this kind. Here we wish to know whether LINE.1 contains "AROUND" followed by "SUN" with perhaps something between. The symbols *FILLER* represent a string variable which takes care of this "something." If LINE.1 is formed as in Section 2.2, this scan would be successful. A string variable may be any string name bounded by asterisks.

A byproduct of successfully matching a pattern containing a string variable is the formation of a new string which has the name given between the asterisks of the string variable. This newly formed string contains a copy of the substring of the scanned string where the string variable fitted, i.e. the "something" previously mentioned. In the example given, a string named FILLER would be formed with the literal contents ", AROUND THE". This newly formed string is entirely independent of the scanned string.

2.5 REPLACEMENT. One final rule permitting alteration of the contents of a string will complete the basic string manipulations. Suppose in the string LINE.2 we wished to replace "EARTH" by "GLOBE". The following rule will accomplish this

```
LINE.2  "EARTH" = "GLOBE"
```

This rule scans LINE.2 for an occurrence of "EARTH". If this scan is successful, "EARTH" is then replaced by "GLOBE". Thus LINE.2 would become "THE MOON GOES ROUND THE GLOBE.". If the scan fails, the string being scanned is not altered.

As before, the pattern may be any combination of named strings, literals, and string variables. Only the substring matching the pattern is replaced. As a case of special interest, writing nothing to the right of the equal sign causes the substring found by the scan to be deleted. Thus LINE.2 "EARTH" = would delete "EARTH" from LINE.2.

Any string formed as the result of a successful pattern match of a string variable on the left side of the equal sign can be used in the replacement on the right side. Thus

LINE.1 "AROUND" *FILLER* "SUN" = FILLER

would result in the deletion of "AROUND" and "SUN" from LINE.1.

2.6 Back Referencing. In the example above, the string formed as the result of a string variable in a successful pattern match was used for replacement in the same rule. It is even possible to use strings tentatively matched by string variables in the course of the scan. Thus a pattern may contain a string name which is the same as the name of a string variable used previously in the pattern. For example, *X* M X is a pattern containing such back referencing. Since the scan proceeds from left to right, an attempt to find an occurrence of X will only be made after X is tentatively defined by *X*. If

TEXT = "(C,D)(A,B)(D,C)(A,B)"

then the rule

TEXT "(" *X* ")" *Y* "(" X ")"

would succeed, forming a string named X with the contents "A,B".

2.7 Other Types of String Variables. The string variable described in Section 2.4 was completely arbitrary in the sense that it could match any substring depending on the particular pattern and string being scanned. However, it is often desirable to restrict the types of substrings a string variable can match. For this purpose, there are two other types of string variables.

2.7.1 *Balanced String Variables.* Balanced string variables are useful for analyzing algebraic structures. A balanced string variable can only match a nonvoid substring which is balanced with respect to parentheses. Some examples of balanced substrings are:

A A+(BC) (((,A,B)ACD))

The following substrings are not balanced:

)((((A+B))+C))

To indicate a balanced string variable, the string name is bounded by parentheses and then by asterisks, e.g. *(CATCH)*.

2.7.2 *Fixed-Length String Variables.* A fixed-length string variable can only match a substring of specified length. A fixed-length string variable is indicated by appending to the string name a slash and the length. The length may be expressed either by a literal integer or the name of a string containing an integer. Thus *PAD/"3"* is a fixed-length string variable which can only match a substring of three characters. Similarly, MATCH/N* where N = "15" can only match a substring of 15 characters.

3. *Program Structure*

In order to make use of the string manipulation facilities of SNOBOL, the rules are assembled into a program consisting of a number of statements which are executed in a prescribed order.

3.1 STATEMENT FORMAT. A statement, in general, consists of three parts, separated by blanks, in the following order:

(i) a *label*, naming the statement;

(ii) a *rule*, which may be one of the types described in Section 2; and

(iii) a *go-to*, which may conditionally specify which labeled statement is to be executed next.

3.1.1 *Labels.* A label may be any permissible string name, and must start at the beginning of the statement. The label on a statement is optional. If a statement has no label, it must begin with a blank. A line beginning with an asterisk is a comment and is not executed.

3.1.2 *Rules.* Various types of rules were described in Section 2. In all of these types, a rule may be considered to consist of four parts, separated by blanks, in the following order:

(i) a string to be manipulated, called the *string reference*;

(ii) a *left side*, specifying a pattern;

(iii) an *equal sign*; and

(iv) a *right side*, specifying a replacement.

The string reference is mandatory. Any of the rest of the rule parts may be absent, depending on the particular rule.

3.1.3 *Go-to.* The go-to consists of a slash followed by one or more of the following parts:

(i) *An unconditional transfer*, which has the form (BA), specifying that upon completion of the statement, the next statement to be executed is the statement with label BA.

(ii) *A conditional transfer on failure*, which has the form F(BB), specifying that if the statement fails, the statement with label BB is to be executed next.

(iii) *A conditional transfer on success*, which has the form S(BC), similar to failure transfer but with transfer to BC made on success.

Some examples of go-to's are:

/(MORGAN) /F(TIME) /S(ARBOR)F(RESET)

3.2 PROGRAM FORMAT AND EXECUTION. A program consists of a sequence of statements followed by a statement with the label END and a string reference which is the label of the first program statement to be executed.

Statements are executed in succession unless a go-to specifies a transfer to some other statement in the program. In all situations where a go-to is not specified, control is transferred to the next statement in the program. The program execution terminates when a transfer to END is made.

As an example, consider the following simple program to remove all occurrences of the letters A, E, I, O and U from a string named TEXT (presumed to be already defined):

```
START   VOWEL = "A,E,I,O,U,"
V1      VOWEL *V* "," =          /F(END)
V2      TEXT V =                 /S(V2)F(V1)
END     START
```

The program execution begins with the statement labeled START, consequently forming a string named VOWEL. The next statement executed is V1 which names the first vowel in VOWEL to be V, and deletes this vowel and the comma following it. This rule will not fail the first time it is executed; hence control is transferred to the subsequent rule V2.

V2 looks in TEXT for the vowel and if successful deletes it, transferring control to V2 once more. This loop continues until all occurrences of the vowel have been removed. When V2 finally fails, control is transferred to V1 which selects another vowel from VOWEL, and so on. When VOWEL is exhausted, the program is terminated by transferring to END.

4. *Arithmetic*

Simple arithmetic may be performed on strings whose contents are integers. Binary operations of addition, subtraction, multiplication, division and exponentiation may be performed on the right side of any rule. The symbols for these operations are the operators

$$+ \quad - \quad * \quad / \quad .$$

respectively. For example L = A + B would form a string named L containing the arithmetic sum of the contents of strings A and B.

This arithmetic expression can be considered as a single element on the right side, and may occur in place of any right side element. For example, suppose a string has two indices, such as "L.1.3". We may increment these indices by using the arithmetic operation. Suppose the name of "L.1.3" is MARKER. The rule

```
MARKER  "L." *I* "." *J* = "L." I + "1" "." J + "1"
```

would increase both indices, so that MARKER would contain "L.2.4".

Any number of these binary operations can be performed, but more complicated expressions such as A + B + C and A + (B * C) are not permitted.

5. *Indirectness*

It is frequently convenient, and for many purposes necessary, to be able to introduce a level of indirectness. This is accomplished in SNOBOL by writing $ in front of the string name. Thus if the string FACTOR contains the literals "TERM", writing $FACTOR is the same as writing TERM.

An example of the utility of such a feature is the ability of altering the effective go-to of a rule. Suppose I and J are strings containing numbers generated in the program. The rule

```
LABEL = "B" I "." J          /($LABEL)
```

first creates a string with literal contents depending on I and J. Suppose I is "3" and J is "2". Then LABEL would be "B3.2". The go-to then transfers to the rule labeled B3.2. Thus indirectness here permits alteration of program flow depending on data (here I and J).

The indirect feature is useful for specifying the return address of a subroutine. Suppose CAP is the label of the first rule of a subroutine and /($RET) is the go-to of the last rule executed in CAP. A call to the subroutine which returns to the rule with label A5 is given by the following rule:

```
RET = "A5"          /(CAP)
```

6. *Input-Output*

Input and output are accomplished by use of the two commands .READ and .PRINT following the string reference SYS.

In the case of .READ, any pattern may appear following the .READ command. The statement causes a string to be read from the input device and scanned for the pattern. For example,

```
S1  SYS  .READ  *DATA*  " "
```

reads in a string and names everything up to the first blank DATA. A statement containing a .READ command will fail if the input device contains no more data or if the pattern match fails.

In the case of .PRINT, any right side may appear following the .PRINT command. The statement causes the string specified by the right side to be written on the output device. For example,

```
S2  SYS  .PRINT  "L="  L
```

would print out the literals L= followed by the contents of the string named L.

7. *Scanning Algorithm*

In general, a pattern specified on the left side of a rule consists of a number of elements, i.e. named strings, literals or string variables. Examples in the

preceding sections have described the substrings which each type of element can match. The way that a specified pattern matches a given string is usually clear. In cases where questions may arise, the following scanning algorithm, which describes the details of the pattern matching process, may be useful.

Rule 1. An attempt is made to match the first pattern element starting at the first symbol of the string. If this match cannot be made, the match is attempted starting at the next symbol of the string, and so on.

Rule 2. The matching process proceeds from left to right, successively matching pattern elements. Each pattern element matches the shortest possible substring.

Rule 3. If at some point an element cannot match a substring, an attempt is made to obtain a new match for the preceding pattern element. This new match is accomplished by extending the substring formerly matched to obtain the next shortest acceptable value. If this extension cannot be made, Rule 3 is applied again. If there is no preceding element a new match is attempted according to Rule 1.

Rule 4. If the last pattern element is an arbitrary string variable (i.e. not fixed-length or balanced), its matching substring is extended to the end of the string.

The pattern match succeeds when the last pattern element has been matched. The pattern match fails when the first element cannot be matched.

8. *Conclusion*

SNOBOL has been implemented for the 7090 by the authors and Miss L. P. White. During the past eight months this system has been used in a wide variety of symbol manipulation problems. The most significant application has been in the areas of program compilation and generation of symbolic equations.

Several additional features are being planned for inclusion in SNOBOL in the near future. One of these is string functions which will facilitate programming of functional subroutines. Extended input-output facilities which are consistent with the string orientation and rule structure of the language will also be added.

Acknowledgment. The authors wish to thank C. Y. Lee for his support and encouragement. During the development of SNOBOL, the authors have benefited from discussion with many people. The enthusiasm and numerous suggestions of M. D. McIlroy have been particularly helpful. In addition, his string macro package [5] greatly facilitated the implementation.

APPENDIX

Example of a SNOBOL Program

The problem of alphabetizing a list of words using a radix sort illustrates the use of many of the features of SNOBOL. The program shows the format of the implementation.

In this procedure, 26 bins corresponding to the letters of the alphabet are

```
NØRMAL EXIT FRØM SNØBØL, REQUESTED DUMP FØLLØWS

RUN STATISTICS:PRØGRAM SIZE =    94,EXECUTIØN CYCLES=    1336,SCANNER=    652

STØRAGE ALLØCATIØN STATISTICS.    473 STRINGS STØRED,    3497 WØRDS FØR STØRED STRINGS,    1800
WØRDS  ASSIGNED FØR REFERENCES,    0 REF. CØLLECTIØNS,    0 GARBAGE SØRTS.


DUMPS FØR            JØB    5088,.005   R E GRISWØLD   X5653

    2073 CØREPM      0,02060,02107

AC 00 +002100077103  +.14721990-38  MQ +000000000000  +.00000000-39  SI 000000000000  EK 000000000000  SW 00  SL 00  ØVF 1  TM 0
IR1 00136  00094   IR2 77776  32766                    IR4 75706  31686

02060    077777002074   IXA   4070,4   PXA   0,4    XCA           STR             LDQ   10014   STR             LDQ   10013
02070    STR            TSX   77025,4  TSX   L07,4  STR           740301300060    516445606263    216331626331    236272475146
02100    275121446062   317125601360   450673011030   732567252364   633146456023   702343256213   451073113073   622321454525

ALL DUMPS CØMPLETED.

DATE  TIMES: ØN      ØFF       ELAPSED  SETUP  0  CHARGED  1401    SYSTEM  2  FØRTRAN  0  FAP  0  RUN  1  DUMPS  PIT  PØT  CØDE
10/11 5088  16.8353  16.8381   0.0028   0.         0.0028   0.0025  0.0005     0.            0.        0.0022  0.0001 0421 1393 H38
PRINTED ØN 1401 SYSTEM 2
```

```
*  ALPHABETIZATIØN USING A RADIX SØRT TECHNIQUE                         1
*                                                                       1
*                                                                       1
*  FIRST THE SIZE ØF THE LØNGEST STRING, AND THEN THE LIST ØF WØRDS IS  1
*  READ INTØ STRINGS ØF CØRRESPØNDING NAMES.  AFTER PRINTING THE LIST   1
*  THE WØRDS ØN "LIST" ARE EXAMINED USING THE FIXED-LENGTH STRING       1
*  VARIABLE FEATURE.  IF THE WØRD IS TØØ SHØRT, THE WØRD IN ADDED TØ THE 1
*  SPECIAL BIN ( NAMED "BIN").  ØTHERWISE THE LETTER CØNTAINED IN "PIT" 1
*  IS THE NAME ØF THE BIN INTØ WHICH THE WØRD IS FILED USING THE        1
*  INDIRECT FEATURE.  AFTER ALL WØRDS HAVE BEEN FILED, THE LIST IS      1
*  REASSEMBLED AT STATEMENT L5 AND FØLLØWING STATEMENTS.  NØTE THAT L5  1
*  PLACES THE CØNTENTS ØF "BIN" IN "LIST" AND AT THE SAME TIME VØIDS    1
*  "BIN" FØR THE NEXT PASS.  NEXT EACH ØF THE BINS IS ADDED TØ "LIST"   1
*  IN ALPHABETIC ØRDER, AND THEN VØIDED.  THE NEXT PASS IS THEN MADE.   1
*  WHEN "SIZE" BECØMES NEGATIVE, THE LAST PASS HAS BEEN MADE AND THE    1
*  ALPHABETIZED LIST IS PRINTED ØUT                                     1
*                                                                       1
BEGIN   SYS       .READ   *SIZE*   " "                                  1
START   SYS       .READ   *WØRDS*  " "               /F(L0)             6
        LIST      =  LIST  WØRDS                     /(START)           12
L0      SYS       .PRINT  "THE LIST TØ BE ALPHABETIZED IS :   "  LIST   18
        SYS       .PRINT  VØID                                          23
L1      ALPHABET  =  "ABCDEFGHIJKLMNØPQRSTUVWXYZ"                       27
L2      SIZE      =  SIZE  -  "1"                                       31
        SIZE      "-"                                /S(FIN)            37
L3      LIST      *WØRD*  ","  =                     /F(L5)             41
        WØRD      *HEAD/SIZE*   *PIT/"1"*            /F(L4)             47
        $PIT      =  $PIT  WØRD  ","                 /(L3)              54
L4      BIN       =  BIN   WØRD  ","                 /(L3)              61
L5      BIN       *LIST*  =                                             68
L6      ALPHABET  *PIT/"1"*  =                       /F(L1)             72
        LIST      =  LIST  $PIT                                         78
        $PIT      =                                  /(L6)              83
FIN     SYS       .PRINT  "THE ALPHABETIZED LIST IS :   "  LIST         87
END     BEGIN                                                           92
```

```
****** SNØBØL ØUTPUT ******
THE LIST TØ BE ALPHABETIZED IS :   ARMY,TEST,GLØBAL,ARMØRY,GLØBE,ARM,TENSØR,ALIBI,ARE,GLØW,TENSE,TØTAL,CANCEL,TØNSIL,G
LADIATØR,MØBILE,MØTILE,ANY,TØRSIØN,PLATITUDE,FUMBLE,

THE ALPHABETIZED LIST IS :   ALIBI,ANY,ARE,ARM,ARMØRY,ARMY,CANCEL,FUMBLE,GLADIATØR,GLØBAL,GLØBE,GLØW,MØBILE,MØTILE,PLA
TITUDE,TENSE,TENSØR,TEST,TØNSIL,TØRSIØN,TØTAL,
```

FIG. 1

used for filing words on successive passes. Suppose n is the number of letters in the longest word. The first pass is made on the nth letter of the words, with each word being added to a bin corresponding to this nth letter. Words which are shorter than n letters are filed in a special bin. After this pass, the list of words is reassembled from the bins starting with the special bin, followed by the contents of bins A through Z. The next pass is made on the $(n - 1)$st letter and so on until n passes have been made. When the list is reassembled the last time, the words are in alphabetical order.

The SNOBOL program in Figure 1 executes this radix sort. For simplicity it is assumed that the number, n, of characters in the longest word appears left-

justified on the first data card. Successive data cards contain the list of words with a comma following each word, and with each data card terminating with blanks.

REFERENCES

1. An introduction to COMIT programming. The Research Lab. of Electronics and the Computation Center, M.I.T., 1961.
2. NEWELL, A., Ed. *Information Processing Language-V Manual.* Prentice-Hall, 1961.
3. MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Comm. ACM 3* (Apr. 1960), 184.
4. LEE, C. Y., ET AL. A language for symbolic communication. (Unpublished)
5. MCILROY, M. D. A string manipulation system for FAP programs. (Unpublished)