SP/1—A FORTRAN integrated string processor

I. A. Macleod*

Department of Computing and Information Science, Queen's University, Kingston, Ontario

This paper describes a string processing system implemented by means of a set of routines embedded in FORTRAN. The syntax of the operations provided is described in terms of a set of macros. The system can, however, be usefully applied by the direct use of the embedded routines.

(Received November 1969)

In general, string processing systems deal with data which is in the form of unstructured strings of characters. COMIT (Yngve, 1962), SNOBOL-3 (Farber, Griswold and Polansky, 1966) and SNOBOL-4 (Griswold, Poage and Polansky, 1968) are three well-known string processing languages. Typical of the types of operation possible in these languages are matching, insertion, replacement and concatenation of strings and substrings. With the increasing usage of computers in many different fields, the distinction between numeric and non-numeric applications is becoming less apparent, as for example in information retrieval problems. Consequently it seems desirable that a single programming system should incorporate efficient numeric and non-numeric capabilities. The SP/1 system described here has been designed and implemented as a string processing system embedded in FORTRAN-IV.

To avoid adding to the diversity of programming systems already in existence and since SNOBOL is a well-known language whose syntax is readily adaptable to a FORTRAN environment, the operations provided in SP/1 are similar to those available in SNOBOL-3. Unlike, for example DASH (Milner, 1967), which is a string processing extension embedded in ALGOL, SP/1 is both a syntactic and semantic extension to FORTRAN. The string processing statements can be represented by a set of macros which are expanded into FORTRAN statements by a macro generator (Macleod and Pengelly, 1969) prior to compilation. The macros have been designed so that there is a close similarity between the syntax of the corresponding SP/1 and SNOBOL-3 statements.

For example, the SNOBOL-3 statement

```
REPEAT E "(" *V* ")" = V /S (REPEAT)
```

deletes all the pairs of left and right parentheses from a string E. The corresponding SP/1 statement is

```
100 E : "(" & *V* & ")" —> V GO TO 100
```

In addition, SP/1 provides a data type known as an association which may have a range of alternative values associated with it. This data type is in some ways similar to the pattern type in SNOBOL-IV and the assertion type in AXLE (Cohen and Wegstein, 1965).

A further distinctive feature of SP/1 is that strings are stored as sequences of atoms where an atom is the smallest meaningful unit of the string. The size of an atom is determined on input as shown below, but normally an atom may be regarded as a single character symbol or as a group of consecutive alphameric characters. The latter could be the case for example in text processing where one atom would be equivalent to a word of text. This approach allows the processor to operate on strings composed of text words while retaining the capability to manipulate strings of individual symbols where required. This provides faster operation with a considerable saving in storage requirements in the text processing types of applications where the smallest logical unit of information is a word of text. Thus, essentially, there are two modes of operation, character and text, corresponding to the two types of storage. In the current version of SP/1 mixed mode operations are not allowable. The method of string storage, which involves a hash table, is described elsewhere (Macleod, 1969a).

Statements of the language

The statements described below illustrate how the formats of the macros representing the operations have been designed. The macros are processed into routine calls as is described subsequently. The system may of course be applied directly by calls to these routines without having recourse to the corresponding macros. String and association names are implicitly declared when they are assigned values. There is no explicit name declaration statement.

Matching, insertion and assignment

These operations are fundamental to any pattern directed string processing system and those defined below are similar in effect to the equivalent operations in SNOBOL-3. Assignment is a special case of replacement and the most general form of the match and replacement statement is as follows:

⟨match and replacement⟩ ::= ⟨string name⟩ : ⟨left pattern⟩⟨right pattern⟩$[^1{}_0$ GO TO⟨label⟩]
⟨string name⟩ ::= ⟨FORTRAN unit⟩
⟨FORTRAN unit⟩ ::= ⟨FORTRAN variable name⟩| ⟨FORTRAN array element⟩

where the expression $[^m{}_n]$ of the meta language denotes that the bracketed syntactic unit may occur from n up

* *Work carried out in the Department of Computer Science, Queen's University, Belfast*

to *m* times. An indefinite upper limit is denoted by replacing *m* with an asterisk.

The match and replacement statement causes the concatenated elements of the left pattern to be matched against the named string. If the match is unsuccessful control passes to the next statement. Otherwise the action specified by the right pattern is carried out as described below and if the GOTO field is present, control is passed to the referenced statement.

The left pattern is constructed as below:

```
⟨left pattern⟩ ::= ⟨match list⟩|⟨null⟩
  ⟨match list⟩ ::= ⟨left element⟩|ANCHOR⟨left
                      element⟩|
                   ⟨left element⟩ & ⟨match list⟩
⟨left element⟩ ::= ⟨string name⟩|⟨literal string⟩|
                   ⟨string variable⟩|⟨multiple
                      association⟩|
                   ⟨recursive association⟩
```

The effect of ANCHOR preceding a left element is described below with the matching procedure. Associations are discussed in the following sub-section.

The right pattern takes the following form,

```
⟨right pattern⟩ ::= —⟩⟨insertion list⟩|—⟩⟨null⟩|⟨null⟩
⟨insertion list⟩ ::= ⟨right element⟩|⟨right element⟩ &
                     ⟨insertion list⟩
⟨right element⟩ ::= ⟨string name⟩|⟨literal string⟩|
                    ⟨multiple association⟩
```

except that both right and left patterns cannot be null.

Thus the statement can take the following general forms:

(i) S : L —⟩ R
(ii) S : —⟩ R
(iii) S : —⟩
(iv) S : L
(v) S : L —⟩

In case (i), the substring L, if found in S, is replaced by R. In case (ii), the previous contents, if any, of S are erased and replaced by R., i.e. this is a string assignment operation. In case (iii), the contents of S are erased.

Case (iv) is purely a matching operation and may serve as a conditional branch if the GOTO field is present or as a method of assigning values to strings.

In case (v) the substring L, if found in S, is deleted from S.

The GOTO field may be optionally present in any of the five cases.

Literal strings are written as follows:

⟨literal strings⟩ ::= "[$^*_1$ ⟨character⟩]"

and string variables take one of the following forms

```
⟨string variable⟩ ::= ⟨simple string variable⟩|
                      ⟨balanced string variable⟩|
                      ⟨fixed length string variable⟩
     ⟨simple string variable⟩ ::= *⟨variable name⟩*
   ⟨balanced string variable⟩ ::= *(⟨variable name⟩)*
⟨fixed length string variable⟩ ::= *⟨variable name⟩|
                                   ⟨integer expression⟩*
            ⟨variable name⟩ ::= ⟨string name⟩|⟨null⟩
```

A string constant in either a left or right pattern is simply an occurrence of a string name. They are termed constants in this context in that the substrings they represent are defined before the matching process. The same is also true for literals. On the other hand, symbolic content of the substring matched by the string variable is not predefined and is only determined during the matching process. A simple string variable may match any substring, including a null substring (i.e. one with no symbolic content). A balanced string variable may match any non-null substring which is balanced with respect to left and right parentheses. Finally, a fixed length string variable may match any substring whose length in atoms is that given by the value of the integer expression associated with the name.

The left pattern is matched against the named string element by element and from left to right and a match will succeed when each element has been matched against consecutive substrings of the named string. String variables may have values before matching and where the left pattern cannot be matched, these values will remain unchanged. In the case of a successful match, the variable names will be assigned string values equivalent to the substrings they have been matched against. Where the variable name is null, the matching process is unaltered but no values will be assigned to the matched variables. In all cases matching is performed atom by atom. The format of literals is independent of the atom size as during matching all literals are automatically broken down by the system into the same format as the strings named in the match operation. Further details of the matching procedure, including the special case where a string constant and a string variable with the same name appear together in a left pattern, are described below.

Associations

Essentially an association is the term given to a data type with which more than one value may be associated. Where an association appears in a left pattern it will match a sub-string identical to one of the set of values associated with it. The value set is defined by an association assignment statement.

```
 ⟨multiple association⟩ ::= ⟨association name⟩
⟨recursive association⟩ ::= ⟨association name⟩!
     ⟨association name⟩ ::= %⟨FORTRAN unit⟩
⟨association assignment⟩ ::= ⟨association name⟩ :—⟩
                            ⟨value set⟩
              ⟨value set⟩ ::= ⟨association value⟩|
                            ⟨association value⟩ &
                            ⟨value set⟩
      ⟨association value⟩ ::= ⟨literal string⟩|
                            ⟨string name⟩|
                            ⟨association name⟩
```

In matching, attempts are made to match each value of the association from left to right in the order in which the values were assigned. An association appearing in a right pattern is assumed to represent the substring matched in the last successful match operation.

The simplest use of an association is as a multiple association where an attempt is made to match each of the set of values independently until a successful match is found. Alternatively, an association may be employed recursively, in which case once one value has been matched an attempt may later be made to extend this match by matching a second or further values of the association. For example, if S1 contains the string "ABC + BCD;" then if

```
    %A :—> "A" & "B" & "C" & "D"
    %B :—> %A
then
    S1 : %A! & *SV* & %B! & ";"
```

would assign the value "+" to the variable SV and %A and %B would represent the strings "ABC" and "BCD" respectively, in a replacement operation.

Associations have been developed as they would appear to be useful in two distinct applications corresponding roughly to the two modes of operation, character and text. Firstly, an association can be used to match groups of related characters. In the example above the recursive use of %A allows character strings consisting of any combination of any number of the characters A, B, C and D to be matched. This recursive usage would most often only be of interest when processing strings of individual characters. An example of the second type of application could be as a means of representing synonyms in text processing.
Thus

```
%A :—> "COMPILER" & "TRANSLATOR"
```

would mean that %A would match any occurrence of either COMPILER or TRANSLATOR in a match operation. This type of application will probably be most used when operating in text mode.

Input/Output

The I/O operations are designed to resemble those of FORTRAN. The facilities provided are quite extensive and include the ability to input various types of string, to control the atom size and to file or compact strings into contiguous blocks for possible output to backing store.

The input and output statements take the form

```
 ⟨input statement⟩ ::= READ STRING (1, ⟨label⟩)
                       ⟨name list⟩
⟨output statement⟩ ::= WRITE STRING (2, ⟨label⟩)
                       ⟨name list⟩
       ⟨name list⟩ ::= ⟨string name⟩|⟨string name⟩,
                       ⟨name list⟩
```

The label refers to a labelled format statement which is defined as

```
⟨format statement⟩ ::= %FORMAT (⟨descriptor list⟩)
  ⟨descriptor list⟩ ::= ⟨descriptor⟩|⟨descriptor⟩
                        ⟨separator⟩⟨descriptor⟩
        ⟨separator⟩ ::= , | /
```

The format statement is scanned in much the same way as in FORTRAN and the descriptor separators have the same function as in the FORTRAN FORMAT statement.

On input the descriptors allowed are A, B and T corresponding to fixed length, balanced and strings terminated by a specified character respectively. The maximum atom size is written after the descriptor and can be omitted if it is 1. In A format the length of the string precedes A and in T format the terminator character follows the atom size.

For example

```
     READ STRING (1,901) A, B, C
901  %FORMAT (T;, B4/60A8)
```

will input a character string terminated by a semicolon, a balanced string of atom size 4, take a new card and input a 60 character string of atom size 8.

On output A and B formats are allowed. Here the length specification for A specifies the maximum number of characters to be printed on each line and in B format each parenthesised substring is output on a new line.

H and X formats are also allowed on output and these have their usual FORTRAN meaning.

For example

```
    WRITE STRING (2, 902) LIST
902 %FORMAT (18H LISTING OF OUTPUT /// 30A)
```

prints the heading, skips three lines and outputs the contents of LIST with up to 30 characters per line.

Atom size has no effect on output except that atoms will not be split between lines, i.e. a new line will be taken if the next atom to be printed will overflow the specified line length.

A fuller discussion of input and output including filed I/O is contained in Macleod (1969b).

Other operations

The interaction between string operations and the FORTRAN arithmetic operations is small. The only real area of overlap is in handling of strings representing numeric quantities. Consequently two statements are provided with which it is possible to convert numeric quantities to strings and vice versa. These take the following form:

```
⟨string to numeric⟩ ::= ⟨FORTRAN unit⟩—>
                        NUMERIC⟨string name⟩
⟨numeric to string⟩ ::= ⟨string name⟩—>
                        STRING ⟨numeric⟩
          ⟨numeric⟩ ::= ⟨FORTRAN unit⟩|⟨integer⟩|
                        ⟨real⟩
```

The overflow register will be set, either if an attempt is made to convert a non-numeric string or if the number generated is outside the permitted range. Both real and integer quantities may be converted and no floating or truncation effects will take place.

Alphabetic ordering of strings may be obtained through the application of the string comparison statement. This takes the following form:

```
⟨string comparison⟩ ::= IF STRING⟨string name⟩.
                        ⟨SOP⟩ . ⟨string name⟩ GO TO
                        ⟨label⟩
              ⟨SOP⟩ ::= LT|GE
```

The LT comparison will hold where the first string alphabetically precedes the second, and the GE comparison will hold if the converse is true or if the two are equal. Where the string atoms are not single characters the system accesses the individual characters of the atoms during the comparison process.

Finally, the size of a string given as the number of atoms in the string may be determined by the following statement:

⟨stringsize⟩::=⟨FORTRANunit⟩—⟩SIZE⟨stringname⟩

Note that strings may be copied by matching a string against a left pattern consisting of a single string variable.

Example:

The small example of Fig. 1 program converts FORTRAN II PRINT and READ statements into the FORTRAN IV equivalents. No allowance is made for statements punched starting other than in column 7 but continuation cards are generated if necessary. Anchored mode matching is used to improve efficiency.

The program inputs strings in character format since the length in characters is needed in order to recognise the need to generate continuation lines. The program ends when a FINISH statement is encountered.

The matching procedure

There are two further statements which affect the matching procedure. These are defined as follows:

```
⟨mode declaration⟩ ::= MODE ANCHOR|
                       MODE UNANCHOR
```

Normally a match for the left pattern is attempted starting at the first atom of the named string and if this fails a further match is attempted from the second atom and so on. However, if the statement MODE ANCHOR appears, then all subsequent matches must either succeed starting from the first atom or else they will fail. For example, if S1 contains the string ABCDE, then

```
S1 : "BCD" GO TO 4
```

would succeed and cause a branch to label 4. But

```
MODE ANCHOR
S1 : "BCD" GO TO 4
```

would fail and the succeeding statement would be executed. Once the mode has been set in this way it will remain set until a subsequent MODE UNANCHOR statement is encountered. Alternatively, local anchoring is permitted in which case the mode is only set for that element. An element anchored this way is written as ANCHOR ⟨left element⟩ as shown earlier. So in this case

```
S1 : "BCD" GO TO 4
```

would succeed and

```
S1 : ANCHOR "BCD" GO TO 4
```

would fail, but the anchor mode would not be set for the subsequent statement.

In general, the matching procedure is outlined by the following set of rules.

1. Each element proceeding from left to right, of the left pattern, must match consecutive substrings of the referenced string.
2. Initially, matching is attempted starting from the first atom of the referenced string. Where this fails a new match will be attempted at each subsequent atom either until a match is found or the end of the referenced string is reached. If matching is in anchor mode the match can only be attempted from the first atom.
3. When an element cannot be matched a *re-match* is attempted for the preceding element.
4. Once a locally anchored element is matched none of the preceding elements can be rematched.
5. The last element of a left pattern will always match the longest substring possible.

A string constant, i.e. a string name, will only match a

```
           MASTER CONV
           MODE ANCHOR
      10   READ STRING (1,901) LINE
     901   %FORMAT (72A)
           LINE : *L/6* & "PRINT" & *LAB* & "," & *LIST*
         1      —> L & "WRITE (2," & LAB & ")" & LIST GO TO 30
C          REPLACES PRINT WITH WRITE STATEMENT
           LINE : *L/6* & "READ" & *LAB* & "," & *LIST*
         1      —> L & "READ (1," & LAB & ")" & LIST GO TO 30
C          REPLACES READ STATEMENT
C          ADJUSTS FORMAT OF READ STATEMENT
           LINE : *L/6* & "FINISH" GO TO 40
C          TEST FOR FINISH STATEMENT
      20   WRITE STRING (2,901) LINE
           GO TO 10
      30   N —> SIZE (LINE)
           IF (N.LE.72) GO TO 20
C          BRANCH UNLESS STATEMENT LONGER THAN 72 CHARS
           LINE : *L/72* —> "     X"
C          REPLACE FIRST 72 CHARS WITH CONTINUATION
           WRITE STRING (2,901) L
C          OUTPUT FIRST 72 CHARS
           GO TO 20
C          OUTPUT CONTINUATION LINE
      40   STOP
           END
           FINISH
```

Fig. 1. Example to convert PRINT and READ statements to equivalent FORTRAN IV statements

substring whose atomic content is identical to that of the constant. This is also the case with literals. A fixed length variable will match any substring whose length is that specified in the variable.

A simple string variable initially matches a null substring. A balanced string variable will match the shortest non-null substring balanced with respect to left and right parentheses. A multiple association will match the first substring which is identical to one of its values, and attempts are made to match these in the order in which they were assigned. This is also the case for recursive associations.

Where a string name appears first as a variable and then as a constant, it is said to be *back referenced* and the constant will be matched against a substring equal to that of the substring matched at that time by the corresponding variable, irrespective of the previous content, if any, of the constant.

By rematching is meant an attempt to match the element against an alternative substring of the string reference. Rematching follows the rules below:

1. String constants, literals and fixed length variables can only match one substring from any given atom and therefore cannot be rematched.
2. Simple string variables are rematched by extending the previous match by one atom.
3. Balanced string variables are rematched by extending the previous match by the minimum number of atoms required to maintain a balanced match.
4. A multiple association can be rematched if any of the succeeding association values can be matched in place of the current matched value.
5. A recursive association can be rematched if the current match can be *extended* by matching any of the association values.

For example, if

```
%MA:—)"A" & "B" & "C"
%RA:—)"X" & "Y" & "Z"
```

and the string T contains "ABAXYA", then the matching operation,

```
T: *S* & %MA & *V* & %RA! & V
```

would proceed as follows (assuming character mode):

```
S = null, MA = "A", V = null,       RA fails
                    V = "B",        RA fails
                    V = "BA",       RA = "X",
                                      V fails
                                    RA = "XY",
                                      V fails
                                    RA fails
                    V = "BAX",      RA = "Y",
                                      V fails
                                    RA fails
                    V = "BAXY",     RA fails
                    V = "BAXYA",    RA fails
                    V fails
          MA fails
S = "A", MA = "B"   V = null,       RA fails
                    V = "A",        RA = "X",
                                      V fails
                                    RA = "XY",
                                      V = "A"
```

and the match finally succeeds. This example illustrates the use of both types of associations and also of back referencing.

A further example is given by the following statements which transform the fully parenthesised expression in E into Polish form.

```
    %OP—) "**" & "*" & "/" & "+" & "—"
1 E : "(" & * (U) * & %OP & * (V) * & ")"
          —)%OP & U & V GOTO 1
```

The statement labelled 1 would be repeated until the match no longer held and if, for example, E held the string ((A*B) + (C/(D–E))) this would be transformed by the successive match and insert operations as follows:

```
+(A*B)  (C/(D–E))
+*AB(C/(D–E))
+*AB(C/–DE)
+*AB/C–DE
```

When the final parenthesis free string is reached, the next successive statement is automatically executed.

Implementation

SP/1 has been implemented on an I.C.L. 1907 computer. The routines used are mostly written in PLAN, the 1900 assembly code, mainly because character handling in FORTRAN can be extremely inefficient. Matching and insertions are the most complex of the routine set which also includes input, output and storage administration routines.

Match operations are carried out by a routine MATCH, and this is called by a statement of the form

CALL MATCH (S, N, E_1, T_1, E_2, T_2, . . ., E_N, T_N)

where S is the string reference and the elements of the left pattern are each represented by an E_i, T_i pair generated by the macro processor. The E_i are either names or literals and the T_i are integers specifying the type of each element. In addition, fixed length variables generate a third parameter giving the length of the variable.

For example, the statement

```
A: X & *Y /3 * & %Z!
```

is transformed by the macroprocessor to the form

```
CALL MATCH (A, 3, X, 1, Y, 4, 3, Z, 8)
```

where the integers 1, 4 and 8 identify constants, fixed length variables and recursive associations respectively.

The parameters are held in three word entries in a table and each entry contains the parameter name, type and the address of the current position on the string currently matched by this element and this latter is set for each element after it has been successfully matched.

Internal routines, appropriate to each element type, perform the matching and when all the parameters have been matched the parameter table is rescanned and all string variables and associations have string values assigned to them identical to the substrings they match.

The insert routine performs both the insertion and assignment operations. It is generated in the form

CALL INSERT (K, S1, N, E_1, T_1, E_2, T_2, . . ., E_N, T_N)

The parameters generated are equivalent to those in the MATCH routine except that K is either a flag with the value −1, indicating assignment, or a call to the match routine. The latter returns a value of zero for an unsuccessful match, in which case no insertion takes place, and 1 for a successful match, in which case the concatenated string value of the parameters is substituted for the matched portion of the string. The position of the matched substring is indicated by pointers held in a common block of store.

For example,

```
A:—⟩X & Y
```

is generated as

```
CALL INSERT (−1, A, 2, X, 1, Y, 1)
```

and

```
A: %X!—⟩ Z & Y
```

is generated as

```
CALL INSERT (MATCH(A, 1, X, 8), A, 2, Z, 1, Y, 1)
```

Routines appropriate to each element type then insert each substring associated with each of the elements.

The use of these statements as simply FORTRAN calls is of course possible whether or not the code is being preprocessed. This has the advantage that the number of significant elements in a match or insert operation and even the type of these elements can be altered dynamically during the execution of the program if required.

For example,

```
CALL MATCH (S1, N, E1, I1, E2, I2)
```

would, with N = 2, I1 = 1 and I2 = 5, be equivalent to the statement

```
S1 : E1 & *E2*
```

while, with N = 1 and I1 = 6, this same statement would be equivalent to

```
S : *(E1)*
```

Descriptions of storage administration and other operations are given, together with a fuller discussion of the system, in Macleod (1969b).

Summary

The system is in principle similar to SNOBOL–3. Two major additions are the provision of the association data type and the option of treating data as strings of either characters or text words. It is hoped that these additions will noticeably enhance the system although their true worth can only be gauged after considerable experience.

An important SNOBOL feature which has been omitted is the indirect referencing facility. In SNOBOL, if S1 is the string "ABC" then $S1 refers to the string whose name is ABC and $ ($S1) refers to the string whose name is the contents of the string ABC. This allows strings to be referenced indirectly during the execution of a program. Because SP/1 uses FORTRAN names, the symbolic content of string names is lost after compilation and thus there is no correspondence between a string name and a string with the same symbolic content as the name. It would be possible to give strings literal names but this would increase considerably the time required to locate a particular string and since string names can be represented by FORTRAN array elements a degree of indirect referencing can still be retained.

As well as providing a useful extension to FORTRAN, this system also illustrates how a suitable macroprocessor can be applied to give a considerable syntactic extension to an existing high level language.

Acknowledgements

Thanks are due to Dr. R. M. Pengelly for many helpful suggestions given during the course of this work, and also to the referee for several useful criticisms.

References

Cohen, K., and Wegstein, J. H. (1965). AXLE: An Axiomatic Language for Symbol Manipulation, *CACM*, Vol. 8, pp. 657–666.

Farber, D. J., Griswold, R. E., and Polansky, I. P. (1966). The SNOBOL-3 Programming Language, *Bell System Technical Journal*, Vol. XLV, pp. 895–944.

Griswold, R. E., Poage, J. F., and Polansky, I. P. (1969). *The SNOBOL-4 Programming Language*, Prentice-Hall.

Macleod, I. A. (1969a). A Method of String Storage, *Quarterly Bulletin of the Information Processing Society of Canada*, Vol. 9, pp. 13–17.

Macleod, I. A. (1969b). An Information Processing Language, Ph.D. Thesis, Queen's University, Belfast.

Macleod, I. A., and Pengelly, R. M. (1969). The MP/1 Macroprocessor (unpublished).

Milner, R. (1967). String Handling in ALGOL, *The Computer Journal*, Vol. 10, pp. 321–324.

Yngve, V. H. (1962). COMIT as an IR Language, *CACM*, Vol. 5, pp. 19–28.