On string concepts and their implementation

R. J. W. Housden

School of Computing Studies, University of East Anglia, Norwich, NR4 7TJ

A model is described for string handling, distinguishing between fixed length vectors of characters and strings which vary in length under string operations. String processing facilities in PL/I and ALGOL 68-R are considered with particular reference to variability of length of segments of strings. Operators and procedures are defined in ALGOL 68-R for processing both fixed and variable length strings.

(Received March 1974)

On string concepts and their implementation

A string is a set of scalar objects having a linear positional relationship one to another. Some would say that the notion of strings is an unnecessary one and that all strings are vectors. Nevertheless, it is convenient to separate the concepts of strings and vectors because of the different operations that are performed on them. The string concept may be applied to any variety of scalar data; strings of characters and strings of bits are recognised in some programming languages. In practice the concept of a string is most often utilised with character data and we shall consider only strings of characters.

Character-string attributes

Two important string characteristics which are independent of any context in which a string is used are
(*a*) current length of a string
(*b*) variability of string length.
The first of these indicates at any time the number of characters residing in the string. The variability of string length is a distinguishing feature between strings and vectors.

String operations

The fundamental operations on character strings include the following:
1. Create a string of characters.
2. Concatenate two strings to form a new string.
3. Extract a segment of a string.
4. Search within a string for a given substring.
5. Compare two strings.
6. Delete a substring or replace it with another substring.
7. Insert a string within another string at a specified position.
8. Interrogate the length of a string.

In addition to these processes there must also be defined procedures for the transput (i.e. input and output) of strings. From these and other basic processes can be built much more complex operations on strings.

A model for string handling

1. String constants are enclosed in quotes,
   e.g. “THIS$_\wedge$IS$_\wedge$A$_\wedge$STRING$_\wedge$CONSTANT”
   is a string constant of length 25. To retain a string constant in a string variable S, say we use the assignment operator.

   $$S = \text{“BLACK}_\wedge\text{CAT”}$$

   S now identifies BLACK$_\wedge$CAT, a string of length 9. An embedded quote may be represented by a double quote; thus “”“” has the value ”. A null string is represented by “”.
2. A common operation performed on strings is concatenation; one string is concatenated to another by placing one at the end of the other so forming a new string whose length is the sum of the lengths of the original strings. Concatenation is indicated by the operator +.
   Example “SUN” + “DAY” has the value SUNDAY
3. A segment consisting of the Mth through Nth characters of a string, S, is referred to as $S(M:N)$. Thus as a result of executing the two instructions

   $$S = \text{“ALIBI”}$$
   $$T = S(2:4)$$

   T has the value LIB. The nature of the expression $S(M:N)$ is further illustrated by the instructions

   $$S = \text{“THE}_\wedge\text{BLACK}_\wedge\text{CAT”}$$
   $$S(5:10) = \text{“”}$$

   as a result of which S has the value THE$_\wedge$CAT. It is clear from this example that $S(M:N)$ returns addressability to the desired segment of S. It does not create a copy of the desired segment. Such functions are referred to as *pseudo variables* (PL/I) or *selection operators*; they select and return addressability to data.
4. For searching within a string we define a function INDEX which returns the index to the start of the leftmost substring of its first argument that matches the second argument.
   The value of INDEX(“BANANA”, “NA”) is 3. We use the convention that zero is returned if the search fails.
5. For comparisons of strings we use the standard relational operators <, < =, >, > =, =, ≠. If the two strings are of different lengths, the shorter string is extended by fictitious ‘empty’ characters on the right to match the longer one in length. The empty character is defined to collate before any genuine character. Thus “FRED” < “FRED$_{\wedge\wedge\wedge}$” has the value **true**. The comparisons yield lexical ordering.

6,7. Substring insertion, deletion and replacement may be expressed in terms of operations involving segments of strings. For example

$$\text{S(M:N)} = \text{“ABC”}$$

replaces the Mth through Nth characters of the string S by ABC. The length of the replacement string need not be the same as that of the segment $S(M:N)$. We have already seen that

$$S(M:N) = \text{“ ”}$$

deletes the Mth through Nth characters of S. Since a substring is also a string it follows that substrings also exhibit variability of length under string operations.

Insertion of one string in another at a specified position may also be regarded as a special case of substring replacement in which a null segment is replaced by the string to be inserted. This raises the problem of notation for a null segment. We adopt the convention that $S(M:N)$ for $M > N$ represents the null segment *preceding* the Mth character of S. Insertion of ABC *before* the Mth character of S is then the result of

$$S(M:M - 1) = \text{“ABC”}$$

Another way of denoting insertion of a string T before the Mth character of S is

$$S(M:L) = T + S(M:L)$$

where L is the length of the original string S.

What meaning, if any, can be associated with $S(M:N)$ when either or both of the segment bounds, M and N, have values outside the range of the current value of S, i.e. if M or N is less than 1 or greater than the current length, L, of S? We have already dealt with the case $N < M$. If we also adopt the convention that $S(M:N)$ is equivalent to $S(\text{MAX}(M, 1): \text{MIN}(N, L))$ then $S(M:N)$ refers to a valid segment of S (possibly null) for all integer values M and N. It follows that if S has the value BE then as the result of

$$S(10:16) = \text{“WARE”}$$

S has the value BEWARE, a string of length 6. This is consistent with the convention that no gaps are created as a result of operations on strings.

Implementations of strings

Having set up a model for string handling we now consider selected features of the string handling facilities provided in a variety of programming languages and the extent to which our model is realised in practice. There are a number of languages, and extensions to languages, designed to handle strings. COMIT (Yngve, 1962), SNOBOL-3 (Farber, Griswold and Polonsky, 1966) and SNOBOL-4 (Griswold, Poage and Polonsky, 1969), are three well known string-processing languages. The early 'general purpose' languages were almost entirely arithmetic and made no provision for strings. With the increasing usage of computers in many different fields the distinction between numeric and non-numeric computing has become less apparent. The desire for a single programming system incorporating efficient numeric and non-numeric capabilities led to the development of several extensions to FORTRAN and ALGOL to provide character and string handling facilities. Two examples of such extensions, each providing string facilities of the SNOBOL type, are DASH, a set of procedures for dynamic string handling in ALGOL (Milner, 1968) and SP/I, a string processor based on a set of routines embedded in FORTRAN (Macleod, 1970). More recent general purpose languages, notably PL/I and ALGOL 68, recognise strings but make no distinction between strings and vectors and regard a string as a vector of characters. BASIC, designed specifically for computing science education, provides simple facilities for string handling. Another teaching language with simple but adequate text processing capabilities is EASNAP, an extension of SNAP (Barnett, 1969; Housden, 1971).

It would be impossible to give here a full description of the string handling facilities in such a wide range of languages. The reader interested in such details is referred to Sammet's comprehensive volume, *Programming languages: History and Fundamentals* (1969).

SNOBOL and imitations of SNOBOL provide powerful string processing and pattern matching facilities at a high level. The SNOBOL user need not concern himself with the low level realisation of these facilities PL/I and ALGOL 68 on the other hand provide only primitive operations on strings in terms of which more complex procedures may be defined. Since we are concerned only with the fundamental concepts of strings we shall confine our attention to their implementation at the lower level. The two features in which we are primarily interested are variability of string and substring lengths, and addressability of segments of strings.

String length

Many systems impose some restrictions in that strings must be declared with a maximum length. This is mainly for the convenience of implementors so that sufficient space may be allocated for each string when it is first encountered. It also avoids the inefficiencies of dynamic storage allocation and collection which would otherwise result from runtime variations in string length, but it can waste a lot of space, since generally an over-large margin of error is allowed. For most applications the restriction is not serious.

The differences between implementations of string handling facilities are more apparent when we consider the results of operations on segments of strings. In our model, the length of a segment of a string also exhibits variability under string operations. This characteristic is reflected in only a few implementations.

String handling in PL/I

String declarations

Example:

```
DECLARE S1 CHAR(15), S2 CHAR(20) VARYING,
        S3(12) CHAR(20) VARYING;
```

S1 is here declared as a fixed length string of 15 characters that is the length of S1 returned by the function, LENGTH (S1) is 15 throughout execution. A string shorter than 15 characters assigned to S1 would first be extended by space characters on the right to a length of 15. Thus S1 is indistinguishable from a vector of characters.

S2 is a variable length string with a maximum length of 20 characters. The current length of S2 at any time during execution is returned by the length function, LENGTH (S2).

S3 is an array of 12 character strings each of variable length not exceeding 20 characters.

Any attempt to assign to a string variable a string of length greater than the declared maximum length for that variable results in truncation at the right.

Concatenation

String assignments and concatenation are illustrated by the following example. The double vertical bar is the built-in concatenation operator.

```
DECLARE DAY CHAR(9) VARYING;
DAY = 'SUN'||'DAY';
```

Result: DAY identifies the string SUNDAY.

Searching

Searching within a string is accomplished by the INDEX function as described in our model for string handling.

Example:

```
DECLARE FRUIT CHAR (12) VARYING;
FRUIT = 'BANANA';
N = INDEX(FRUIT, 'AN');
```

Result: N = 2.

Substring selection

The PL/I substring selector function SUBSTR has the general form

$$\text{SUBSTR(string}, i, j)$$

where 'string' is the parent string from which the substring is to be extracted, i is the index to the first character of the substring and j is the length of the substring. Note the difference between SUBSTR(S, I, J) and S(I:J) as defined in our model. Some implementations of PL/I allow the form SUBSTR(S, I) meaning the Ith and successive characters of S.

The length of the substring returned by SUBSTR is equal to J if J is specified, or is LENGTH(S) − I + 1.

SUBSTR may be used on the left of an assignment statement and in PL/I terms is therefore a pseudo-variable, i.e. it returns addressability.

It is important to note that SUBSTR(S, I, J) is a substring of *fixed* length whether applied to a fixed length string or to a variable length string, so SUBSTR does not have the properties of S(I:J) in our model.

```
Example:  X = 'BEWARE';
          SUBSTR(X, 3, 3) = 'IG';
Result:   X = 'BEIG‸E';
```

Similarly an assignment of length greater than J to a substring SUBSTR(S, I, J) results in truncation at the right before the assignment takes place. Thus we see that the utility of SUBSTR as a substring selector function which returns addressability is very limited.

The desired results may be obtained by instructions such as

```
          S = 'BEWARE';
          S = SUBSTR(X, 1, 2)||'IG'||SUBSTR(X, 6, 1);
Result:   S = 'BEIGE'
and       S = 'THE‸BLACK‸CAT.';
          S = SUBSTR(S, 1, 4)||'GREY'||SUBSTR(S, 10, 5);
Result:   S = 'THE‸GREY‸CAT.'
          S = SUBSTR(S, 1, 4)||'BLACK‸AND‸WHITE'||
            SUBSTR(S, 9, 5);
Result:   S = 'THE‸BLACK‸AND‸WHITE‸CAT.'
```

Execution of such instructions involves unnecessary creation and copying of substrings whose value and position in the string are not changed as a result of these operations. For example to delete the *M*th through *N*th characters of a string it is only necessary to replace the *M*th and successive characters by the $(N + 1)$th and successive characters leaving the first $M - 1$ characters unaffected.

String handling in ALGOL 68

ALGOL 68 recognises the need for flexibility in string declarations. The user may choose between efficient operations on character strings declared with fixed length, and inefficient manipulation of strings which are truly variable in length. The declaration

[1:100] **char** *s*;

defines *s* as a character array of 100 elements. At no time can the length of *s* differ from 100. Although this form of declaration of *s*, as a fixed vector of characters, ensures efficient handling of *s* and of segments of *s*, the ALGOL 68-R implementation imposes restrictions which are not immediately apparent. For example when a string value for *s* is input from a data stream the input is terminated after reading 100 characters or sooner on encountering some specified string terminator. In the latter case the length of *s* is reduced from 100 to the number of characters actually input to *s*. The length of the string *s* cannot then be increased as a result of further operations on *s*. Once the length of *s* has been fixed by execution of either a string assignment instruction or an input instruction, any attempt to vary it without first redeclaring *s* results in an execution error.* Clearly this facility is inadequate for general string processing. However, it is possible in ALGOL 68 for the user to define his own assignment operator for vectors of characters so as to overcome this difficulty. User defined operators are described below in connection with operations on segments of strings.

An alternative form of string declaration is

[1:100 **flex**] **char** *s*;

(or **flex** [1:100] **char** *s*; in the Revised Language) ,

s is here defined as a one dimensional character array (or row of characters) of variable length. Initially sufficient space is allocated to store up to 100 characters but at any time during execution this length will be automatically extended or contracted if necessary to accommodate a string of different length. The actual length of the string value currently held in the variable *s* is obtained by applying the array operator *upb* which gives the upper bound of the array, in this case the number of characters in *s*. A special case of the declaration of a character string variable of flexible length is

string *s*;

The mode **string** is defined in the standard library prelude as

mode string = [1:0 **flex**] **char**;

A string variable declared in this way is allocated no space until a string value is actually assigned to it. The necessary space is then obtained from the **heap**, a pool of free space. Any array variable declared with flexible bounds is allocated **heap** space and as a consequence incurs considerable run-time overheads.

Apart from assignments, comparisons and transput operations ALGOL 68-R provides few built-in string operations. The use of string constants in assignment and print instructions is illustrated by the following example.

Example:

```
begin string s;
  S := "THE‸BLACK‸CAT.";
  print (("STRING‸S:‸‸", S, Newline,
         "LENGTH:‸‸", upbS, Newline))
end
```

Resulting output

```
STRING S: THE‸BLACK‸CAT.
LENGTH : +14
```

Although defined in full ALGOL 68 concatenation is not a system defined string operation in ALGOL 68-R but the concatenation operator, +, is easily defined as follows.

```
op + = (string A, B) string:
begin int m = upbA, n = upbB; [1:m + n] char s;
  if m > 0 then s[1:m] := A fi;
  if n > 0 then s[m + 1:m + n] := B fi;
  s
end;
```

First we declare a row of characters just long enough to hold string *A* followed by string *B*. Then if *A* is not null its value is assigned to the 'trimmed' array consisting of character positions 1 to **upb***A* of *s* (trimmed arrays are described below). Similarly if *B* is not null its value is assigned to the remaining character positions in *s*† and finally the value of *s* is returned as the result of the operation.

The concatenation operator, +, may now be used as follows.

```
string s;
s := "SUN" + "DAY";
```

Two further string concatenation operators **plus** and **prus** are defined in the official report. *a* **plus** *b* is equivalent to $a := a + b$; *a* **prus** *b* is equivalent to $b := a + b$.

String segments

To refer to a string segment we use the concept of a trimmed array. Thus if *s* is of mode **string** or [] **char** (pronounced row of characters) the trimmed array $S[M:N]$ refers to the *m*th through *N*th elements (characters) of *s*. Unlike the segment S(M:N) of our model, no meaning can be associated with $S[M:N]$ unless it is within S; that is $S[M:N]$ is not defined for $M > N$, $M < 1$ or $N >$ **upb***S*.

*These observations are based on test programs compiled and run on an ICL 1900 series computer using the ALGOL 68-R compiler (Currie, 1970; Woodward and Bond, 1972).

†In Ch. 0.10.6 of their *Informal Introduction to Algol* 68 (1971), Lindsey and van der Meulen define a concatenation operator possessing a routine in which these assignments are unconditional but tests on ALGOL 68-R show that assignments to *s*[1:0] are illegal in this implementation (this is due to a known bug in the index checking). Hence the necessity for conditional assignments.

The length of $S[M:N]$ is $N - M + 1$ and even though S may be declared with flexible bounds the length of $S[M:N]$ for given values of M and N remains constant under all string operations. It follows that a sequence of instructions such as the following will not produce the results that we may desire.

Example:

```
string S := "THE∧BLACK∧CAT∧CLIMBED∧A∧
   TREE.";
print((S,newline));
S[5:10] := "WHITE∧";
print((S,newline));
S[5:10] := "GREY∧";
print((S,newline));
S[5:9] := "";
print((S,newline));
S[5:upbS] := "BLACK∧AND∧WHITE∧" + s[5:upbS];
print((S,newline));
      :
```

A program containing this sequence of instructions was compiled and run using the ALGOL 68-R compiler and produced the following output.

```
THE∧BLACK∧CAT∧CLIMBED∧A∧TREE.
THE∧WHITE∧CAT∧CLIMBED∧A∧TREE.
THE∧WHITE∧CAT∧CLIMBED∧A∧TREE.
THE∧WHITE∧CAT∧CLIMBED∧A∧TREE.
THE∧WHITE∧CAT∧CLIMBED∧A∧TREE.
```

We see that assignment of a string to a trimmed row of characters, or segment of a string, is performed only if the length of the string to be assigned is equal to the length of the segment on the left of the assignment, otherwise the instruction is ignored and no warning given. This is a rather worrying feature of this implementation. It would be reasonable to expect either that fixed length assignments would be performed as in PL/I, that is with short strings extended by spaces on the right and long strings truncated at the right before assignment, or that an error message would be output. The following instructions produce the desired results but are subject to the same criticism as the similar sequence in PL/I above. We assume that the concatenation operator + has already been defined.

```
string S := "THE∧BLACK∧CAT∧CLIMBED∧A∧
   TREE.";
print((S,newline));
S[5:10] := "WHITE∧";
print((S,newline));
S := S[1:4] + "GREY∧" + S[11:upbS];
print((S,newline));
S := S[1:4] + S[10:upbS];
print((S,newline));
S := S[1:4] + "BLACK∧AND∧WHITE∧" +
   S[5:upbS];
print((S,newline));
```

Assignment to fixed length strings

It is of course possible to define our own assignment operator for fixed length strings to produce results compatible with those obtained in PL/I. We define a string assignment operator '< −' so that the result of

S1 '< −' S2

is assignment to S1 of a copy of S2, extended by spaces or truncated at the right, if necessary, to a length equal to that of S1.

```
op '< −' = (ref string A, string B) void:
begin int m = upbA, n = upbB;
   if flexible A then A := B elsf
      m = 0 then skip elsf
      n = 0 then clear A elsf
      m < n then A := B[1:m]else
      A[1:n] := (clear A; B)
   fi
end;
```

The first condition deals with the case in which the operand A has been declared with flexible bounds and therefore no adjustment to the length of B is necessary. The operators **flexible** and **clear** are special features of ALGOL 68-R.

Example:

```
   [1:25]char S; clear S;
   S '< −' "A∧BLACK∧AND∧WHITE∧CAT.";
   S[9:18] '< −' "";
Result:    S = "A∧BLACK∧∧∧∧∧∧∧∧∧∧∧CAT.∧∧∧"
```

String segments of variable length

In our model the segment $S(M:N)$ of a string S was defined, but possibly null, for all possible integer values of M and N. $S(M:N)$ could be used on the left of assignment instructions and exhibited variability of length under assignment operations. We now consider the implementation of $S(M:N)$ in ALGOL 68. We have seen that the trimmed row of characters $S[M:N]$ does not have the desired properties. First we define a mode **substring** as a structure composed of three fields S, M and N.

```
mode substring = struct (ref string S, int M, N);
```

Next we define an operator **str** which when applied to an object of mode **substring** extracts and returns the value of the string segment to which the object refers.

```
op str = (substring A) string:
begin string T; int I, J;
   T := S of A; I := M of A; J := N of A;
   if I < 1 then I := 1 fi;
   if J > upbT then J := upbT fi;
   if I > J then "" else T[I:J] fi
end;
```

Now we can define our operator '< −' for assignment to a segment of a string.

```
op '< −' = (substring A, string B) void;
begin substring SS1, SS2;
   SS1 := (S of A, 1, M of A − 1);
   SS2 := (S of A, N of A + 1, upbS of A);
   S of A := str SS1 + B + str SS2
end;
```

Having defined a mode **substring** and operations on objects of this mode, we may wish to write instructions of the form

(S, 5, 10) '< −' "BLACK∧AND∧WHITE";

to replace a segment of a string S by a longer string or (S, 5, 10) '< −' ""; to delete a segment and (S, 5, 4) '< −' "GREY∧AND∧"; to insert a string before the fifth character of S. Unfortunately this is not possible. A 'collateral' such as (S, M, N) can only be used to display a structure when the context clearly distinguishes it from an array, i.e. only in strong positions. The following example illustrates assignment to segments of strings using the operators defined above.

example:

```
begin string S := "THE∧BLACK∧CAT.";
   print((S,newline));
   (substring val (S, 5, 10)) '←' "WHITE∧";
   print((SS,newline));
   (substring val (S, 5, 10)) '←' "";
   print((S,newline));
   (substring val (S, 5, 4)) '←' "BLACK∧AND∧WHITE∧";
   print((S,newline))
end
```

```
output:  THE‸BLACK‸CAT.
         THE‸WHITE‸CAT.
         THE‸CAT.
         THE‸BLACK‸AND‸WHITE‸CAT.
```

The Revised ALGOL 68 provides a new form of the cast which allows us to write instead of

(substring val (*S*, 5, 10)) '←' "WHITE";

the tidier form

substring (*S*, 5, 10) '←' "WHITE";

At this stage we might ask what we have achieved. Clearly we have demonstrated that the concept of a segment of a string, as defined in our model, can be implemented in ALGOL 68. The implementation is obviously so inefficient as to be of no practical value. Each time the operator '< –' is encountered the routine possessed by it is invoked to achieve the desired effect. This routine in turn invokes the routine possessed by the operator **str**. That an experienced ALGOL 68 programmer could define more efficient procedures for assignment to segments of strings is not doubted, but it is questionable whether it is possible to improve on simple concatenation and assignment such as

S := *S*[1:4] + "BLACK‸AND‸WHITE‸" + *S*[5:**upb***S*];

The last clause in our definition of '< –' has precisely this form. It appears that there is no simple way of avoiding the inefficiency inherent in variable length assignments to trimmed strings. On the positive side the routine possessed by **str** deals with all possible integer values for *M* and *N*.

String comparisons

When comparing strings of characters in ALGOL 68, successive characters from each string are compared, using the relational operators <, < =, > and > =, until a decision is made or the shorter string is exhausted. Details of the routine possessed by the operator < are given in Ch. 0.10.6 of *An informal introduction to Algol* 68 (Lindsey and van der Meulen, 1971).

The ordering so obtained is lexicographic, that is "AB" < "ABC" is **true** and "AAB" < "AB" is **true**.

To test the identity of two strings the operators = and ≠ are used.

Searching a string

The procedure index receives as parameters two strings, searches for the first occurrence of the second string as a substring of the first and returns the index of the start of the substring if found and zero otherwise.

```
proc index = (string A, B)int:
begin int M := 0, N: = upbB − 1;
  if upbB > 0 then
    for L to upbA − N while M = 0 do
    if B = A[L:L + N]then M := L fi fi;
  M
end;
```

A similar procedure **char in string**, to locate a given character in a string, is described in the ALGOL 68 report, section 10.5.1.2(n).

ALGOL 68 operations on strings—conclusion

Many of the string operation declarations considered in the preceding paragraphs belong to the standard-prelude of ALGOL 68, i.e. they are built-in. Some of the built-in ALGOL 68-R operations do not satisfy the requirements of our model and as we have seen some phrases although legal do not have the desired effects. In some cases there are alternative and perhaps better ways of achieving the desired results, particularly in the case of variable length assignments to segments of strings and of course nothing prevents us from declaring our own string operators within each particular program. If we wish to apply our own operations in several string processing programs, or to enable others to use them, then they can be implemented efficiently as a 'library-prelude', i.e. as an extension to the standard-prelude.

String handling in BASIC

There is as yet no agreed standard for BASIC and the differences between various dialects are most apparent when we consider the facilities for handling strings. A preliminary specification for the language has been prepared (Bull and Freeman, 1971) which conforms, with few exceptions, to Dartmouth BASIC version six. In many implementations string variables may be declared with a maximum length. The current length of any string is returned by the LEN function. The proposed standard provides a variety of string functions for insertion, deletion and replacement of substrings. Also provided is a function POS(A$, B$, X) which is similar to our **index** function but returns the index to the *X*th occurrence of B$ in A$ (0 if not found). The value of a segment consisting of the *X*th through *Y*th characters of a string A$ is returned by the function SEG$(A$, X, Y), whilst the substring of *Y* characters of A$ starting with the *X*th is returned by SUB$ (A$, X, Y). Neither function returns addressability but the function REP$(A$, B$, C$, X) is available for replacing the *X*th occurrence of B$ in A$ by C$. It is not clear from the preliminary specification whether REP$ performs 'fixed length' replacements, with truncation or appended spaces at the right as necessary, or 'variable length' replacements as in our model. Concatenation of strings and assignment of string expressions are available as primitive operations in BASIC.

An interesting dialect of BASIC particularly with regard to string handling is the Hewlett Packard implementation (1970). This provides a substring expression A$(I, J) which returns a reference to the *I*th through *J*th characters of A$. It can therefore appear on the left of string assignment instructions. The abbreviated form A$(I) refers to the *I*th and successive characters of A$. In assignments to a substring A$(I, J) the source string is truncated or extended by spaces if necessary to the length J − I + 1 of A$(I, J).

A FORTRAN string processor

Standard FORTRAN does not recognise objects of type string. There are in existence several extensions to FORTRAN to provide facilities for handling strings. Some are purely semantic extensions consisting of library routines for string declaration, storage organisation and processing. Other extensions provide additional syntax so that string operations may be expressed in a more natural form. The syntax extensions must then be preprocessed to produce standard FORTRAN. The FORTRAN implementation of our model for string processing is at the semantic level only and consists of a package of library functions and subroutines. The use of some of these routines is illustrated by the text analysis program below. For comparison an equivalent ALGOL 68 program is given in Appendix 1.

```
$JOB,A240 HOUSDEN
$LIB,STRINGS
      MASTER TEXT ANALYSIS
      WRITE(2,101)
101   FORMAT(14H1TEXT‸ANALYSIS ///)
C   Declare a string variable ITEXT, maximum length 500
      CALL DECLST(ITEXT,500)
C   Declare a string IS of length 6 characters
      CALL DECLST(IS,6)
C   Assign the constant "‸ . , ; −←" to IS
        CALL COPYST(IS,ICONST(6,'‸ . , ; −←'))
C   Read a string and print it
      CALL READST(ITEXT)
      CALL WRITST(ITEXT)
      CALL NEWLINE(2)
C   Initialise counts and a flag,LASTSP
      NS,NW,NC,LASTSP = 0

      DO 20 I = 1, LENGTH(ITEXT)
C   Search IS for Ith character of ITEXT
      L = 1 + INDEX(IS,ISUBST(ITEXT,I,I))
      GO TO (10, 12, 11, 20, 20, 20, 12), L
C   Ith character of ITEXT not in IS
10    NC = NC + 1
      LASTSP = 1
      GO TO 20
C   full stop: increase sentence count
11    NS = NS + 1
C   space or newline: increase word count
12    IF(LASTSP.EQ.0) GO TO 20
      LASTSP = 0
      NW = NW + 1
C   ignore comma, hyphen and semicolon
20    CONTINUE
      AVW = FLOAT(NW)/NS
      AVC = FLOAT(NC)/NW
      WRITE(2, 100) NS,NW,NC,AVW,AVC
100   FORMAT(1HO, 'NUMBER OF SENTENCES =',I4//
  1     1H‸,'NUMBER OF WORDS              =',I4//
  2     1H‸,'NUMBER OF SYMBOLS            =',I4//
  3     1H‸,'AV. NUMBER OF WORDS/SENTENCE =',
                                            F8.2
  4     1H‸,'AV. NUMBER OF SYMBOLS/ WORD  =',
                                            F8.2)
      STOP
      END
      FINISH
$DATA
HOUSTON IS ON THE TEXAS GULF COAST. THE      *
CLIMATE IS SEMI-TROPICAL AND CONDUCIVE TO    *
OUTDOOR ACTIVITIES. HOUSTON IS THE SIXTH     *
LARGEST CITY IN THE UNITED STATES. IT IS ALSO *
THE THIRD LARGEST SEAPORT.                   *
```

In this system string input is normally terminated by a newline but an asterisk immediately preceding the newline causes the newline character to be stored and not recognised as the terminator. The ALGOL 68 program in the Appendix uses a similar input procedure. As one might expect, the FORTRAN string processor is not particularly efficient compared with the equivalent ALGOL 68 program.

Concluding remarks

The principle features of our model for string handling are first the variability of length of both strings and segments of strings and second the addressability of segments of strings. The importance of variability of length of a string segment under string operations is debatable and it is not surprising perhaps that few implementations distinguish clearly between strings and rows or vectors of characters. Most disturbing is the disparity between implementations of assignments to segments of strings. In most languages it is a simple matter to write a suite of subroutines for string processing as defined by our model but equivalent syntactic extensions would be more elegant. Of the systems considered only ALGOL 68 allows the user to define his own string operators and even with this facility we were unable to express $S(M:N)$ in a simple form suitable for use in string expressions and assignations.

Appendix 1 An example string processing program in ALGOL 68

Text Analysis

```
begin
  comment define concatenation operator + comment;
  op + = (string A, B) string:
  begin int m = upbA, n = upbB; [1:m + n]char a;
    if m > 0 then s[1:m] := A fi;
    if n > 0 then s[m + 1:m + n] := B fi;
    s
  end;
  comment long string input procedure comment;
  proc read string = string:
  begin string ss := "", s;
    while read((s,newline)); ss := ss + s; ss[upbss] = "*"
    do ss[upbss] := "←";
    ss
  end;
  comment procedure to output a long string comment;
  proc write string = (string s):
  begin for i to upb s do
    if s[i] = "←" then print(newline) else
      print(s[i]) fi
  end;
  comment procedure to search a string comment;
  proc index = (string a, b) int:
  begin int m := 0, n: = upb b − 1;
    if upb b > 0 then
      for k to upb a − n while m = 0 do
      if b = a[k:k + n] then m := k fi
    fi;
    m
  end;
  string s;
  int ns := 0, nw := 0, nc := 0, lastsp := 0;
  s := read string; write string (s); print((newline, newline));
  for i to upbs do
  begin
    case index("‸ . , ; − ←", s[i]) + 1
    in goto L10, goto L12, goto L11, goto L20,
       goto L20, goto L20, goto L12
    esac;
  L10; nc plus 1; lastsp := 1; goto L20;
  L11: ns plus 1;
  L12: if lastsp ≠ 0 then
     lastsp := 0; nw plus 1 fi;
  L20: skip
  end;
  print
  (("NUMBER OF SENTENCES =",ns,newline,newline,
    "NUMBER OF WORDS      =",nw,newline,newline,
    "NUMBER OF SYMBOLS    =",nc,newline,newline,
    "AV. NUMBER OF WORDS/SENTENCE =",
                             nw/ns,newline,newline,
    "AV. NUMBER OF SYMBOLS/WORD   =",
                                   nc/nw,newline))
end
finish
```

References

BARNETT, M. P. (1969). *Computer Programming in English*, Harcourt Brace and World.

BULL, G. M., and FREEMAN, W. (1971). *BASIC—a preliminary specification*, The Hatfield Polytechnic, Department of Computer Science, Technical Memorandum No. 1.

CURRIE, I. F. (1970). *Working Description of ALGOL 68-R*, Royal Radar Establishment, Malvern, Memorandum No. 2660.

ELSON, M. (1973). *Concepts of Programming Languages*, Science Research Associates.

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

GRISWOLD, R. E., POAGE, J. F., and POLONSKY, I. P. (1969). *The SNOBOL-4 Programming Language*, Prentice-Hall.

Hewlett Packard (1970). 2000*B*: *A Guide to Time Shared Basic*, Hewlett Packard Software Publication HP 02000-90010.

Housden, R. J. W., and Kujawa, R. T. (1971). EASNAP—an on-line system for Arts students, *The Computer Bulletin*, Vol. 15, No. 8, pp. 295-299.

Lindsey, C. H., and van der Meulen, S. G. (1971). *An informal introduction to ALGOL* 68, North Holland.

Macleod, I. A. (1970). SP/I—A FORTRAN integrated string processor, *The Computer Journal*, Vol. 13, No. 1, pp. 255-260.

Milner, R. (1968). String handling in ALGOL, *The Computer Journal*, Vol. 10, No. 4, pp. 321-324.

Sammet, J. E. (1969). *Programming Languages*: *History and Fundamentals*, Prentice-Hall.

Van Wijngaarden, A. (Ed.) *et al.* (1969). Report on the algorithmic language ALGOL 68. *Numerische Mathematik*, Vol. 14, pp. 79-218.

Woodward, P. M., and Bond, S. G. (1972). *ALGOL 68-R Users Guide*, London: HMSO.

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

Books received

We give below a list of new books received recently from publishers. Reviews of many of these may be expected to appear in future issues of *The Computer Journal* or *Computer Bulletin.*

A Technical Index of Interactive Information Systems, National Bureau of Standards technical note 819

Cost-Benefit Analysis of Computer Graphics Systems, by Ira W. Cotton, 1974; 47 pages National Bureau of Standards technical note 826

Computer Representation and Manipulation of Chemical Information, edited by W. T. Wipke, S. R. Heller, R. J. Feldman and E. Hyde, 1974; 328 pages, John Wiley £8·00

Computer-based information services in science and technology—principles and techniques, by M. F. Lynch, 1974; 96 pages, Peter Peregrinus Limited £3·65

Design Methods for Digital Systems, by J. Chinal, translated by A. Preston and A. Sumner, 1973; 506 pages, Springer-Verlag US$36.10

Introduction to Switching Theory and Logical Design, by F. J. Hill and G. R. Peterson, second edition, 1974; 596 pages, John Wiley £8·50

Applied Finite Mathematics, by H. Anton and B. Kolman, 1974; 475 pages, Academic Press Inc US$10.95

Numerical Methods, by G. Dahlquist and A. Bjorck translated by N. Anderson, 1974; 573 pages, Prentice-Hall

Decisions, Strategies and New Ventures, by W. G. Byrnes and B. K. Chesterton, 1973; 195 pages, George Allen and Unwin Limited £4·25

Digital Electronic Circuits and Systems, by Noel M. Morris, 1974; 143 pages, Macmillan £2·25