# String handling in ALGOL

*By* R. Milner*

**DASH (Dynamic ALGOL String Handling) is a set of procedures designed to extend ALGOL to the expression of non-numerical or partly non-numerical algorithms for which it is normally unsuited.**

## 1. Introduction

DASH is designed to allow a user to handle strings and perform efficient arithmetic in the same language. There are a number of languages designed to handle strings (Farber, Griswold and Polonsky, 1964 and 1966; Guzman and McIntyre 1966), but their arithmetic facilities are slight. On the other hand, on many computers the only "general purpose" languages at present implemented (e.g. ALGOL 60, FORTRAN) are almost purely arithmetic. This hampers both users and those whose job is computer science education.

The procedures to be described aim to provide string processing of the SNOBOL type (Farber *et al.*, 1966) to the extent that is reasonably possible within ALGOL. The description is informal, and some details are omitted for brevity. The reader will benefit from a knowledge of SNOBOL, though it is hoped an understanding can be gained without this knowledge.

The superimposition of procedures of this type on ALGOL will be rendered easier and more natural with a facility such as the proposed record handling of Wirth and Hoare (1966).

## 2. String variables, constants and expressions

As implemented on the I.C.T. 1905, DASH processes strings on the alphabet consisting of letters (upper and lower case), digits, space and

: ; < = > ? ! " # £ % & ' ( ) * + , — . / @ [ ]

Strings are represented by **reals**; a **real** variable becomes a string variable after appropriate use of the system procedure *make* or *makearray* at the beginning of the body of the block in which it is declared. For example, in the following block

**begin real** *A*,*B*,*C*,*D*,*E*; **array** *S*[0 : 100]; *make* (*B*,3); *makearray* (*S*,101); . . . ; *destroy* **end**

the variables *B*,*C*,*D* and all elements of *S* are used as string variables. They are assigned null initial values by *make* and *makearray*. The procedure *destroy* must appear at the end of any block with string variables local to it.

String constants are represented by the **real** procedure *str* applied to ordinary ALGOL strings; the effect of *str* is to convert the string storage to chained form.

Henceforward we refer to DASH strings as "strings", and the conventional ALGOL strings as "ALGOL strings".

String expressions may be built from string (and other ALGOL) constants and variables, using the DASH procedures described in **Table 1**.

For example, if the string variable *X* has value '23.0*A*' then the expressions

*tfrs*(*tfsr*(*X*) + *tfsr*(*str*('—14.14')),2,1)
*field*(*cc*(*X*,*cc*(*str*('*BCD*'),*str*('*EFG*'))),5,9)

have respective string values '⊔⊔8.9' and '*ABCDE*'.

**Table 1**

| PROCEDURE IDENTIFIER AND PARAMETERS | PARAMETER TYPES | RESULT TYPE | RESULT |
|---|---|---|---|
| *cc*(*U*1, *U*2) | *string*, *string* | *string* | The concatenation of *U*1, *U*2 |
| *field* (*U*1, *i*, *j*) | *string*, **integer**, **integer** | *string* | The *i*th to *j*th characters inclusive of *U*1 |
| *tfsr* (*U*1) | *string* | **real** | The **real** value of the real number represented by an initial substring of *U*1 |
| *tfrs* (*x*,*m*,*n*) | **real**, **integer**, **integer** | *string* | The string representing the **real** value of *x*; *m*, *n* are format descriptors |
| *lgth* (*U*1) | *string* | **integer** | The number of characters in *U*1 |
| *eq* (*U*1, *U*2) | *string*, *string* | **boolean** | **true** if *U*1, *U*2 are identical strings, otherwise **false** |

* *The City University, St. John Street, London, E.C.*1.

## 3. Input/output procedures

The procedure *stringout* (*U*1) outputs the value of *U*1 to the currently selected channel. The value of *stringin* (*U*1,*j*) is the characters of currently selected input stream up to and excluding the appearance of the first character of *U*1, or *j* characters from the input stream, which ever is less. A typical use would be

*stringin* (*str*(⌜;⌝),120)

## 4. Simple string-processing procedures

Most of the system procedures still to be described alter the value of at least one of their string parameters; such a parameter is therefore restricted to being a string variable, not a constant or expression. We will also divide string variables into two classes, "actuals" and "shadows" (see Section 6), and some parameters are restricted to one or other of these classes of variable. We will use the following notation for parameters;

| | |
|---|---|
| Integers | *i*, *j*. |
| Any string expression | *U*1, *U*2, . . . |
| String variables only | *M*1,*M*2, . . . |
| Actuals only | *A*1,*A*2, . . . |
| Shadows only | *S*1,*S*2, . . . |

**Table 2** describes some procedures for performing routine tasks.

**Table 2**

| PROCEDURE IDENTIFIER AND PARAMETERS | EFFECT |
|---|---|
| *set* (*M*1, *U*1) | Assign to *M*1 the value of *U*1. |
| *append* (*A*1, *U*1) | Append to *A*1 the value of *U*1. |
| *prefix* (*A*1,*U*1) | Prefix to *A*1 the value of *U*1. |
| *setfield* (*M*1, *U*1, *i*, *j*) | Equivalent to set (*M*1, *field* (*U*1,*i*,*j*)), but see Section 6. |
| *behead* (*M*1,*j*) | Delete first *j* characters of *M*1. |

Note that one must write *set* (*M*1,*U*1) and not *M*1 : = *U*1.

## 5. Pattern-matching

It is often required to determine whether a string, or a substring of it, satisfies a certain property, or consists of a sequence of substrings satisfying a given sequence of properties in order. In SNOBOL, and other languages, it is possible to specify a complex pattern (or property sequence) in a single statement; in ALGOL this cannot be easily done, due to the restriction that any procedure has a fixed number of arguments. In DASH therefore we match the pattern elements one at a time. There are two main procedures provided for this purpose. The first is *remove* (*M*1,*U*1,*M*2), a **boolean** procedure which scans *M*1 for a substring equal to *U*1. If found, then the preceding part of *M*1 is assigned to *M*2, the succeeding part to *M*1, and **true** is returned, otherwise *M*1,*M*2 are unaltered and **false** is returned.

As an example, if *X* has the value ⌜*EASING*⊔*THE*⊔ *SPRING*⌝ then

*remove* (*X*, *str* (⌜⊔*THE*⊔⌝), *Y*)

will return **true**, leaving ⌜*SPRING*⌝ in *X* and ⌜*EASING*⌝ in *Y*. After the success of the match, *X* or *Y* may be the object of further matching tests.

The second pattern-matching procedure is *matchpred* (*M*1,*P*,*M*3,*M*2), a **boolean** procedure which scans *M*1 for a substring for which the predicate *P* (a **boolean** procedure) is **true**, and if found, assigns this substring to *M*3, and the preceding and succeeding parts of *M*1 to *M*2, *M*1 respectively, and returns **true**, otherwise **false** is returned and *M*1,*M*2,*M*3 are unaltered. The actual procedure parameter replacing *P* is called a "string-resolving predicate" and must be written in a particular way, which can best be explained by a brief description of how *matchpred* works. After *matchpred* is entered, it hands a string *MM*, initially equal to *M*1, to *P* for an answer to the question "does *MM* have an initial substring with the required property?" If the answer is "no", *matchpred* removes one character from the head of *MM* and repeats the question; the process continues until either "yes" is returned by *P* or *MM* is null.

A string-resolving predicate therefore has two parameters: the string *MM* which it is to examine and an **integer** variable *j* to which it assigns the length of the initial substring of *MM* which has the required property. There is one system procedure of this type, *num*(*M*1,*j*), which returns **true** if and only if an initial substring of *M*1 is a number; to *j* is assigned the length of the longest such substring.

Thus, if *X* has value ⌜*AB*⊔0.12*BC*⌝ then

*matchpred* (*X*, *num*,*number*,*head*)

will return **true**, and leave ⌜*BC*⌝, ⌜⊔0.12⌝, ⌜*AB*⌝ in *X*, *number* and *head* respectively.

As an example of a user-defined string-resolving predicate, suppose we require to find a substring of *X* (if any) which is of length 3 and is immediately followed by itself in *X*. The SNOBOL pattern-match would be

*X* **A*/"3"* *A*

which, if successful, would assign the required substring to *A*.

In DASH, if we have the procedure declaration

```
boolean procedure rpt 3 (M,j); real M; integer j;
begin rpt 3 := eq(field(M,1,3),field(M,4,6)); j := 3 end;
```

Then the **boolean** expression

*matchpred* (*X*,*rpt*3,*A*,*Junk*)

will have the appropriate truth value and side effects, though *X* will have been mutilated (beheaded, in fact). A copy of *X* may be made before pattern-matching begins if it is needed, but this may be embarrassing in storage, and shadow strings (described below) give a means of preservation during pattern-matching without copying.

The **boolean** procedure *replace* ($A1,U1,U2$) scans $A1$ for an appearance of $U1$, as in *remove*, and if found returns **true** and replaces $U1$ by $U2$ in $A1$.

All scanning in DASH is performed left to right. During a scan, it may be required to ignore any part of the string enclosed between two given characters, often '(' and ')'. For example, if $X$ has the value '$(P + Q)+R$' then we may want

*remove* ($X$,*str*('+'),$Y$)

to leave '$(P+Q)$' and '$R$' in $Y$, $X$ respectively; that is, the pattern-matching process must ignore anything between '(' and ')'. This selective mode of scanning with respect to any given pair of characters $\alpha$, $\beta$ (not necessarily parentheses) is adopted after the procedure

*balance* ($U1$)

has been obeyed, where $\alpha$, $\beta$ are the first two characters of $U1$. Thus in the above case, the statement *balance* (*str*('()')) should have been obeyed. Normal scanning is resumed after obeying the parameterless procedure *unbalance*.

## 6. Shadow strings

Hitherto all string variables were considered to have "actual" status, i.e. each variable has a value which is independent of the values of other variables. Any string variable may be given "shadow" status by use of the procedure *shadow* ($M1$); it may then be used only to designate a substring of an actual string variable, called its "parent". A shadow variable may be processed in three ways:

(1) It may be assigned to a parent, and made to designate all or part of the parent, e.g. by appearing as the first parameter of *set* or *setfield*.
(2) It may be truncated at its left end by appearing as the first parameter of *behead*, *remove*, or *matchpred*.
(3) It may be moved from left to right over its parent by the procedure *step* ($S1,i,j$), which moves its left, right ends, $i,j$ places respectively (strictly, only $i$ is required to be non-negative).

The simplest use of shadows is in analysing the structure of a string with the minimum of copying. Thus, to determine whether the value of $X$ is of the form '$u\ A\ v\ B\ u$' where $u, v$ are arbitrary strings, we may write

```
set (S1,X);
if remove (S1,str('A'),S2) then
  begin if remove (S1,str('B'),S3) then
  begin if eq (S1,S2) then goto yes end end;
goto no;
```

and if $S1,S2,S3$ all have shadow status and the test is successful they will all be assigned to $X$ as parent and will designate different parts of $X$. Moreover, no copying of strings will have taken place.

A shadow is rendered meaningless (until reassigned to a parent) as soon as that part of its parent which it designates is altered. However, a shadow may be used as a means of altering its parent: the procedure *change* ($A1,S1,U1$) assumes that $A1$ is the parent of $S1$, replaces the designated part of $A1$ by the value of $U1$, and makes $S1$ designate this new substring of $A1$.

An important restriction is that a shadow may not designate part of the value of a string expression which is not a string variable; *set*($S1$,*field*($M1$,1,5)) is inadmissible and will give rise to an error message. On the other hand, *setfield*($S1$, $M1$,1,5) is admissible.

## 7. User-defined string-handling procedures

The user may freely construct procedures with string parameters and results. It is simplest to call string parameters by name always; if the result is a string, it must be assigned to the procedure identifier after applying the procedure *result*($U1$). As an example, *cc* could be defined in terms of *append* as follows:

```
real procedure cc(U1,U2); real U1,U2;
    begin real B; make (B,1); set (B,U1); append
    (B,U2);
        cc := result (B); destroy end;
```

Another example appears in Section 9.

## 8. Indirect addressing

It is convenient (and often necessary) to be able to refer to a string variable indirectly through a string expression whose value is its name. But ALGOL identifiers are forgotten at run time, and hence not eligible as names for this purpose. DASH allows the assignment of a name to any string variable by the procedure *givename* ($M1$, $U1$); at any subsequent time the procedure *ref* ($U1$) will address the variable $M1$ indirectly. For example, after

*givename* ($X$,*str*('$ABC$'))

the expression

*ref*(*cc*(*str*('$A$'),*str*('$BC$')))

can occur anywhere that $X$ can occur, with exactly the same effect. Finally the procedure *nameof* ($M1$) yields as result the string which is the name of $M1$.

## 9. Examples

Programs in DASH have been written for several types of problem, including radix sorting, manipulation and differentiation of algebraic expressions, and table-driven syntax analysis. Two short examples of procedures are given below; neither illustrates indirect addressing, but the second illustrates the use of an array of strings, a feature not present in pure string handling languages. Both procedures have worked, but are not necessarily the best that could have been written.

*Example* 1

```
real procedure mult (e,ee); real e,ee;
comment e,ee are assumed to designate unparenthesized
expressions made up of identifiers and the operators *
and + , represented by the global string variables times
and plus.  The expressions are assumed to terminate
with + .  The result is the algebraic product in the
same form.   For   example   ‘A + B + `   and
‘X*Y*Z + CAT +`   will   yield   ‘A*X*Y*Z +
A*CAT + B*X*Y*Z + B*CAT +`;
begin real a,aa,s,ss,prod ; make(a,5) ; set(a,e);
  next: if ¬ remove (a,plus,s) then goto done;set(aa,ee);
  again: if ¬ remove (aa,plus,ss) then goto next;
  append (prod, cc(s,cc (times cc(ss, plus)))); goto again;
  done: mult := result (prod); destroy end;
```

*Example* 2

```
procedure printroutes(s,m,n); value m,n; real s; integer
m,n; comment the procedure, which is recursive,
enumerates and prints all the routes from one given
node to another in a directed network assumed to be
without loops (if there are loops it will not terminate).
The network is assumed to be stored in a global string
array a : nodes are denoted by integers, and the value
of a[i] is a string in which the nodes immediately
accessible from node i are listed—for example, if there
are connections from node 1 to nodes 4,5 and 27 then
the value of a [i] is ‘4,5,27,`.  All node numbers are
considered to be < 1000.  The string parameter s
contains a list of the nodes traversed so far from the
start node: node m is one of those immediately accessible
from the last node in s, and node n is the goal.  The
statement
                  print routes(null,j,k)
will print all the routes starting at node j and ter-
minating at node k;
begin real sofar, nextnode, steplist; integer mm;
                  make(sofar,3); shadow(steplist);
  set(sofar, cc(s,tfrs(m,3,0))); set(steplist,a[m]);
  again: if remove(steplist,str(‘,`), next node) then
    begin mm := tfsr(next node); if mm ≠ n then print-
    routes(sofar, mm,n) else
      begin newline(1); stringout(sofar); print(n,3,0) end;
      goto again
    end; destroy
  end of printroutes ;
```

## 10. Operation

The user's program is in the form of a block, which has to be embedded in the system block containing the procedure declarations. The system block is held on magnetic tape, and the user prefaces his block on paper tape with a steering line which causes its insertion into the system block at compile time. He may also specify the amount of string storage he requires.

Several run-time error messages are provided, mostly concerned with inadmissible combinations of string parameters.

## 11. Implementation

The implementation is largely machine independent. In fact the only machine dependence is embodied in a set of about a dozen simple procedures written in PLAN (the I.C.T. 1900 Assembly Language) totalling less than 150 machine-code instructions. Two or three man-weeks should be ample to implement the system on any other machine with an ALGOL compiler. The currently operating version falls short of the foregoing description in a few minor respects, the most important of which is a different, slightly less convenient, indirect addressing system. Since this is temporary, a description of it would not be of interest.

## 12. Conclusion

It appears that DASH, while lacking some of the conciseness of SNOBOL, nevertheless extends ALGOL to the expression of algorithms, in more than one subject area, for which it is normally unsuited. This unsuitability arises not from the structure of the language but from the paucity of data types.

## References

Farber, D. J., Griswold, R. E., and Polonsky, I. P. (1964). SNOBOL, A String Manipulation Language, *J. Assoc. Comp. Mach.*, Vol. 11, p. 21.

Farber, D. J., Griswold, R. E., and Polonsky, I. P. (1966). The SNOBOL 3 Programming Language, *Bell System Technical Journal*, July–August 1966, p. 895.

Guzman, A., and McIntosh, H. V. (1966). CONVERT, *Comm. Assoc. Comp. Mach.*, Vol. 9, p. 604.

Wirth, N., and Hoare, C. A. R. (1966). A contribution to the development of ALGOL, *Comm. Assoc. Comp. Mach.*, Vol. 9, p. 413.