String Handling in ALGOL

Citation for published version:
https://doi.org/10.1093/comjnl/10.4.321

Digital Object Identifier (DOI):
10.1093/comjnl/10.4.321

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
The Computer Journal

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and/or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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 ‘23.04’ 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 ‘8.91’ and ‘ABCDE’.

<table>
<thead>
<tr>
<th>PROCEDURE IDENTIFIER AND PARAMETERS</th>
<th>PARAMETER TYPES</th>
<th>RESULT TYPE</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc(U1, U2)</td>
<td>string, string</td>
<td>string</td>
<td>The concatenation of U1, U2</td>
</tr>
<tr>
<td>field(U1, i, j)</td>
<td>string, integer, integer</td>
<td>string</td>
<td>The ith to jth characters inclusive of U1</td>
</tr>
<tr>
<td>tfrs(U1)</td>
<td>string</td>
<td>string</td>
<td>The real value of the real number represented by an initial substring of U1</td>
</tr>
<tr>
<td>tfrs(x, m, n)</td>
<td>real, integer, integer</td>
<td>real</td>
<td>The string representing the real value of x; m, n are format descriptors</td>
</tr>
<tr>
<td>lgth(U1)</td>
<td>string</td>
<td>integer</td>
<td>The number of characters in U1</td>
</tr>
<tr>
<td>eq(U1, U2)</td>
<td>string, string</td>
<td>boolean</td>
<td>true if U1, U2 are identical strings, otherwise false</td>
</tr>
</tbody>
</table>

---

* The City University, St. John Street, London, E.C.1.
3. Input/output procedures

The procedure `stringout (U1)` outputs the value of `U1` to the currently selected channel. The value of `stringin (U1,j)` is the characters of currently selected input stream up to and excluding the appearance of the first character of `U1`, or `j` characters from the input stream, which ever is less. A typical use would be

\[
\text{stringin (str(';'),120)}
\]

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: `U1, U2, ...`
- String variables only: `M1, M2, ...`
- Actuals only: `A1, A2, ...`
- Shadows only: `S1, S2, ...`

Table 2 describes some procedures for performing routine tasks.

<table>
<thead>
<tr>
<th>PROCEDURE IDENTIFIER AND PARAMETERS</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set (M1, U1)</code></td>
<td>Assign to <code>M1</code> the value of <code>U1</code>.</td>
</tr>
<tr>
<td><code>append (A1, U1)</code></td>
<td>Append to <code>A1</code> the value of <code>U1</code>.</td>
</tr>
<tr>
<td><code>prefix (A1, U1)</code></td>
<td>Prefix to <code>A1</code> the value of <code>U1</code>.</td>
</tr>
<tr>
<td><code>setfield (M1, U1, i, j)</code></td>
<td>Equivalent to <code>set (M1, field (U1, i, j))</code>, but see Section 6.</td>
</tr>
<tr>
<td><code>behead (M1, j)</code></td>
<td>Delete first <code>j</code> characters of <code>M1</code>.</td>
</tr>
</tbody>
</table>

Note that one must write `set (M1,U1)` and not `M1 := U1`.

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 (M1,U1,M2)`, a boolean procedure which scans `M1` for a substring equal to `U1`. If found, then the preceding part of `M1` is assigned to `M2`, the succeeding part to `M1`, and `true` is returned, otherwise `M1,M2` are unaltered and `false` is returned.

As an example, if `X` has the value `'EASING\l\lTHE\l\lSPRING'` then

\[
\text{remove (X, str ('\l\lTHE\l\l'), Y)}
\]

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 (M1,P,M3,M2)`, a boolean procedure which scans `M1` for a substring for which the predicate `P` (a boolean procedure) is `true`, and if found, assigns this substring to `M3`, and the preceding and succeeding parts of `M1` to `M2, M1` respectively, and returns `true`, otherwise `false` is returned and `M1,M2,M3` 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 `M1`, 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(M1,j)`, which returns `true` if and only if an initial substring of `M1` is a number; to `j` is assigned the length of the longest such substring.

Thus, if `X` has value `'AB\l\l0.12BC'` then

\[
\text{matchpred (X, num, number, head)}
\]

will return `true`, and leave `'BC'`, `\l\l0.12'`, `'AB'` in `X`, `number` and `head` respectively.

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

\[
X *A\l\l3'\l\lA
\]

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

In DASH, if we have the procedure declaration

```dasm
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

\[
\text{matchpred (X,rpt3,A,Junk)}
\]

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 \((A_1,U_1,U_2)\) scans \(A_1\) for an appearance of \(U_1\), as in remove, and if found returns true and replaces \(U_1\) by \(U_2\) in \(A_1\).

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

\[
\text{remove} \ (X, str('+'), Y)
\]

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

\[
\text{balance} \ (U_1)
\]

has been obeyed, where \(\alpha, \beta\) are the first two characters of \(U_1\). Thus in the above case, the statement balance \((str('0'))\) 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 \((M_1)\); 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 selffield.

(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 \((S_1,i,j)\), 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 \(uA\nu B\) where \(u, \nu\) are arbitrary strings, we may write

\[
\text{set} \ (S_1,X);
\]

\[
\text{if remove} \ (S_1, str('A'), S_2) \text{ then}
\]

\[
\begin{align*}
&\text{begin if remove} \ (S_1, str('B'), S_3) \text{ then} \\
&\quad \text{begin if eq} \ (S_1, S_2) \text{ then goto yes end end;}
\end{align*}
\]

\[
\text{goto no;}
\]

and if \(S_1, S_2, S_3\) 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 \((A_1,S_1,U_1)\) assumes that \(A_1\) is the parent of \(S_1\), replaces the designated part of \(A_1\) by the value of \(U_1\), and makes \(S_1\) designate this new substring of \(A_1\).

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\((S_1, field\((M_1,1,5)\))\) is inadmissible and will give rise to an error message. On the other hand, setfield\((S_1, M_1, 1, 5)\) 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\((U_1)\). As an example, \(cc\) could be defined in terms of append as follows:

\[
\text{real procedure} \ cc(U_1,U_2); \ \text{real} \ U_1,U_2;
\]

\[
\begin{align*}
&\text{begin real} \ B; \ \text{make} \ (B,1); \ \text{set} \ (B,U_1); \ \text{append} \\
&\quad (B,U_2); \\
&\quad cc := \text{result} \ (B); \ \text{destroy end;}
\end{align*}
\]

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 givenname \((M_1,U_1)\); at any subsequent time the procedure \ref\((U_1)\) will address the variable \(M_1\) indirectly. For example, after

\[
\text{givenname} \ (X, \text{str('ABC'))}
\]

the expression

\[
\text{ref} (cc(\text{str('A')}, \text{str('BC'))})
\]

can occur anywhere that \(X\) can occur, with exactly the same effect. Finally the procedure nameof \((M_1)\) yields as result the string which is the name of \(M_1\).

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 —i remove (a,plus,s) then goto done;set(aa,ee);
again: if —i remove (aa,plus,ss) then goto next;
append (prod,  cc(s,cc (times cc(ss, plus))));%oto 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[1] 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 terminating at node k;
begin realsofar, nextnode, steplist; integer mm;
make(sofar,3); shadow(steplist);
set(sofar, cc(s,tfors(m,3,0))); set(steplist,a[m]);
again: if remove(steplist,str(’ ’), next node) then

References