

Strings in PL/I

Robert F. Rosin

Introduction

The accompanying study proposal and the appended comments have resulted from many months of thought and speculation about the string manipulation facilities in PL/I. To be sure, the existing language tools are complete in the sense that one can program any reasonable scan, replacement, extraction, etc. . However, it has been considered by some that a more natural set of string operations should be considered for this language so that reasonable applications could be approached in a straightforward way.

Much of the flavor of this scheme will appear to be reminiscent of SNOBOL, and not without cause. SNOBOL has been used with great satisfaction by the author and his students. It is considered by many to be most successful of all attempts to provide a special purpose programming tool built around the character string data type.

On the other hand, every effort has been made to preserve the style and spirit of PL/I. Indeed, the one concept which might tend to cause some consternation is the generality with which the left-hand-side expression, or pseudo- variable, has been used. (This becomes a very powerful tool when freed from the bounds of functional notation.) But this proposal does not include non-PL/I concepts, such as the arbitrary use of assignment within expressions, which does predominate in SNOBOL.

It is hoped that by presenting this material that interested parties will consider it and respond. Much of the facility described here has been implemented as a set of defined operators and a new data type in MAD at Yale. We hope to use this extended language tool to get a better feel for the suitability of this proposal, to test alternative definitions, and to better understand its strengths and weaknesses,

Special problems associated with BIT as opposed to CHARACTER have not been considered in depth, but no difficulties have appeared so far.

Study Proposal for PL/I String Facilities

1. The default attribute for strings should be VARYING, with the option to declare FIXED. The maximum length, if specified, would serve as a guide to the compiler, but is optional.
2. SUBSTR(X(A),I,J) should be replaced by X(A:I...J+I-1) or some similar notation.

It follows that:

$X(A:I) \equiv X(A:I \dots I)$

$X(:I) \equiv X(:I \dots I)$

$X(A \dots J) \equiv X(A:1 \dots J)$

$X(A:I \dots) \equiv X(A:I \dots \text{LENGTH}(S(A)))$

3. Five new operators should be defined. Their purpose is to scan strings, left-to-right, and return a string temporary.
 $X \text{ UPTO } Y$ returns $X(1 \dots \text{INDEX}(X, Y) + \text{LENGTH}(Y) - 1)$
 $X \text{ BEFORE } Y$ returns $X(1 \dots \text{INDEX}(X, Y) - 1)$
 $X \text{ AFTER } Y$ returns $X(\text{INDEX}(X, Y) + \text{LENGTH}(Y) - 1 \dots)$
 $X \text{ FROM } Y$ returns $X(\text{INDEX}(X, Y) \dots)$
 $Y \text{ IN } X$ returns $X(\text{INDEX}(X, Y) \dots \text{INDEX}(X, Y) + \text{LENGTH}(Y) - 1) \equiv Y$

In any of these operations, if Y does not occur in X , then the scan is said to fail, and the value of the resultant string temporary indicates this. (See 5 and 6)

4. The relative precedence of these new operators is:
 arithmetic
 concatenation (moved from its current position)
 UPTO FROM AFTER BEFORE IN
 relationals
 replacement
5. The notation in 2 and 3 above can be used on the left-hand-side of assignment statements in what might be called pseudoexpressions, which can be considered in the same class as pseudovariables. Expressions used in this way are not substantially different from using ordinary subscription on the left-hand-side. For example:
 $X = \text{'THIS' } \square \text{'BELIEVE' } \square \text{'IS' } \square \text{'TRUE'}$;
 $X \text{ AFTER ',' UPTO ',' } = \text{' '}$;
 results in X containing
 $\text{'THIS' } \square \square \text{'IS TRUE'}$
6. Replacement is conditioned upon lack of failure in all expressions in a statement. For example:
 $Z = \text{'XYZ'}$; $\text{'A' IN } Z = \text{'MN'}$;
 results in Z containing 'XYZ' , and the statement is said to fail.
7. Replacement of pseudoexpressions in varying strings causes the designated string to be adjusted (expanded or contracted) to contain the expression on the right-hand-side. For example:
 $X = \text{'ABCDE'}$; $X \text{ UPTO 'B' } = \text{'XYZ'}$;
 results in X containing 'XYZCDE' , and
 $Q = \text{'PUT' } \square \text{'OUT' } \square \text{'THE' } \square \text{'CAT'}$; $\square \text{'IN } Q = \text{' '}$
 results in Q containing 'PUTOUT THE CAT' .
8. Two additional built-in generic functions should be provided: SUC and FAIL. When used without arguments they return a truth value (bit string of length 1) indicating the success or failure of the most recent attempt at string formation, and the indicator is reset to success. When used with a string expression as an argument, the corresponding truth value is returned. Examples:

```
X = 'PUT□OUT□THE□CAT';
DO WHILE(SUC); '□'IN X ="; END;
results in X containing 'PUTOUTTHECAT', and
DIGIT = '1234567890'; ST = 'AB123D'
DO I = 1 BY 1 WHILE(FAIL(ST(:I) IN DIGIT)); END;
results in I containing 3.
```

1. There should also be a pseudovvariable SUC which could be used to set the success-fail value.
2. The occurrence of an array as the criterion in a scan results in a left to right scan which is terminated by the first successful match of any element in the array. The longest array element is selected if two or more satisfy that criterion. (The inclusion of array constants in PL/I would greatly enhance this function.)
Example:
DCL DIGIT (10) CHAR(1) INITIAL ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
X = 'ABC123DEF'; A = X BEFORE DIGIT;
results in A containing 'ABC'.
3. Success and failure are local to the block in which string formation takes place, although they can be passed between procedures as attributes of string arguments and results.
4. All function calls in an expression are evaluated, whether or not any previously computed subexpression in the statement has failed.

Discussion

1. It is possible that the notation shown for the five new operators is ambiguous, at least in a pragmatic sense if not a syntactic sense. This arises since they could be confused with identifiers, and PL/I is premised on a philosophy of no reserved words. One viable alternative would be to use an escape character in the prefix position providing that this notation is totally unambiguous. One possibility would be "UPTO "AFTER etc.
This would also fit nicely into a scheme for abbreviation such as "U "A etc.
2. It has been pointed out (by M. D. McIlroy) that the generality implied by the extensive use of left-hand-side expressions (pseudovvariables) leads to a definite ambiguity with respect to the equal sign (=). It is worthwhile to note that this highlights the unfortunate situations which arise when a symbol has two distinctly different meanings, supposedly always delineated by context. Consider the example:
DCL A BIT(10), B BIT(1); A BEFORE B = '1'B
An interim solution, which is far from adequate, would be to define the first occurrence of the equal sign in an assignment statement to mean replacement, and all others to be the relational operator, as is currently the case. It is also possible to

consider this rule with the additional proviso that an equal sign in a parenthesized expression can imply only the comparison operation.

3. The meaning of replacement changes depending on the success or failure of all string operations in a statement. The definition included in the proposal says that failure implies no replacement. An alternative would be to have failure imply replacement by the null string, although exactly what should be replaced is not clear in every case. SNOBOL uses the definition in the proposal and its success leads one to support adopting that rule. Another alternative is to allow the programmer to change the mode of replacement by calling a built-in function.
4. It is interesting to consider the addition of an indirectness operator. This has been provided for in the MAD facility and will be explored.

One might imagine the following sequence of statements using the indirectness operator IND:

```
X = 'ABC';  
Y = 'X'.  
Z = IND Y;
```

This results in Z containing 'ABC'.

5. A similarity can be noted between the operators introduced in this proposal and the traditional relational operators.

```
>      AFTER  
>=     FROM  
<      BEFORE  
<=     UPTO  
=       IN  
(¬=    NOTIN)
```

It is important to note that the string operators return general string values, whereas the relationals return only bit strings of length 1.. The value of NOTIN is, if anything, always the empty string: the one situation in which it might appear useful is resolved by using \neg in conjunction with SUC. For example:
IF \neg SUC(X IN Y) THEN...

Brief Examples

```
/* THIS ROUTINE SCANS THE STRING T FOR THE FIRST SUBSTRING  
OCCURRING BETWEEN A PAIR OF BLANKS AND ASSIGNS THIS VALUE  
TO X. IF NO SUCH STRING EXISTS THEN CONTROL IS TRANSFERRED  
TO Y. THE PORTION OF T UPTO AND INCLUDING X IS THEN  
DELETED. */
```

```
/* EXISTING NOTATION */  
II INDEX (T, ' ');  
IF I1 = 0 THEN GO TO Y;  
T1 = SUBSTR (T, I1+1);  
I2 = INDEX (T1, ' ');  
IF I2 = 0 THEN GO TO Y;  
X SUBSTR (T1, 1, I2-1);
```

```

T = SUBSTR (T, I2);

/* PROPOSED NOTATION */
X = T AFTER '□' BEFORE '□';
IF FAIL THEN GO TO Y;
T UPTO X = '□';

/* DELETING ALL BLANKS FROM STRING X */
/* EXISTING NOTATION */
LOOP: I = INDEX(X, '□'); IF I = 0 THEN GO TO DONE;
X = SUBSTR(T, 1, I-1) || SUBSTR T, I+1); GO TO LOOP;

/* PROPOSED NOTATION */
DO WHILE (SUC); '□' IN X = ''; END;

```

This paper originally appeared in *PL/I Bulletin* No. 4 (Sep 1967), 6-12.