77-10,320

HUTCHISON, Perry Charles, 1949-
EXTENSIONS TO A BLOCK-STRUCTURED
PROGRAMMING LANGUAGE TO SUPPORT
PROCESSING OF SYMBOLIC DATA AND
DYNAMIC ARRAYS.

Iowa State University, Ph.D., 1976
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106

Extensions to a block-structured programming language
to support processing of symbolic data and dynamic arrays

by

Perry Charles Hutchison

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1976

TABLE OF CONTENTS

INTRODUCTION 1

OVERVIEW OF SPL 4

EXTENDED CONTROL CONSTRUCTS 7

THE SPL MASK AND FORMAT OPERATORS 9

DEFICIENCIES OF THE SPL MASK OPERATOR 14

PROPOSED EXTENSIONS AND GENERALIZATIONS OF THE MASK OPERATOR 16

DEFICIENCIES OF THE SPL FORMAT OPERATOR 19

PROPOSED EXTENSIONS AND GENERALIZATIONS OF THE FORMAT OPERATOR. 21

APPLICATION OF OPERATORS TO STRUCTURES 24

SCALAR-STRUCTURE CONVERSION OPERATORS 26

SUBSCRIPTION 28

THE MATCH OPERATOR 30

EXAMPLES OF THE USE OF EXTENDED SPL 32

CONCLUSION 35

ACKNOWLEDGEMENTS 37

REFERENCES 38

INTRODUCTION

The SYMBOL-2R computer system[1] was designed and constructed by the Digital Systems Research group at Fairchild Camera and Instrument Corporation in the late 1960's for the purpose of re-examining a number of traditional assumptions regarding computing systems, including the functional division between hardware and software. One goal of the project was to demonstrate that the capabilities of hardware had been grossly underestimated. This demonstration was accomplished by constructing a computer system, SYMBOL-2R, which incorporates an interpreter for a very high-level programming language, and an operating system to supervise its multiprogramming/multiprocessing/demand-paging environment, entirely in the hardware. The system is capable of supporting up to 15 terminals in a time-shared environment. No software is needed to accomplish this operation.

SYMBOL-2R was not intended to be a production prototype, and therefore a number of simplifying assumptions were made in the design of the machine and of the SYMBOL Programming Language (SPL)[2] which it implements. For example, many "features" were omitted when their inclusion would not have furthered the goals of the project or demonstrated significant principles. In particular, no claim of completeness[1] has been made for SPL.

One of the goals of the SYMBOL project at Iowa State University has been to evaluate the SYMBOL-2R system and SPL. It will be our purpose here to examine SPL, identifying its deficiencies and proposing modifications and extensions to correct them.

Although SPL contains an unusually powerful string-manipulation

---

[1] In reference to a language, "complete" is difficult, if not impossible, to define satisfactorily. Since additional "features" can be added to any language definition, no language can be "complete" in the sense that "nothing useful could possibly be added." Attempts at producing languages which are complete in this sense lead to such abominations as PL/I or SNOBOL4[3]. We are inclined to consider a language "complete" if it contains no obvious omissions, but this definition does not escape subjectivity since what is obscure to one observer may be obvious to another. We shall leave to the reader the judgment as to whether the language which we propose deserves to be called complete.

operator (MASK), its facilities for testing and examining the contents of strings are limited to lexicographic comparison[2, p. 58]. SPL shares this limitation with most other general-purpose programming languages [4,5,6,7]. Such limitations become troublesome in many applications, such as compilers, interpreters, and editors, involving the processing of text. The complexity and correctness difficulties regularly encountered in present-day compilers and interpreters, particularly in the lexical-analysis sections, are at least partially chargeable to the lack of sufficiently powerful string-processing facilities in the languages used to write them.

Specialized pattern-matching languages[8,9,10] provide greatly expanded string-examination capabilities, but their control structures are typically limited to procedure-calls (which in some languages may be recursive) and GO TO statements (which usually incorporate some form of conditional). They do not provide such more recently developed constructs as nested conditionals (IF-THEN-ELSE), iteration statements (WHILE-DO, REPEAT-UNTIL, etc.), and multiple-choice conditionals ("case statements"), and their arithmetic capabilities are typically limited and inefficient. Furthermore, the pattern-matching operations themselves are generally quite complex and difficult to understand fully, and the determination of the manner in which a match "succeeded" involves dependence on side-effect assignment operations built into the "pattern". Since the specialized languages would require rather major overhaul jobs to correct their deficiencies, it is perhaps not surprising that little has been attempted in this area. It is however quite surprising how little attention has been paid to the incorporation of pattern-matching facilities into general-purpose programming languages. In one of the few publications on this subject, Balzer and Farber[11] have proposed a brute-force combination of the SNOBOL4 pattern-matcher with PL/I. They could scarcely have chosen a worse host language for such a transfusion; Dijkstra[12] has correctly pointed out that PL/I is already excessively baroque.

For the benefit of those readers who may not be familiar with SPL, we shall begin by giving a brief overview; this overview will be followed by a detailed description of certain areas, pointing out the deficiencies which have been found. We shall conclude with detailed descriptions of the modifications and extensions which we propose in the interest of remedying the deficiencies, including a description of a simple yet powerful pattern-matching operator.

Our proposed modifications and extensions are not all of equal importance. The extended control constructs WHILE and SELECT are included in the interest of completeness and because the WHILE is used in some illustrations and examples. Many of the mentioned deficiencies of MASK and FORMAT have been discovered in the course of operational experience with the SYMBOL-2R system; the correction of these deficiencies is considered to be of some significance and, in some cases, non-trivial and less than obvious. The notion of applying to aggregates operators defined upon scalars has been implemented in APL and to a lesser extent in PL/I, but the application of dyadic operators to structures of arbitrary and non-conformable shapes is believed to be new. The proposed scalar-structure conversion operators merely make available, in contexts other than I/O, transformations already contained in SPL. The redefinition of subscription to permit subscript lists of varying length addresses a problem to which we know of no previous satisfactory solution. (The variable-length subscript lists are also used in defining the MATCH operator.)

The definition of the MATCH operator is considered to be the primary contribution of the research herein reported. MATCH is intended to make available the sort of string-searching capabilities found in SNOBOL4[1] and other specialized pattern-matching languages, without introducing side-effect assignment operations and large numbers of difficult-to-remember "pattern primitives." Some examples of its use are included.

---

[1] A knowledge of SNOBOL4 is not required to understand our proposals. The reader who is unfamiliar with the SNOBOL languages should not be alarmed.

OVERVIEW OF SPL

The syntax of SPL is given in [2], together with a description of its semantics. SPL is a block-structured, ALGOL-like language having two manipulable data types called _scalars_ and _structures_. A scalar is a character-string of unlimited and dynamically-variable length; a structure is a vector containing one or more (but not more than 9999) _components_, each of which is either a scalar or a structure. Certain subsets of the scalars are recognized semantically: _numbers_ are scalars which can be interpreted as representing numerical values (see [2] for details), _Booleans_ are scalars containing only the characters 0, 1, and space, and _truth-values_ are the single-character Booleans 1 and 0 (to which are assigned the interpretations true and false, respectively). The default scope of a variable is local, i.e. if the same name is used in two different blocks the two uses reference different variables unless the name is declared GLOBAL (which extends its scope outward one level)[13].

Operators are defined upon scalar operands and produce scalar results. The arithmetic operators (addition, subtraction, multiplication, division, negation, and absolute value) require that their operands be numbers and produce results which are numbers. The JOIN operator produces as its result the concatenation of its operands. The FORMAT and MASK operators provide powerful editing capabilities for numbers, and for scalars in general, respectively; these two operators will be described in detail in a later section. The string-comparison operators BEFORE, SAME, and AFTER produce truth-values based on the lexicographic ordering of their operands; the six numeric-comparison operators produce truth-values based on the ordering of the numerical values represented by their operands (which must be numbers). The logical operators AND, OR, and NOT produce Boolean results by applying the corresponding operations of Boolean algebra to their operands (which must be Booleans) on a character-by-character basis. (Blanks in the operands are skipped and do not influence the result.)

A conventional assignment operation permits the value of a variable or of a structure component to be replaced. The right-hand side (new value) may be a variable (having either a scalar or structure value), an expression (which will always have a scalar value), or an <u>assignment-structure</u> (which has a structure value). (An assignment-structure is a linearized representation of a vector in which components are separated by <u>field-mark</u> characters and the entire vector is enclosed in <u>group-mark</u> characters. Each component of the vector may be a variable, an expression, or an assignment-structure.)

A component of a structure-valued identifier may be selected by a <u>subscripted reference</u>, in which the identifier is followed by a list of <u>subscripts</u> separated by commas and enclosed in brackets. A subscript may be a constant, a variable, or an expression; its value must be a non-negative number less than 10,000. A subscripted reference may be qualified by being preceded by the word IN, in which case the result is a truth-value designating whether the specified component exists rather than an access to the component.

A substring of a scalar-valued structure component or identifier may be selected by means of a <u>partial reference</u>, which is specified by a <u>bound-pair</u> consisting of two subscripts separated by a colon. (Syntactically, a bound-pair is handled like a single subscript.) The first subscript of the pair, which must be at least 1, specifies the starting character position in the string; the second specifies the length of the substring. Thus the SPL reference X[I:J] is equivalent to the PL/I construct SUBSTR (X, I, J) . Following PL/I a bit farther, the subscript following the colon (J in the above example) may be omitted to denote that the substring extends to the end of the original string. Unlike SUBSTR, a partial reference produces a value rather than an access, and hence cannot serve as a recipient in an assignment or INPUT statement.

Procedures, labels, and GO TO statements are handled in a conventional manner, with the restriction that procedures may not be used recursively. Procedures may be called as functions, i.e. they may return values. The equivalent of the PL/I "label array" is provided by the SWITCH statement, which creates a structure whose components may be used in GO statements.

The conditional (IF-THEN-ELSE) construct is somewhat unusual syntactically in that multiple statements are accommodated in the THEN and ELSE branches without recourse to such devices as enclosing them in a "begin-end" pair. This is accomplished by requiring that each conditional statement conclude with the word END, which serves to delimit the ELSE branch. The THEN branch is delimited by the word ELSE (or by the END if the statement has no ELSE branch). The semantics of the conditional statement are conventional.

Input and output are handled via INPUT and OUTPUT statements, which are unusual in that they provide no formatting. (The idea is that, instead of putting the formatting in the I/O and then resorting to some kind of "core-to-core I/O" facility to make it available elsewhere, generalized formatting capabilities are provided in the form of the MASK and FORMAT operators which are usable in any context.) The I/O statements do contain a STRING qualifier (which influences the manner in which structure values are transmitted) and a DATA qualifier (whose effect is similar to the FORTRAN NAMELIST or PL/I GET/PUT DATA statement), as well as TO and FROM qualifiers whose purpose is similar to FORTRAN "unit numbers" or PL/I "file names".

EXTENDED CONTROL CONSTRUCTS

SPL's set of control constructs is complete in the sense that it is sufficient to express any algorithm; however the absence of any "looping" construct requires that repetition be specified by means of the GO statement and controlled with the conditional. We shall not rehash here the plethora of arguments concerning the desirability or undesirability of GO TO statements[14,15,16], but shall simply observe that a repetition construct is a very useful thing for a programmer to have available, and a "Case Statement," while not greatly different from a series of IF-THEN-ELSE's, is generally easier to follow when the algorithm involves a choice among more than two alternatives. We therefore propose to add to SPL two additional control constructs: the WHILE-DO-END and the SELECT-WHEN-END. The WHILE-DO-END is a conventional loop; the SELECT-WHEN-END is a form of case statement. In the notation of [2], the syntax of these statements is as follows:

```
loop-stm ::= WHILE exp DO body END

case-head ::= SELECT (FIRST|EACH) CASE;

case-clause ::= WHEN exp:body

any-clause ::= WHEN ANY:body

none-clause ::= WHEN NONE:body

case-stm ::= case-head List;case-clause
             (any-clause [none-clause]|[none-clause] [any-clause])
             END
```

Additionally, the definition of "compound-stm" must be changed to:

```
compound-stm ::= conditional-stm|environment-stm|case-stm|loop-stm
```

The semantics of the WHILE-DO-END are conventional: the body of the loop is executed as long as the "exp" is true. If the exp is false when the statement is encountered, the body is not executed.

The semantics of the SELECT-WHEN-END are inspired by similar constructs in other languages[6,17]: The exp's are evaluated in the order in which they appear. Whenever an exp is "false", the next exp is evaluated. When an exp is "true", the body if its case-clause is executed; if the case-head specified EACH the next exp is evaluated, otherwise the body of the any-clause (if one exists) is executed and the statement terminates. When no more exp's remain to be evaluated, the body of the none-clause or any-clause is executed if none or at least one of the exp's produced a "true" result (provided that the appropriate clause exists).

It is our belief that the provision of more than one repetitive control construct in a general-purpose language constitutes an unnecessary complication of the language, and that there is little objective basis for selecting between the WHILE-DO and REPEAT-UNTIL forms. Our choice of the WHILE-DO form is largely arbitrary.

Our selection of what may be termed a "multiple Boolean" case statement over the more common "indexed" case (in which an expression is evaluated and, based on the value obtained, one of several succeeding statements or groups of statements is executed) is based on generality. The equivalent of the indexed case statement is readily constructed using the multiple Boolean construct by specifying case-clauses such as WHEN I=1: ... WHEN I=2: ... etc. The indexed case statement, on the other hand, does not readily lend itself to situations in which the successive tests are not restricted to various possible values of a single variable or expression. We recognize that the price of this selection is likely to be reduced implementation efficiency, since a rather sophisticated (and hence probably slow) compiler would be required to recognize those instances in which an indexed realization of a particular case statement could be used to advantage. If the compiler were not so sophisticated, the resultant evaluation of several Boolean expressions when one indexed jump would suffice would be somewhat wasteful of computational capacity. However, given the current (and presumably future) trend of ever-decreasing hardware costs and rapidly rising programming costs, we feel that the more general and hence more useful construct will render programming enough easier, faster, and more reliable to justify the cost.

THE SPL MASK AND FORMAT OPERATORS

We shall now undertake to describe in detail the SPL operators MASK and FORMAT. Each produces an edited version of its left-hand operand (to which we shall refer as the source). MASK treats its source as simply a string of characters; FORMAT is concerned with the numerical value represented by its source. Each of these operators treats its right-hand operand as a control-string which directs the editing operation.[1]

A MASK or FORMAT control-string consists of a series of control codes. In the case of MASK, these control codes are executed in sequence and the MASK operation is complete when the last control code in the string has been executed. In the case of FORMAT, the entire series of control codes, taken as a whole, forms a template onto which the source value is mapped; the mapping process will be described shortly.

A control code for either operator consists of a control character chosen from Table I or Table II as appropriate, optionally preceded by a replicator and followed (in some instances) by a qualifier. A replicator is a one- or two-digit number (indicating that the control character should be repeated that number of times), or the letter F (indicating that the control character should be repeated zero or more times until the source is exhausted). An omitted replicator is assumed to be 1.

A qualifier is a character or a series of characters which modifies or further specifies the action to be performed by the control character which it follows. Those control characters which require qualifiers are identified as such in the tables.

---

[1] R. F. Rosin (formerly with the ISU Computer Science Department) has pointed out that arrangements of this sort are in fact languages-within-languages, and Dakins[18] has defined a grammar for the SPL MASK and FORMAT control-strings. The changes to MASK and FORMAT which we shall propose may in this sense be considered as changes to these specialized editing languages rather than as changes to SPL itself.

TABLE I. MASK CONTROL CHARACTERS

Character	Replication[1]	Qualifier	Semantics
S	F,n	none	Append[2] current source character. If source is empty, append a blank.
I	F,n	none	Discard current source character.
B	F,n	none	Append a blank. If F-replicated, also discard current source character.
/	n	none	Append a carriage-return.
E	F,n	none	Append hex-unpacked current source character (2-character result).
H	F,n	none	Append character formed by hex-packing current and following source characters.
U	F,n	none	Append binary-unpacked current source character (4-character result).
P	F,n	none	Append character formed by binary-packing current and 3 following source characters.
A	F,n	one char	Append current source character unless it is the same as the qualifier.
C	F	none	Discard all remaining source characters and append 4-digit count of them. Must be F-replicated.
'	none	rest of literal	Append literal (i.e. everything between this apostrophe and the next apostrophe which doesn't have another immediately following). An apostrophe within the literal is represented as two adjacent apostrophes.

[1] "n" denotes a one- or two-digit number.

[2] Append to the result, and discard from the source.

TABLE II. FORMAT CONTROL CHARACTERS

Character	Replication[1]	Qualifier	Semantics
D	F,n	none	Put digit in result.
N	F,n	none	Put digit in result unless it is a leading zero.
Z	F,n	none	Put digit in result unless it is a leading zero, in which case put a space in result.
*	F,n	none	Put digit in result unless it is a leading zero, in which case put an asterisk in result.
I	F,n	none	Discard digit.
B	n	none	Put a blank in result.
/	n	none	Put a carriage-return in result.
C	none	none	Put a comma in result unless the preceding digit-selector selected a leading zero, in which case put in result the same character as that digit-selector.
$	1	none	Put a dollar sign ahead of the first digit in the result, following any blanks or asterisks inserted by Z or * controls ("floating" dollar sign). If used, $ must precede all control characters except B, Q, R, /, and '.
+	1	none	Put a floating + or - sign, as appropriate, in the result. Positioning rules are the same as for $; if both $ and + are used in the same template + must follow $ and the floating sign will immediately follow the $ in the result.
-	1	none	Same as +, but a blank will appear in the result in place of the + if the source is positive. + and - may not both be used in the same template.
.	1	none	Put decimal point in result. Also serves as the decimal-point alignment reference for the template.
V	1	none	Serves as the decimal-point alignment reference but puts nothing in the result. V and . may not both appear in the same template.

[1] "1" designates a character which may not be replicated and may appear only once in a template.

TABLE II. (continued)

Character	Replication	Qualifier	Semantics
$_{10}$	1	none	Causes the result to be in exponential form, and serves to separate the mantissa part of the template from the exponent part.
X	none	none	Put "EM" in the result if the source is "empirical", otherwise "EX". If used, X must follow all control characters except B, Q, R, /, and '.
M	none	none	Same as X, except that the EX is omitted.
Q	none	literal	Put literal (enclosed in apostrophes) in result if source is negative, otherwise nothing.
R	none	literal	Put literal in result if source is positive, otherwise nothing.
'	none	rest of literal	Put literal in result.

The FORMAT control characters D, N, Z, *, and I are collectively referred to as <u>digit-selectors</u>. Each occurrence of a digit-selector in a template causes one digit to be taken from the source and placed in the result. (Exception: The digit-selector I does not place anything in the result.) Except for I, the digit-selectors differ only in their treatment of "leading" zeros. (A leading zero is one which precedes the decimal point and all significant digits of the source.) D represents a leading zero as 0, Z as a blank, * as an asterisk, and N as no character at all.

Each FORMAT template is of either <u>exponential</u> or <u>non-exponential</u> form, depending on whether it does or does not contain the control code $_{10}$. The two forms are most readily understood if described separately.

In order to map a source value onto a non-exponential template, the decimal point of the value is aligned with the decimal-point reference of the template. (Unless explicitly established by the V or . control-code, the decimal-point reference is at the right-hand end of the template.) Each digit of the integer part of the source (i.e. that part left of the decimal point) is paired with a digit-selector left of the decimal-point reference; an F-replicated digit-selector will be paired with zero or more digits so as to pair the most-significant digit of the value with the left-most digit-selector. If the part of the template left of the decimal-point reference contains neither an F-replicated digit-selector nor enough digit-selectors to account for all significant digits of the integer part of the value, a processing error occurs.

Beginning at the decimal point, the digits of the fractional part of the value are paired with the digit-selectors to the right of the decimal-point reference. Here, an F-replicated digit-selector is paired with all remaining significant digits (if any exist), so any following digit-selectors can only produce zeros in the result.

The mantissa part of an exponential template is treated very similarly to the fractional part of a non-exponential template. There are no leading zeros to worry about, and so the N, Z, and * digit-selectors behave like D. The first digit-selector in the template is always paired with the most-significant digit of the value, and an F-replicated digit-selector will pair with all remaining significant digits. Floating dollar signs, however, are not permitted.

The exponent part of an exponential template is treated like a one- or two-digit non-exponential template, except that a Q or R control character will interrogate the sign of the mantissa rather than the sign of the exponent. The value of the exponent is adjusted in accordance with the position of the decimal-point reference in the mantissa part.

The result is constructed by replacing each control code (except $, +, and -) in the template with its paired digit(s) (in the case of a digit-selector), or the appropriate other character(s). Finally, the floating dollar sign and/or arithmetic sign is inserted immediately preceding the first digit.

DEFICIENCIES OF THE SPL MASK OPERATOR

The SPL MASK operator contains a number of special cases, asymmetries, omissions, and lacks of generality, including the following:

The construct "nnS" (where nn represents any 1- or 2-digit number) has the effect of left-justification in a field of width nn, with space-fill or truncation as required. No construct is provided for right-justification. It is not possible to specify a different fill character.

In general, the F replicator means "repeat the following control until the source-string is exhausted." One would expect, therefore, that the construct "FB" would result in an infinite loop (or be forbidden) since the B control does not consume any source-characters. However, "FB" has been defined as if the B did consume a source-character, i.e. "append to the result-string as many blanks as there are characters remaining in the source-string." Thus, in the case of the "FB" construct, replication has altered the semantics of the control character in addition to causing repetition. Strangely, the almost-identical construct "F/" is forbidden.

The A control has been defined as appending to the result-string the current source-character, unless it is the same as the character following the A in the control-string. It may however be useful to view the A as (equivalently) appending to the result-string either a null character or the current source-character, depending on whether the source-character does or does not match the character following the A. This second interpretation gives rise to a generalization: Append to the result-string either a specified replacement character (which may or may not be null) or the current source-character, depending on whether the source-character is or is not contained in a given set. If the replacement character is now permitted to be determined as a function of the source-character, and the given set is allowed to encompass all possible characters, the result is a general one-for-one conversion operation, similar to the PL/I "TRANSLATE" built-in function.

The ' (literal) control is the only MASK control which cannot be replicated. This is probably a concession to the hardware implementation, as replicated literals would require either that the literal be copied into some kind of temporary storage or that the control-string be "backed up" for each repetition.

Replicators are limited to two digits. This is definitely an implementation concession; it limits the size of the counter required.

The C control is indeed a pathological case. It is required to be F-replicated; the construct "FC" consumes all remaining source-characters and appends to the result-string the number of characters which it consumed (as a four-digit number). This may be another implementation concession, for if C were required to count the remaining source-characters without consuming them it would be necessary to "back up" (or copy) the source-string. It may well be questioned whether this "string-length" function belongs in MASK at all, bearing as it does virtually no relation to the other controls.

PROPOSED EXTENSIONS AND GENERALIZATIONS OF THE MASK OPERATOR

Those aspects of the SPL MASK operator which we propose to redefine are summarized below. (Table III contains the complete set of control characters for this extended MASK operator.)

A replicator may be of unlimited magnitude, and may be applied to a literal.

Any sequence of controls may be enclosed in parentheses, and a replicator may be applied to it. (We shall refer to such a parenthesized sequence as a group.) Parentheses may be nested to any depth.

The controls ← and → permit reversal of the scan of the source-string.

The F replicator may be applied to any control or group (the replicand), with the effect of repeating it as long as at least one character remains in the source string. If the replicand does not explicitly consume at least one source-character, an I control will (in effect) be appended to it.

The C control is eliminated. (Its function is served by the monadic operator LEN, described in a later section.)

The R control is added to permit right-justification. It consumes all remaining source-characters and right-justifies them in a field whose width is equal to the value of its replicator. (Note that "FR", "FL", and "FS" are equivalent.)

The L control is added for mnemonic consistency with R; it is equivalent in all respects to S.

The X control permits specification of the (extra) fill-character to be used by L, R, and S. During each MASK operation, the fill-character will be a space until an X control is encountered, after which the character which follows the X in the control-string will be used. Any non-null member of the external character set may be specified.

The A control is eliminated and replaced by the T control, which performs a general translation operation. A literal-string, enclosed in apostrophes, follows the T in the control-string. The literal-string is composed of character-pairs; if the source-character is the same as the first character of any pair, it is replaced in the result-string with the second character of the pair. Otherwise, it is copied to the result-string unchanged. If some character appears as the first character of more than one character-pair, the first pair encountered in the literal-string is used. Within the literal-string, an apostrophe is represented as two consecutive apostrophes and a null character is represented by the pair "'N". (No ambiguity can arise from this arrangement, since N is not a valid control character.) If a null character is the last character in the literal-string, it may be omitted.

These extensions and generalizations correct the previously-mentioned deficiencies. They add the capability to scan the source-string backwards, to right-justify the source in the result, to perform character translation, and to repeat a series of controls a given number of times or until the source is exhausted.

TABLE III. EXTENDED MASK CONTROL CHARACTERS

Character	Replication	Qualifier	Semantics
S	F,n	none	Append current source character. If source is empty, append the current fill character (see X).
L	F,n	none	Same as S.
R	F,n	none	Take all remaining source characters and right-justify them in a field of width n. (FR is equivalent to FS).
X	none	one char	Change the fill character to the qualifier. (The fill character is set to a blank at the beginning of the operation.)
I	F,n	none	Discard current source character.
B	F,n	none	Append a blank.
/	F,n	none	Append a carriage-return.
E	F,n	none	Append hex-unpacked current source character (2-character result).
H	F,n	none	Append character formed by hex-packing current and following source characters.
U	F,n	none	Append binary-unpacked current source character (4-character result).
P	F,n	none	Append character formed by binary-packing current and 3 following source characters.
T	F,n	literal	See text.
'	F,n	rest of literal	Append literal (i.e. everything between this apostrophe and the next apostrophe which doesn't have another immediately following). An apostrophe within the literal is represented as two adjacent apostrophes.
←	none	none	If presently scanning the source from left to right, switch to right-to-left; the last character selected from the source will be selected again by the next control code which selects a source character. If presently scanning right to left, do nothing.
→	none	none	Reverse of ←.
(	F,n	rest of group	Execute the group (i.e. everything up to the matching right parenthesis) n times (if n-replicated) or until the end of the source has been reached (if F-replicated).

DEFICIENCIES OF THE SPL FORMAT OPERATOR

Unlike MASK, the SPL FORMAT operator contains conditional elements which are or are not placed in the result, or which appear in different forms in the result, depending on such circumstances as the sign of the mantissa or the exponent, the significance of adjacent digits, and the "exact/empirical" attribute of the number. The deficiencies of FORMAT are similar to those of MASK, consisting mainly of omissions and lacks of generality. Many derive from the handling of conditional elements.

The - and + controls cause a "floating" arithmetic sign to appear in the result. When they are used in a non-exponential template or in the mantissa part of an exponential template, the character placed in the result is selected on the basis of the sign of the number; when they are used in the exponent part of an exponential template the selection is based on the sign of the exponent. In contrast, the Q and R controls (which permit the conditional insertion of arbitrary character sequences depending on the sign) always interrogate the sign of the number, even when they appear in an exponent part. The ability to interrogate the sign of the exponent ought to be provided, at least within the exponent part.

The $ control produces a "floating" dollar sign in the result. $ is not permitted in an exponential template. (There is really no such thing as a "floating" sign in the result produced by an exponential template, because such templates cannot produce leading zeros. However, + and - are permitted in exponential templates, and $ might as well be since the prohibition complicates the rules and serves no useful purpose.) No provision is made for "floating" anything else except for the arithmetic sign.

The four digit-selectors D, N, Z, and * permit the programmer to specify that leading zeros be represented as 0, null, blank, or *, respectively. Considerable simplification as well as added generality would result if only one digit-selector (in addition to I) were provided and another control code (with a qualifier) were defined to specify the character to be used for leading zeros.

The C control code places in the result a comma (if the preceding digit-selector selected a significant digit) or the same zero-suppression character as the preceding digit-selector (otherwise). No other means of interrogating the zero-suppression status is available. Only the comma can be handled in this way.

The X and M control codes behave somewhat like + and - except that they interrogate the exact/empirical attribute of the number (instead of the sign) and produce the tag "EX" or "EM" as appropriate. No provision comparable to the Q and R codes is provided for this attribute.

The ability to replicate a series of control codes would be even more useful in FORMAT than in MASK, owing to the frequency with which one requires, for example, a template specifying a comma every three positions.

PROPOSED EXTENSIONS AND GENERALIZATIONS OF THE FORMAT OPERATOR

Our proposed changes to the FORMAT operator are summarized below. (See Table IV for the complete set of control characters.)

As in MASK, a replicator may be of unlimited magnitude. Replication may be applied to any control character except V, ., and $_{10}$ (for which it would not be meaningful), and Z (for which it could have no effect). Replication of parenthesized groups and nesting of parentheses are permitted.

The F-replicator is treated as in SPL FORMAT, with straightforward extension to groups. F-replication is permitted only for digit-selectors (and groups containing them) and is restricted to one F-replicator in an exponential template, or one F-replicator on each side of the decimal-point reference in a non-exponential template.

D and I become the only digit-selectors. The functions of N, Z, and * are performed by D, with the zero-suppression character specified by Z. (Although we are aware of no immediate applications for the added generality, we believe that the simplification alone is beneficial.)

The C, +, -, X, and M controls are modified by the addition of qualifiers, and the $ control is replaced by L, to permit handling of arbitrary literals. "X" variants of the Q and R controls are defined to permit interrogation of the sign of the exponent.

As in the case of MASK, the extensions and generalizations to FORMAT correct deficiencies and add capabilities. Of particular note here is the ability to apply a replicator to a group of control codes.

TABLE IV. EXTENDED FORMAT CONTROL CHARACTERS

Character	Replication	Qualifier	Semantics
D	F,n	none	Put digit in result unless it is a leading zero, in which case put the current zero-suppression character in result (see Z).
Z	none	one char	Change the zero-suppression character to the qualifier. A null is represented by the pair 'N; an apostrophe is represented by two apostrophes. (The zero-suppression character is set to 0 at the beginning of the operation.)
I	F,n	none	Discard digit.
B	n	none	Put a blank in result.
/	n	none	Put a carriage-return in result.
C	none	literal	Put literal (enclosed in apostrophes) in result unless the preceding digit-selector selected a leading zero, in which case put in result as many zero-suppression characters as there are characters in the literal.
L	n	literal	Put the literal ahead of the first digit in the result, following any zero-suppression characters ("floating" literal). L may not follow any of the control characters C, D, I, ., or V.
+	none	literal	"Float" the literal in the result if the source is positive. Positioning rules are the same as for L.
-	none	literal	"Float" the literal in the result if the source is negative. Positioning rules are the same as for L and +. If a template contains more than one "floating" element, all will appear in the result adjacent to one another in the order in which they appear in the template.
.	1	none	Put decimal point in result. Also serves as the decimal-point alignment reference for the template.
V	1	none	Serves as the decimal-point alignment reference but puts nothing in the result. V and . may not both appear in the same template.
$_{10}$	1	none	Causes the result to be in exponential form, and serves to separate the mantissa part of the template from the exponent part.

TABLE IV. (continued)

Character	Replication	Qualifier	Semantics
X	n	literal	Put literal in result unless the source is "empirical".
M	n	literal	Put literal in result if the source is empirical.
Q	n	literal or Xliteral	Put literal in result if source is negative, otherwise nothing. If X appears between Q and the literal, test the sign of the exponent.
R	n	literal or Xliteral	Put literal in result if source is positive, otherwise nothing. X has same effect as for Q.
'	n	rest of literal	Put literal in result.
(	F,n	rest of group	As if the group (i.e. everything up to the matching right parenthesis) appeared n times in the template. If F-replicated, the group must contain at least one digit-selector and may not contain another F-replicator. It will be treated as an n-replicated group with n the smallest possible integer (including zero) such that all remaining significant digits are accounted for.

APPLICATION OF OPERATORS TO STRUCTURES

The domain of the SPL operators is limited to the scalars. We propose to define the result of applying a monadic operator to a structure to be a structure of the same shape, with each scalar component replaced by the result of applying the operator to it. Fig. 1. recursively defines the resulting interpretation. (APL[19] applies substantially the same interpretation in such cases, the primary difference being that APL does not have arbitrarily-shaped aggregates.)

The generalization of dyadic operators to non-scalar values is slightly more complicated. We define the result of applying a dyadic operator to a scalar and a structure to be (again following APL) a structure of the same shape as the structure operand, with each scalar component replaced by the result of applying the operator to the scalar operand and the component. We then define the result of applying a dyadic operator to two vectors as a vector each of whose components is the result of applying the operator to the corresponding components of the operands. The definition is exemplified (for the case of addition) by the program in Fig. 2. (These examples should not be construed as implying that an implementation must employ recursive techniques.)

We also propose to recognize the assignment-structure construct as equivalent to any other structure value, thus permitting it to appear anywhere that an expression would be permitted.

The ability to apply operators to aggregates is useful in applications involving matrices, as in Gaussian elimination where each element of the pivotal row must be multiplied by the inverse of the pivotal element. The equivalence of assignment-structures with other structure values is of interest primarily as the elimination of a special case.

```
PROCEDURE NEG(X);

NOTE - APPLIES THE NOT OPERATOR TO AN ARBITRARY VALUE X;

  A ← X;
  IF SCALAR(A)
  THEN RESULT ← NOT A;
  ELSE J ← 1;  RESULT ← <>;
       WHILE IN A[J]
       DO   RESULT[J] ← NEG(A[J]);  J ← J + 1;
       END
  END
  RETURN RESULT;

 PROCEDURE SCALAR(X);  RETURN NOT IN X[1];  END

END
```

Fig. 1. Application of a Monadic Operator to an Arbitrary Value

```
PROCEDURE SUM(X, Y);

NOTE - ADDS ARBITRARY VALUES X AND Y, AND RETURNS THE SUM;

  A ← X;  B ← Y;
  IF SCALAR(A)
  THEN IF SCALAR(B)
       THEN RESULT ← A + B;
       ELSE J ← 1;  RESULT ← <>;
            WHILE IN B[J]
            DO   RESULT[J] ← SUM(A, B[J]);  J ← J + 1;
            END
       END
  ELSE J ← 1;  RESULT ← <>;
       IF SCALAR(B)
       THEN WHILE IN A[J]
            DO   RESULT[J] ← SUM(A[J], B);  J ← J + 1;
            END
       ELSE WHILE IN A[J] OR IN B[J]
            DO   RESULT[J] ← SUM(A[J], B[J]);  J ← J + 1;
            END
       END
  END
  RETURN RESULT;

 PROCEDURE SCALAR(X);  RETURN NOT IN X[1];  END

END
```

Fig. 2. Application of a Dyadic Operator to Arbitrary Values

SCALAR-STRUCTURE CONVERSION OPERATORS

SPL has eliminated much of the special handling traditionally fcund in I/O statements, in favor of providing MASK and FORMAT as operators usable in any context. There remain, however, three distinct variants of the INPUT and OUTPUT statements. We propose to eliminate the STRING and DATA variants, allowing INPUT and OUTPUT to perform as INPUT STRING and OUTPUT STRING and defining general operators to perform the special conversions.

The monadic operator STRING, applied to a structure, produces a scalar containing the external representation of the structure. Applied to a scalar, it encloses the value in field marks.

The monadic operator NAME, applied to a variable, produces a scalar containing the name of the variable. Applied to an expression, it produces a null. Applied to a formal parameter, it produces the name cf the actual parameter or a null, depending on whether the actual parameter is a simple variable or an expression.

The monadic operator DATA, applied to any variable or expression X, produces the equivalent of "NAME X JOIN (STRING X)". (X is evaluated only once, however.) Thus, the semantics of the statement "OUTPUT DATA X;" are substantially unchanged.

The monadic operator STRUCTURE, applied to a scalar, produces the structure whose external representation is that scalar. If the operand is not a valid external representation of any structure, the result is a null scalar. If the operand of STRUCTURE is a structure, the usual extension of monadic operators defined upon scalars (as defined in the previous section) applies.

The monadic operator LEN produces the length of (number of characters in) a scalar.

The monadic operator SIZE, applied to a vector, produces the number cf components in the vector. Applied to a scalar, it produces a null.

The elimination of SPL's INPUT and OUTPUT variants in favor of generalized conversion operations is a further application of the SPL principle of removing special cases from the I/O and providing operators which are usable in any context, including that of I/O. The LEN operator provides a function whose usefulness is unquestioned but which, in SPL, was lumped in with MASK where it was a rather alien presence. The SIZE operator is not available in any form in SPL; this lack has occasioned the writing of procedures to perform its function, which seems to be of considerable usefulness.

SUBSCRIPTION

SPL permits structures of arbitrary size and shape. Unfortunately, much of the potential power of these objects is unavailable due to the fact that subscript lists cannot be of variable length. Ghandour and Mezei[20] have proposed a set of definitions which are directed toward solving this sort of problem in the context of the APL language, but their proposal rests on data structures of needless complexity. For example, they make a fundamental distinction between a two-dimensional array of scalars and a vector each of whose components is a vector of scalars.

SPL requires that each subscript in a subscript list be a number whose integer part is in the interval [0,9999]. (The zero-valued subscript is a special case[2]; it has not been found particularly useful.)

Our proposal for the representation of subscript lists of varying length makes use of scalars which contain non-numeric characters and thus do not represent valid numeric values. We define a _simple subscript_ to be a character-string containing one or more valid numbers separated by semicolons; the subscript list then consists of those numbers. A bound-pair (p. 5) may appear at the end of the string. Thus if S has as its value the (13-character) string 12;36;42;5:10 the reference X[S] will be equivalent to the SPL reference X[12,36,42,5:10]. (We shall subsequently refer to the variable being subscripted -- X in this example -- as the _referent_.) A zero-valued subscript is treated as if it were a vector of all positive integers for which the components so accessed exist. One effect of this is that (if X happens to be a rectangular array) X[0,5] accesses the 5th column of X; another is that if the entire subscript is the null string the result is an access to the entire referent. A subscript may be a structure, in which case the result is the structure obtained by replacing each scalar component of the subscript with the component which it (as a simple subscript) selects from the referent.

In those cases where a variable number of subscripts is not needed, we permit a subscript to consist of one or more expressions separated by commas or colons. For example, the construct X[α,ϵ:ω] is interpreted as

```
X[ (α) JOIN |;| JOIN (ε) JOIN |:| JOIN (ω) ]
```

and thus in simple cases it has the expected effect. There is however no restriction that the expressions in this construct produce numbers or even scalars, provided that the result of the implied expression is a valid subscript.

We also propose:

1. To permit assignment to a partial reference, with the (expected) effect of replacing the selected substring with the value obtained by evaluating the right-hand side of the assignment (which in this case must be a scalar), and

2. To permit the application of subscription to expressions.

These extensions to SPL's subscript handling make for a very powerful facility for the manipulation of aggregates. By way of illustration, two examples of primitive operations which turn out to be special cases of subscription are the insertion and deletion of components of a vector. The deletion of the Jth component of a vector X is accomplished thus:

```
X ← X[< 1 | 2 | ... | J-2 | J-1 | J+1 | J+2 | ... | SIZE X >];
```

The insertion of a new component Q following the Jth component of a vector X is accomplished thus:

```
N ← SIZE X + 1;  X[N] ← Q;
X ← X[< 1 | 2 | ... | J-1 | J | N | J+1 | J+2 | ... | N-2 | N-1 >];
```

THE MATCH OPERATOR

We have now established the necessary constructs to enable us to define a powerful string-searching operator, which we call MATCH. In the following description, we shall refer to the left-hand operand of MATCH as the <u>subject</u> and to the right-hand operand as the <u>pattern</u>. Each operand may be either a scalar, or a structure of any shape. The result of the operation is formed by replacing each scalar component of the subject with a two-element vector which specifies the manner in which it matches the pattern, or with a null scalar if no match is found. The first element of the vector is the simple subscript which selects the matching component of the pattern; the second element identifies the character position in the subject at which the match was found. Fig. 3. is a program to emulate the MATCH operator.

```
PROCEDURE MATCH(SUBJECT, PATTERN);
  RESULT ← ||;  K ← FIRSTSCALAR(SUBJECT);
  WHILE K AFTER ||
  DO   SK ← SUBJECT[K];  LENSK ← LEN SK;
       L ← FIRSTSCALAR(PATTERN);  LOOK ← 1;
       WHILE (L AFTER ||) AND LOOK
       DO   PL ← PATTERN[L];  LENPL ← LEN PL;
            I ← 1;  STOP ← LENSK - LENPL + 1;
            WHILE (I LTE STOP) AND LOOK
            DO   IF SK[ I : LENPL ] SAME PL
                 THEN RESULT[K] ← < L | I >;  LOOK ← 0;
                 ELSE I ← I + 1;
                 END
            END
            IF LOOK THEN L ← NEXTSCALAR(PATTERN, L);
       END
       K ← NEXTSCALAR(SUBJECT, K);
  END
  RETURN RESULT;

 PROCEDURE FIRSTSCALAR(X);
   SUBS ← ||;
   WHILE IN X[SUBS, 1]
   DO   SUBS ← SUBS JOIN (|;| JOIN 1);
   END
   IF SUBS SAME ||
   THEN RETURN 0;
   ELSE RETURN SUBS[2:];
   END
 END

 PROCEDURE NEXTSCALAR(X, CURRENT);
   SUBS ← CURRENT;  LOOK ← 1;
   WHILE (SUBS AFTER ||) AND LOOK
   DO   J ← LEN SUBS;
        WHILE SUBS[J:1] AFTER |z| AND J GTE 2
        DO   J ← J - 1;
        END
        IF J = 1
        THEN LAST ← SUBS;
        ELSE LAST ← SUBS[J+1:];
        END
        SUBS ← SUBS[1:J-1];  LAST ← LAST + 1;
        IF IN X[SUBS, LAST]
        THEN IF SUBS SAME ||
             THEN SUBS ← LAST;
             ELSE SUBS ← SUBS JOIN (|;| JOIN LAST);
             END
             WHILE IN X[SUBS, 1]
             DO   SUBS ← SUBS JOIN (|;| JOIN 1);
             END
             LOOK ← 0;
        END
   END
   RETURN SUBS;
 END

END
```

Fig. 3. MATCH Operator

EXAMPLES OF THE USE OF EXTENDED SPL

To illustrate the use of some of our proposed extensions to SPL, we shall now show how certain constructs of PL/I and SNOBOL4 may be implemented using Extended SPL (ESPL).

PL/I INDEX Function

The PL/I statement X = INDEX (STR, 'ABXYZ'); (where X has any of various numeric types and STR is of type CHARACTER) is equivalent to the ESPL statement X ← (STR MATCH |ABXYZ|)[2]; . This finds the first occurrence in the string named STR of the substring ABXYZ.

PL/I VERIFY Function

The PL/I statement X = VERIFY (STR, 'ABXYZ'); (under the same conditions as above) is equivalent to the following ESPL statements:

```
X ← LIST(STR) MATCH LIST(|ABXYZ|);
I ← 1;
WHILE IN X[I,1]
DO   I ← I + 1;
END
IF IN X[I]
THEN X ← I;
ELSE X ← 0;
END

PROCEDURE LIST(STRNG);  S ← STRNG;
NOTE - RETURNS A VECTOR CONTAINING THE CHARACTERS OF STRNG;
   VEC ← ||;  J ← 1;
   WHILE (C ← STRNG[J:1]) AFTER ||
   DO   VEC[J] ← C;  J ← J + 1;
   END
   RETURN VEC;
 END
```

This finds the first character in the string named STR which is not a member of the set {A,B,X,Y,Z}.

PL/I TRANSLATE Function

The PL/I statement X = TRANSLATE (STR, 'OA', 'QB'); (where both X and STR are of type CHARACTER) is equivalent to the ESPL statement

X ← STR MASK |FT'OQAB'|; .

Sample of SNOBOL4 Pattern-Matching Operation

The SNOBOL4 statement

```
STR ARB . P1 ('AB' | 'BC') . P2 'DE' ARB . P3 RPOS(0)
```

which looks for the sequence ABDE or BCDE in the string named STR and (if the search succeeds) assigns the part of STR preceding the sequence to the variable P1, the AB or BC to the variable P2, and the part of STR following the DE to the variable P3, is equivalent to the following Extended SPL statements:

```
PAT<ABDE|BCDE>;
J ← (STR MATCH PAT)[2];
IF J NEQ 0
THEN P1 ← STR[1:J-1];
     P2 ← STR[J:2];
     P3 ← STR[J+4:];
END
```

It is worth noting that the ESPL version of this process is more easily understood than the SNOBOL4 version.

A Practical Example

The following example is typical of the sort of processing which is involved in a text-editing program. We shall assume that the user has requested that the program find and print in context the first instance of any of several given words in his text. Only words completely matching a list element are desired, i.e. a request to find "the" should not yield "hypo<u>the</u>tical." The variable STR contains the text; the variable CR contains a carriage-return character. The variable WORDS contains the list of words to be searched for, separated by commas.

As we have noted before, the reader who is not familiar with SNOBOL4 should not be concerned. The SNOBOL4 solution is given only as a contrast to the ESPL solution for the benefit of SNOBOL4 users, and can be safely skipped.

SNOBOL4 solution

```
* ALTERNATION OF WORD-SEPARATOR CHARACTERS
   PUNCT = CR | ' ' | '-' | '.' | ',' | ';' | ':'
   PUNCT = PUNCT | '(' | ')' | '"' | "'" | '?' | '!'

* CONVERT WORDS TO AN APPPOPRIATE ALTERNATION
   PAT =
   WORDS  BREAK(',') . WORD LEN(1) =       :F(L2)
   PAT = WORD
L1 WORDS  BREAK(',') . WORD LEN(1) =       :F(L2)
   PAT = PAT | WORD                        :(L1)
L2 PAT = PAT | WORDS

   &ANCHOR = 0
   STR  (LEN(20) PUNCT PAT PUNCT LEN(20)) . OUTPUT
```

ESPL solution

```
NOTE - VECTOR OF CHARACTERS TO BE RECOGNIZED AS WORD SEPARATORS;
PUNCT<| |-|.|,|;|:|(|)|'|"|?|!>;  PUNCT[1] + CR;

NOTE - INITIALIZE SPECIAL-CHARACTER VARIABLES;
NOTE - LEFT GROUP MARK;   LGM + (STRING <>)[1:1];
NOTE - RIGHT GROUP MARK;  RGM + (STRING <>)[2:1];
NOTE - FIELD MARK;        FM  + (STRING ||)[1:1];

NOTE - CONVERT "WORDS" TO AN APPROPRIATE VECTOR;
M + |FT',| JOIN FM JOIN | '|;
PAT + STRUCTURE (LGM JOIN (WORDS MASK M) JOIN RGM);

J + 1;  TPAT + ||;
WHILE IN PAT[J]
DO   TPAT[J] + PUNCT JOIN PAT[J] JOIN PUNCT;  J + J + 1;
END
J + (STR MATCH PAT)[2];
IF J NEQ 0 THEN OUTPUT STR[J-20:50];  END
```

CONCLUSION

We have proposed modifications and extensions to SPL to correct various deficiencies and to improve the completeness of the language. The extensions include definitions of:

1. a powerful pattern-matching operator,

2. a means of varying, during execution, the number of subscripts used in accessing a structure,

3. iteration and case statements, and

4. the application of operators defined upon scalar values to operands which are structures.

For these extensions, we have adopted a host language which, without sacrificing capability, is exemplary in its simplicity, and we have avoided the baronial splendor of the SNOBOL4 pattern-matching operation in favor of a definition which we hope will be comprehensible to mere mortals.

The primary modifications deal with the MASK and FORMAT operators, which are made more general and from which a number of special cases have been removed. All recognized deficiencies of the MASK operator have been corrected; the result, together with the JOIN and MATCH operators, is a string-processing language whose power is comparable to that of SNOBOL4. The major deficiencies of SNOBOL4, lack of reasonable control constructs and the necessity of recourse to pattern-matching "side-effects" to retain information of interest, are avoided. The power of the FORMAT operator greatly exceeds that of the similar SNOBOL4 facilities.

We claim that the significance of the facilities herein proposed is in no way limited to SPL or to the SYMBOL-2R system. The MASK operator is applicable to any programming language which provides a character-string data type (preferably varying-length strings), and the FORMAT operator is applicable to any such language which also supports the concept of a numerical value. The MATCH operator could easily be adapted to any language which supports character-strings (the subject), vectors of character-strings (the pattern), and vectors of numbers (the result). The SPL facility of dynamically-varying structures of arbitrary size and shape is not needed for MATCH, and indeed is somewhat of a complication.

While we have not mentioned implementation considerations in this discussion, our experience with the SYMBOL-2R system suggests that implementation of the facilities which we propose, given an implementation of SPL as it now exists, would be straightforward. With respect to implementation in the context of other languages, the fundamental requirements for implementing MASK and MATCH are the ability to scan a string on a character-by-character basis in both directions, and the ability to determine whether two characters are or are not the same. Any machine which cannot do such things easily will not likely be used for character manipulation. (We might note that compilers of programming languages must inherently perform a considerable amount of string manipulation, and therefore a machine which is not well suited to such tasks should probably have a companion which can handle them, if only for the purpose of running a compiler.) FORMAT requires, in addition to string-scanning capability, some means of converting from the form in which numerical quantities are kept to a character-string representation. Usually, such facilities are provided, if only for the purpose of output.

We have not addressed the interesting question of whether it is necessary to view a scalar as fundamentally different from a vector of one component. We consider that question, as well as the task of implementing the language proposed herein, as suitable topics for further investigation.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the efforts of Hamilton Richards, whose assistance in familiarization with and understanding of the SYMBOL system was invaluable, and of professors R. J. Zingg, C. T. Wright, and J. P. Basart, who examined the manuscript and made many helpful suggestions. Thanks are also due to Mrs. LaDena Bishop of the ISU Thesis Office for assistance with the form of and annotations in the tables and for pointing out a number of clerical errors which would surely not have been corrected otherwise.

The proposed case statement is very similar to a proposal relating to the COBOL language which the author accidentally found some time ago while looking for something else. An extensive search has failed to turn up the source.

Financial support for the SYMBOL project, with which the author has been associated, has been provided by the National Science Foundation under grant GJ33097X.

REFERENCES

1. Smith, W.R., et al. "SYMBOL -- A Large Experimental System Exploring Major Hardware Replacement of Software." AFIPS Conference Proceedings 38 (Spring 1971): 575-587.

2. Richards, H. SYMBOL II-R Programming Language Reference Manual. Cyclone Computer Laboratory, Iowa State University, 1971.

3. Strong, J., et al. "The Problem of Programming Communication with Changing Machines." Communications of the ACM 1, 8 (August, 1958): 12-18.

4. OS PL/I Checkout and Optimizing Compilers: Language Reference Manual. IBM Corporation, White Plains, New York, 1974.

5. WATFIV User's Guide. University of Waterloo, Waterloo, Ontario, Canada, 1972.

6. Wirth, N. "The Programming Language Pascal." Acta Informatica 1, 1 (1971): 35-63.

7. Green, J. "Remarks on ALGOL and Symbol Manipulation." Communications of the ACM 2, 9 (September, 1959): 25-27.

8. Yngve, V. H. "A Programming Language for Mechanical Translation." Mechanical Translation 5, 1 (July, 1958): 25-41.

9. Farber, D. J., Griswold, R. E., and Polonsky, I. P. "SNOBOL, A String Manipulation Language." Journal of the ACM 11, 1 (January, 1964): 21-30.

10. Griswold, R. E., Poage, J. F., and Polonsky, I. P. The SNOBOL4 Programming Language. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971.

11. Balzer, R. M., and Farber, D. J. "APAREL -- A Parse-Request Language." Communications of the ACM 12, 11 (November, 1969): 624-631.

12. Dijkstra, E. W. "The Humble Programmer." Communications of the ACM 15, 10 (October, 1972): 859-866.

13. Richards, H. and Wright, C. "Introduction to the SYMBOL 2R Programming Language." Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, Association for Computing Machinery, New York, 1973: 27-33.

14. Dijkstra, E. W. "GO TO Statement Considered Harmful." Communications of the ACM 11, 3 (March, 1968): 147-148.

15. Rice, John R. "The GO TO Statement Reconsidered," and reply by E. W. Dijkstra. Communications of the ACM 11, 8 (August, 1968): 538 ff.

16. Knuth, D. E. "Structured Programming with GO TO Statements." Computing Surveys 6, 4 (December, 1974): 261-301.

17. Richards, M., Evans, A., and Mabee, R. F. The BCPL Reference Manual. Project MAC Technical Report MAC-TR-141. Massachusetts Institute of Technology, Cambridge, Mass., 1974.

18. Dakins, M. C. "Nonnumeric Processing in the SYMBOL 2R Computer System." M.S. Thesis, Iowa State University, 1974.

19. Pakin, S. APL/360 Reference Manual. Science Research Associates, Chicago, 1972.

20. Ghandour, Z., and Mezei, J. "General Arrays, Operators, and Functions." IBM Journal of Research and Development 17, 4 (July, 1973): 335-352.