MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 142 September 1967.

STRING

Peter Samson

This document describes the STRING programming language, which has been implemented on the MAC Artificial Group's PDP-6 computer. In the STRING system, all objects -- constants, variables, functions and programs -- are stored and processed in the form of strings of characters. The STRING language is unusually concise, yet at the same time unusually rich in commands, including a strong arithmetic facility.

Basically, the STRING program is used in a conversational manner: the user types in a character string terminated by the character ALT MODE (alias ESC) and the program evaluates the string and types out its value. However, since the evaluation can involve any amount of iteration, recursion, input and output, evaluating a string may equally be regarded as running a program, with the returned value of no consequence.

Notation and Definitions:

␣ is the character "space".

$\Rightarrow$ means "evaluates to".

$a \equiv b \Rightarrow c$ means " a and b each evaluate to c ".

null describes the string of no characters.

a word is a character string delimited by a space.

x^n means "the character x , n times in a row".

ITEMS, OPERATORS AND OPERANDS; THE CURRENT VALUE

A string being evaluated is processed from left to right, and as part of that processing is parsed into substrings of two types: items and operators. Items have substantive meaning; that is, an item has a value "of its own". Operators take one or two operands and produce a value which depends on the values of the operands. An operator which takes one operand is termed unary; one which takes two is binary. A unary operator immediately follows its operand; a binary operator immediately follows its first operand and immediately precedes its second operand, which is an item. (A few operators in fact ignore their left operand.) If a binary operator is immediately followed by another operator, the second operand of the binary operator is the null string between the two operators. (A null string evaluates to null.)

There is always maintained a *current value*: at the beginning of a string this is null; after an item (which is not the second operand of a binary operator) it is the value of that item; after a unary operator the value of that operator; and after the second operand of a binary operator, the value of the binary operator. The current value at the end of a string is what is returned as the value of the string. The current value when a binary operator is encountered is what is taken as the first operand, and likewise as the operand of a unary operator; the value of the item to its right is taken as the second operand of a binary operator. (A very few binary operators use the item to their right immediately as their second operand, rather than its value. This will be made clear in the description of each such operator.)

QUOTING

The simplest type of evaluation is *quoting*, where the value a string is wished to have is given explicitly. The STRING system provides two quote notations:

"stg␣ ⇒ stg , where *stg* is any string not containing a ␣ (space);
[[stg]] ⇒ stg ("superquote"), where *stg* may be any character string; except that any]] in *stg* must be matched by a prior [[.

"stg␣ and [[stg]] are items.

NUMBERS

There are two types of numbers: *precise* and *imprecise*. A non-null character string is a *precise number* if it contains no characters except at least one digit (Ø to 9), and optionally a decimal point (alias period). A non-null character string is an *imprecise number* if it contains no characters except decimal point, digits, and atsign (@), with at least one digit or decimal point, and at least one atsign. (If there are no digits, the decimal point must not be the first character.) Atsign in a number serves as a non-significant digit. A number is *totally imprecise* if the first digit other than leading zeroes is atsign. In a totally imprecise number, the number of atsigns to the left of the decimal point, or the number of leading zeroes to the right of the decimal point, indicates the order of magnitude of the number.

A number is an item; it evaluates to itself (is implicitly quoted). (Note: in a few cases a string of digits terminated by a decimal point or a comma is not taken as a quoted character string, but rather as a numeric modifier for a command or for evaluation control. These cases will be described below.)

SPACE, CARRIAGE RETURN, ETC.

The character ␣ (space) is an *item* which evaluates to null. (Note: in some contexts the space character has a special meaning and is not evaluated. For instance, it is the terminator of a " quote.) Horizontal tab, line feed, vertical tab, and form feed are unary *operators* which ignore their operand and return the null value. Carriage return is a unary operator which returns the current value as its value, i.e. has no effect.

PARENTHESES

It is frequently desired to have an expression more complicated than an item as the second operand of a binary operator. For this purpose are provided ( ) parentheses, which may surround any string; except that a) in that string must be matched by a prior (also therein. The parentheses delimit an item as seen from the string in which they appear.

When (is encountered, the current value is set aside, and the string within the parentheses is evaluated with an initial current value of null. The) acts as the end of the string, and the value returned then is the value of the item which is delimited by ( ) . In other words, (causes a pushdown of one level and) causes a value to be returned to the upper level.

CONCATENATION

' binary operator concatenate. (Example: 1'2⇒12 .)

,' binary operator reverse concatenate. (Example: 1,'2⇒21 .)

ARITHMETIC

+ binary operator add. It is defined on precise numbers in the natural way, with the result similarly a precise number. If both operands are numbers and either is imprecise, the result will be an imprecise number; the leftmost atsign in the result will align with the leftmost atsign in either operand; and for the purpose of determining the carry into the significant portion of the result for each atsign in each operand will be taken a random digit (Ø to 9) during the addition.

If either operand is not a number (with optional leading + or - sign), but is an arithmetic expression (a string whose only operators are arithmetic ones), the result will be the correct arithmetic expression. In such a result all items except numbers appearing in the two operands will appear in the same order (left-to-right) as they appeared in the operands, left operand first. Numeric terms may be rearranged, and possibly combined according to the rules for two numeric operands.

If one operand is null, value is the other operand.

,+ binary operator reverse add. Like + but puts non-numeric terms from second operand before those from first.

- binary operator subtract. Negates second operand, then does + . (Null negated is null.)

,- binary operator reverse subtract. Exchanges operands, then subtracts.

* binary operator multiply. The symbolic terms of the operands are handled as by + .

When multiplying two numeric operands (or numeric parts of expression operands) there is first calculated the "significance of the result". This is a number which is the minimum of: the number of significant digits (excluding leading zeroes) in the first operand; said number in the second operand; and the current significance. (The current significance is set by commands described below.) If an operand is a precise number, its number of significant digits is assumed to be infinite. In imprecise operands, the atsign is numerically of value zero.

The numeric product will be imprecise if either numeric operand is, or if the number of digits in the result is as great as the current significance; and the number of digits to the left of the first atsign in the result will be equal to the precalculated "significance of the result". Otherwise the numeric product will be precise.

,* binary operator _reverse multiply_.

*+ binary operator _positive multiply_. Identical in effect to * .

,*+ binary operator _positive reverse multiply_. Identical to ,* .

*- binary operator _negative multiply_. Negates second argument, then multiplies.

,*- binary operator _negative reverse multiply_. Exchanges operands, negates new second operand, then multiplies.

/ binary operator _divide_. If both operands are numbers, divide precalculates the significance of the result in the same manner as multiply. Then if the significance of the result is infinite, / performs division by the schoolboy algorithm and has the value of the quotient. If the significance of the result is the finite number _n_ , exactly _n_ quotient digits (not counting leading zeroes) are generated; the quotient is marked imprecise by a final atsign (after a decimal point if needed); this quotient is returned as the value.

The remainder is created, and saved; see below. The quotient is negative if exactly one of the operands is; the remainder takes the sign of the dividend. The results are always such that (quotient * divisor) + remainder = dividend , though with a possible loss of precision if the significance of the result is not infinite. The decimal point in the remainder aligns with that of the dividend; the number of digits provided to the right of the decimal point in the quotient (if not zero) will be that number in the dividend minus that number in the divisor.

If not both operands are numbers, / does _indicated division_: it concatenates its first operand, the character / , and its second operand (in parentheses if necessary). Indicated division sets the remainder to null.

In the special case where the first operand is null, / generates the _reciprocal_ of its second operand. If the significance of the result is to be infinite, the operand is divided into $1.\emptyset^{n}$ (1. followed by _n_ zeroes) where _n_ is the total number of digits in the given operand (excluding leading zeroes); if the significance is to be finite number _m_ , then _m_ digits of quotient are developed and the result is marked imprecise. Reciprocation generates a numeric remainder.

In the case where the second operand is null, value is the first operand, and the remainder is not affected.

,/ binary operator _reverse divide_. Exchanges operands and divides.

/+ binary operator _positive divide_: identical in effect to / .

,/+ binary operator _positive reverse divide_: identical to ,/ .

/- binary operator _negative divide_. Negates second operand and divides.

,/- binary operator _negative reverse divide_. Exchanges operands, negates new second operand, divides.

#/ binary operator divide for integer quotient. The quotient will be a precise number if its number of significant digits does not exceed the calculated significance of the result. A correct remainder is generated.

,#/ binary operator reverse divide for integer quotient.
,#/+ binary operator reverse positive divide for integer quotient.
,#/- binary operator reverse negative divide for integer quotient.

```
EXAMPLES:  (assuming current significance is infinite)
           5+3*2⇒16
           5+(3*2)⇒11
           "A+B-2‿,-"C+5.‿⇒C-A-B+3.
           4.6/2⇒2.3          (remainder is .Ø)
           4.6/2.Ø⇒2          (remainder is .6)
           4.6/2.ØØ⇒2         (remainder is .6Ø)
           4.6@/2.ØØ⇒2.3@     (remainder is .Ø@)
           4'2/7⇒6  (remainder is Ø)
           4'"/‿'2⇒4/2        (does not affect remainder)
           "X‿/3⇒X/3          (remainder null)
```

SYMBOLS

A string of letters and digits (atsign being taken as a digit) which contains at least one letter (or a string consisting of nothing but atsigns) is a symbol. A symbol may be of any length. It is an item. A symbol may be either defined or undefined; initially all symbols are undefined (except @ , for which see below).

An undefined symbol is implicitly quoted: it evaluates to itself. A defined symbol has a symbol value, which is a character string that becomes associated with the symbol as it is defined. The value of a defined symbol is the result of evaluating its symbol value. As the processor evaluates a typed-in string, if it encounters a defined symbol it sets aside its current position in that string, the current accumulated value, and other information regarding its processing of that string, and commences evaluting the symbol value in the same manner as it would evaluate a typed-in string. Then if while processing the symbol value it finds it has to evaluate a defined symbol, it again sets aside its work and commences evaluation of that symbol's symbol value. The evaluation of each such string (typed-in string or symbol value) is said to occur on a given level. The typed-in string is on the top level, or level number 1; the symbol value of a symbol encountered on level 1 is evaluated on level 2, etc. (The system as currently arranged has facilities for approximately 1ØØØ levels.)

It is possible to define a symbol in such a way that that definition is effective only until the level on which it was defined returns a value to the next highter level. A symbol so defined is called a level symbol, and the number of the level on which it was defined is associated with it as its definition level. A level symbol may be used on lower (greater-numbered) levels. A symbol not so defined is called a universal symbol, which is indicated internally by giving it definition level Ø. More than one definition of a symbol may exist at a time, though each will have a different definition level. Only one definition is active at a time; that is the one with the largest-numbered definition level (but not greater than the current level number).

DEFINING

; (semicolon) binary operator level define. The operand to the right must be a symbol; it is not evaluated. The accumulated value to the left (the "current value") is preserved as the value of the semicolon operator, as well as being its left operand. This operator causes the symbol to its right to become defined as a level symbol, with a symbol value identical to the left operand and a definition level equal to the current level of evaluation. If there was already a definition of the symbol with the same definition level, that definition is completely removed. Any other existing definitions are allowed to remain, but they will not be active until this definition is removed. This definition is removed as evaluation returns from this level to a higher level (smaller numbered).

, (comma) binary operator define. The operand to the right must be a symbol; it is not evaluated. The current value as this command is encountered is preserved as the value of the comma operator, as well as being taken as its left operand. Comma causes the symbol to its right to become defined as a universal symbol with a symbol value identical to the left operand. Any and all existing definitions of this symbol (except those with definition level numbers greater than the number of the current level) are first removed.

The symbol ā is always defined and always evaluates to ␣ (space).

Note: When a definition is removed whose symbol value is still being processed on the current level or a higher one (smaller-numbered), although the definition is removed the symbol value part of it is preserved until evaluation of it is finished.

```
EXAMPLES: X+3,Y⇒X+3  \      assuming X is undefined
          Y⇒X+3       |     (now)
          5,X⇒5       }           .....illustrates formal
          Y⇒8         |                definition of Y
          6,X⇒6       |
          Y⇒9        /
          X+3,Z⇒9    \      X now is defined
          5,X⇒5       }           .....illustrates numeric definition of Z
          Z⇒9        /

[[U*U+(V*V)]],FN1⇒U*U+(V*V)       function with parameters U and V
[[3;U␣FN1]],FN2⇒3;U␣FN1           calls FN1 with specific value for U, without
                                     affecting upper levels' definition(s) of U
```

EVALUATION CONTROL

Often it is desired to limit the depth in levels to which a symbol is evaluated. Such a usage tends to correspond to the use of a symbol as a string variable, rather than a formal expression.

.sym where sym is any symbol, is an item which evaluates to the symbol value of sym if sym is defined, and null otherwise. This symbol value is quoted, i.e. not evaluated.

..sym commences evaluating sym in the usual manner, but imagines that every symbol to be evaluated in the symbol value of sym is preceded by exactly one period.

...sym like ..sym , but imagines two periods in front of each symbol except those which appear explicitly with one preceding period.

The above description extends to any number of periods. The maximum evaluation depth permitted is a parameter of each level; for the top level it is set larger than the number of available levels; when a symbol is not preceded by any periods the depth limit on the new level is set 1 less than on the current level; if there are _n_ periods before the symbol, the limit on the new level is the _lesser_ of: _n_ , and 1 less than the current limit. When the limit on a level would be 1, that level is not entered, but instead of evaluating a symbol's symbol value on the new level that symbol value is simply quoted.

n.sym where _sym_ is a symbol (except one composed only of atsigns), and _n_ is a string of digits interpreted as a decimal integer, is identical to writing _n_ periods before _sym_. Briefly, $\text{n.sym} \equiv .^{n}\text{sym}$.

REMAINDER

\ (backslash) item _remainder_, defined by divide operation. Backslash possesses many properties of a level symbol: when it becomes defined, a previous definition will be removed only if it occurred on the same level; when evaluation returns to a higher level than the one on which the current definition was created, that definition is removed and any higher-level definition becomes again active. Backslash is not a symbol.

THE CURRENT SIGNIFICANCE

The current significance is a parameter of each level. When a level is entered, its current significance is set to the value at that moment of the current significance on the level above.

;? unary operator _set current significance_. If the value of the operand is a number, its integer part is taken and the current significance on this level is set to it. If it is not a number, current significance is made infinite. Maintains current value.

#? unary operator _set level Ø significance_. This processes its operand like ;? and sets the quantity to which the current significance is set when level 1 is entered. This does not affect the current significance. The current value is maintained.

Initially the level Ø significance is infinite.

ARGUMENT

↑ (up arrow) item _argument_. When encountered, causes evaluation to _visit_ the immediately higher (smaller-numbered) level. That level is the one in which was found the symbol in whose symbol value the up arrow was encountered. The processing pointer on the higher level had been left just to the right of that symbol. The up arrow command commences processing as if back on the upper level, as follows:

(1) the accumulated value is set to null.

(2) if the character to the right of the processing pointer is a space, the pointer is advanced over it.

(3) evaluation resumes in a normal manner on the higher level, _but_ the character ␣ (space) is treated _not_ as an item with

value null, but as a signal to go back to the lower level, carrying the accumulated value on the upper level as the value of the ↑ . (If the end of the string on the upper level is encountered, that acts as such a signal too.) The upper level pointer is now in the position just to the right of the space signal (or at the end of the string). Hence successive uses of ↑ on a level evaluate successive arguments on the level above, passing the pointer over each in turn.

Performance of ↑ is not a <u>return</u> to the upper level and therefore no level symbol definitions are removed. Those created on the lower level are invisible on the upper level, however.

.↑ item <u>quoted argument</u>. Makes use of the pointer on the level above.

(1) if the character to the right of that pointer is a space, the pointer is advanced over it.

(2) then the pointer is advanced until a space (or the end of the string) is encountered; the string of characters so passed over (not including the space at the end) is returned as the value of the .↑ with no further evaluation. The pointer on the upper level is left to the right of the terminating space.

Note: The pushdown caused by (is identical, as far as the lower level can see, to that done to evaluate a symbol; a new level is used in each case. The "pop up" caused by) is identical to that when a value is returned at the end of a string. So a level symbol definition occurring within ( ) parentheses is cancelled at the) ; also note that an unmatched) can be used to mean "return the current value to the upper level".

```
EXAMPLES: 5,X⇒5
          [[X+2]],Y⇒X+2
          [[Y*3]],Z⇒Y*3
          "Z‿⇒Z
          .Z⇒Y*3
          ..Z⇒X+2*3        the * was performed, the + was not
          ...Z≡Z⇒21

          "↑+1‿,P1⇒↑+1
          "↑*2‿,T2⇒↑*2
          P1‿5≡P1‿5‿⇒6
          T2‿P1‿4/2⇒6
          T2‿P1‿4‿/2⇒4
          T2‿P1‿4‿‿/2⇒5
          T2‿P1‿4‿‿‿/2⇒.5       last space evaluated!
```

ATTACH MODE

Besides having reverse mode, the five combinative operators (concatenate, add, subtract, multiply, and divide) have attach mode and reverse attach mode. The attach process is described below for concatenation, with the details applicable to the other attach mode commands listed.

$' binary operator <u>attach concatenate</u>. Operand to the right must either be a symbol, or be one or more periods followed by a symbol (the form n.sym is not permitted in this context). First the $' evaluates its operands conventionally, and does a concatenation like ' , returning the result of that concatenation as the value of the $' . Additionally, however, the $' defines a certain symbol with a symbol value identical to the concatenated result. If the right operand of the $' had no periods, or had just one, the symbol to be defined is the symbol in that right operand. But if the right operand had more than one period, said operand is at this time evaluated again with <u>one fewer period</u>. The resulting value is a character string which should begin with a symbol: and <u>that</u> symbol is the one defined as the result of the concatenation. If the symbol to be defined had an active level symbol definition at the time $' was to (re)define it, the new definition is a level symbol definition; otherwise it is a universal symbol definition.

:' binary operator <u>attach reverse concatenate</u>. Does reverse concatenate, defines original right operand as result.
$+ binary operator <u>attach add</u>. Like $' but of course does addition, not concatenation.
:+ binary operator <u>attach reverse add</u>.
$- binary operator <u>attach subtract</u>.
:- binary operator <u>attach reverse subtract</u>.
$* binary operator <u>attach multiply</u>.
$*+ binary operator <u>attach positive multiply</u>.
$*- binary operator <u>attach negative multiply</u>.
:* binary operator <u>attach reverse multiply</u>.
:*+ binary operator <u>attach reverse positive multiply</u>.
:*- binary operator <u>attach reverse negative multiply</u>.
$/ binary operator <u>attach divide</u>.
$/+ binary operator <u>attach positive divide</u>.
$/- binary operator <u>attach negative divide</u>.
:/ binary operator <u>attach reverse divide</u>.
:/+ binary operator <u>attach reverse positive divide</u>.
:/- binary operator <u>attach reverse negative divide</u>.

Note: attach and reverse attach modes also exist for the #/ operator (divide for integer quotient), including the positive and negative versions of it.

CONDITIONALS

Certain operators, about to be described, are termed <u>conditionals</u>: according to whether or not its operands meet particular conditions, such an operator will or will not <u>skip</u> the next word in the string being evaluated. If the skip condition is not met, processing resumes immediately to the right of the right operand. If the skip condition is met, the following steps are taken: (1) The character to the right of the right operand is examined: if it is a space, the processing pointer is moved over it. (2) The pointer is advanced to the right until it passes over a space (or reaches the end of the string). However, a given space will not be seen if it is (a) inside [[ ]] superquotes; or (b) the terminator of a " quote; or (c) somewhere within ( ) parentheses. The (to be effective in hiding a space must be encountered during the skip, i.e. it may not be outside the string of characters skipped over.

= binary conditional operator <u>skip if equal</u>. Skips if the values of its two operands are identical character strings.
:= binary conditional operator <u>skip if not equal</u>. Skips only if the values of its two operands are not identical character strings.

In the following numeric comparison conditionals, if an operand is an imprecise number each atsign in it is replaced by a random digit (Ø to 9).

#> binary conditional operator skip if numeric greater. Skips if both operands are numbers (with optional leading + or - signs) and the first is greater than the second.

#< binary conditional operator skip if numeric less. Skips if both operands are numbers (with optional leading + or - signs) and the first is less than the second.

:> binary conditional operator skip if not greater. Skips if both operands are (optionally signed) numbers and the first is not greater than the second.

:< binary conditional operator skip if not less. Skips if both operands are (optionally signed) numbers and the first is not less than the second.

#= binary conditional operator skip if numeric equal. Skips if both operands are (optionally signed) numbers of equal numeric value.

#:= binary conditional operator skip if numeric not equal. Skips if both operands are (optionally signed) numbers whose numeric values are not equal.

:#= binary conditional operator skip if not numeric equal. Identical to #:= .

Each of the conditionals in this section returns the value of its right operand; with the numeric conditionals any atsigns in either operand will have been replaced by the random digits actually compared.

TAGS, GOTOS AND DISPATCHES

[sym where sym is any symbol (or string of digits) is a tag. A tag is used to identify a point in a string being processed to which the processing pointer may be sent by a goto. When processing passes through a tag, the current value is not affected.

]sym where sym is similarly any symbol or string of digits is a goto. If it is performed, the string in which it appears is scanned from the beginning (i.e. left end) for an appearance of [sym . If that tag is found, processing resumes to the right of the tag. (If it is not found, an error results.) When a goto is performed, the current value is set to null.

[where the character following the [is a carriage return or line feed is a "new line" command which passes the pointer over characters to the right of the [so long as each is a carriage return or line feed, without affecting the current value.

[where the character following the [is not [, carriage return, line feed, a letter, a digit, or atsign, is a null tag.

] where the character following the] is not] , a letter, a digit, or atsign, is the unary operator dispatch. The current value is taken: if it begins with a symbol (or string of digits), the dispatch operator attempts a goto on that symbol. (If it is not found, processing resumes to the right of the] .) If the current value does not begin with a symbol or string of digits (for instance, if it is null) the dispatch operator performs a null goto. This is a goto to a null tag: to the null tag nearest to the left of the dispatch if there is one; the null tag nearest to the right of the dispatch otherwise; if there is no null tag at all, processing resumes to the right of the dispatch. The dispatch operator always sets the current value to null.

Note: The character [is used both for a tag (followed by a letter or digit), for "open superquote" (followed by another [ ), for "new line" (followed by any number of carriage returns and line feeds), and for the null tag (otherwise). All evaluation, skipping, and searching for tags is done from left to right (even when searching for the null tag nearest to the left), so that the case of multiple ['s in a row is always parsed the same way, viz.: pairs are open superquotes, and then a single one is a tag or new-line command.

ADDITIONAL DEFINITION COMMANDS

,, binary operator define via. The right operand is evaluated; its value is expected to begin with a symbol and that symbol is defined as a universal symbol with a symbol value identical to the left operand, which is also returned as the value of the operator.

;; binary operator level define via. Evaluates right operand like ,, and defines it as a level symbol like ; .

: binary conditional operator undefine. Right operand must be a symbol; it is not evaluated. If it is defined, the active definition is removed. Then if no definition remains (or the symbol was undefined to start with) the : operator skips. The current value is not used, except that it is returned as the value of the operator.

:: binary conditional operator undefine via. Evaluates right operand like ,, and does undefine-and-or-skip like : .

ADDITIONAL CONDITIONALS

#sym where sym is any symbol, skips if sym is defined and not otherwise. The current value is not changed.

:#sym like #sym but skips only if sym is not defined.

binary conditional operator skip if defined via. Evaluates right operand to get a symbol as does ,, then skips if that symbol is defined. Value is left operand.

:## binary conditional operator skip if not defined via. Like ## , except skips only if symbol to which right operand evaluates is not defined.

#␣ unary conditional operator skip if number. If the current value is a number (with optional leading + or - sign) this operator skips. The current value is in any case returned as the value of the operator. The space is absorbed as part of the operator and is not evaluated nor involved in the skipping. (However, it would be seen by a skip by some prior command.)

:#␣ unary conditional operator skip if not number. Like #␣ but with inverted skip sense.

```
EXAMPLES: [[↑;N#>Ø␣1)␣N*FACT1␣N-1]],FACT1
          [[↑;N␣1;F␣[␣N#>Ø␣F)␣N$*F␣1:-N␣]␣]],FACT2
```

These are two alternative definitions of the factorial function. The first is recursive, and uses N+1 levels to evaluate the factorial of N . The second is iterative, and performs the null goto N times in the same case. When a choice exists between these two approaches -- iteration and recursion -- THE ITERATIVE METHOD IS HIGHLY RECOMMENDED. This is because (a) the number of available levels is not unlimited, and (b) a significant amount of overhead time is taken pushing down a level and popping up.

SEARCHING

A very important operation that can be performed on character strings is that of searching in one string for an occurrence of some other string. In the STRING system, all searches are performed on (i.e. in) the search string; a separate search string is maintained on each level. Various commands set up or modify the search string (as well as search in it); initially upon entering a level its search string is set to null. On a given level there is no access in any way to the search strings of other levels. Associated with each search string are two pointers called $\mathcal{L}$ and $\mathcal{R}$. Each pointer may be between any two characters of the search string, or at either end, subject to the restriction that $\mathcal{L}$ will not be to the right of $\mathcal{R}$. (They may coincide.)

← (left arrow) binary conditional operator search.

(1) The right operand is evaluated. If its value is null: no further action is taken; the value of the left operand is returned as the value of the ← ; no skip occurs.
(2) The search string on this level is set to the value of the right operand.
(3) $\mathcal{L}$ and $\mathcal{R}$ are set together at the left end of the search string.
(4) The value of the left operand is sought in the search string; if it occurs therein more than once, the leftmost occurrence is the one found. If the null string is sought, it is found immediately at the beginning of the search string.
(5) In the search string, $\mathcal{L}$ is placed just to the left of the string found, and $\mathcal{R}$ just to its right. If no matching string was found, $\mathcal{L}$ and $\mathcal{R}$ are placed together at the right end.
(6) The value of the left arrow operator is the search string, i.e. the right operand.
(7) If a match was found, the ← skips; if not, no skip occurs.

:← binary conditional operator. Like ← except skips if match not found, or right operand null.

,← binary conditional operator search right-to-left. Like ← , except: $\mathcal{L}$ and $\mathcal{R}$ are started at the right end of the search string; the rightmost occurrence of the sought string is found.

;← binary conditional operator. Like ,← except has skip sense of :← .

! unary conditional operator continue search. Search string is searched rightward from current position of $\mathcal{R}$ for occurrence of operand. $\mathcal{L}$ and $\mathcal{R}$ are positioned around said occurrence, or at the right end if none was found; skips if found, else not. The null string, if sought, is found immediately. Value is the search string.

:! unary conditional operator, like ! but inverts skip sense.

,! unary conditional operator continue search right-to-left. Search string is searched leftward from $\mathcal{L}$ for occurrence of operand. Other details as for ! .

;! unary conditional operator, like ,! but skips if not found.

PARTS OF THE SEARCH STRING

< item left of $\mathcal{L}$. Evaluates to that portion of the search string left of $\mathcal{L}$.

> item right of $\mathcal{R}$. Evaluates to that portion of the search string right of $\mathcal{R}$.

.= item center. Evaluates to that portion of the search string right of $\mathcal{L}$ and left of $\mathcal{R}$.

<> item left of $\mathcal{R}$.

>< item right of $\mathcal{L}$.

.> item character right of $\mathcal{R}$. Has value of that character, and steps $\mathcal{R}$ to right over the character. (If $\mathcal{R}$ is at the right end of the search string, has null value.)

.>< item character right of $\mathcal{L}$. Has value of that character, does not move pointer.

$.^{n}>$ (n periods, then >) item characters right of $\mathcal{R}$. Has value of n characters to right of $\mathcal{R}$ (or as many as there are, if less than n), and steps $\mathcal{R}$ over them.

n.> (where n is a string of digits interpreted as a decimal integer) like $.^{n}>$ has the value of the n characters to right of $\mathcal{R}$, and steps $\mathcal{R}$ over them.

$\left.\begin{array}{l} .^{n}>< \\ n.>< \end{array}\right\}$ n characters right of $\mathcal{L}$, no pointer motion.

.< item character left of $\mathcal{L}$. Has value of that character, steps $\mathcal{L}$ to left over it.

$\left.\begin{array}{l} .^{n}< \\ n.> \end{array}\right\}$ item characters left of $\mathcal{L}$. Has value of those characters in reverse order (right to left); steps $\mathcal{L}$ over them.

$\left.\begin{array}{l} .^{n}<> \\ n.<> \end{array}\right\}$ n characters to left of $\mathcal{R}$ in reverse order. No pointer movement.

$> unary operator variable characters right of $\mathcal{R}$. If the current value is an unsigned (decimal) number, that number's integer part is taken as n and this operator has the value of n.> and steps $\mathcal{R}$ n characters to the right. If the operand is not an unsigned number, this operator has the value of > , but also places $\mathcal{R}$ at the right end of the search string.

$>< unary operator variable characters right of $\mathcal{L}$. If the current value is an unsigned number, that number's integer part is taken as n and this operator has the value of n.><. Otherwise this operator has the value of >< . No pointer is moved.

$< unary operator variable characters left of $\mathcal{L}$. If the current value is an unsigned number, that number's integer part is taken as n and this operator has the value of the n characters left of $\mathcal{L}$ in forward (left-to-right) order. $\mathcal{L}$ is moved to the left that many characters in the search string. If the operand is not an unsigned number, this operator has the value of < , and $\mathcal{L}$ is placed at the left end of the search string.

$<> unary operator variable characters left of $\mathcal{R}$. If the current value is an unsigned number, that number's integer part is taken as n and this operator has the value of n.<> . Otherwise it has the value of <> . No pointer is moved.

,> item word right of $\mathcal{R}$. (1) If the character to the right of $\mathcal{R}$ is a space, $\mathcal{R}$ is stepped over it. (2) The characters right of $\mathcal{R}$ up to but not including the first space (or the end of the search string) are returned as the value of the ,> and $\mathcal{R}$ is stepped over them.

$\left.\begin{array}{l} ,^{n}> \\ n,> \end{array}\right\}$ item words right of $\mathcal{R}$. Like ,> but gets n words. Step (2) continues up to the nth space.

$\left.\begin{array}{l} ,^{n}>< \\ n,>< \end{array}\right\}$ item words right of $\mathcal{L}$. (1) If the character to the right of $\mathcal{L}$ is a space, that character is ignored. (2) Characters right of $\mathcal{L}$ up to but not including the nth space are returned as the value of the operator. Neither pointer is moved.

$,^{n}<$
n,< } item words left of $\mathcal{L}$. (1) If the character to the left of $\mathcal{L}$ is a space, $\mathcal{L}$ is stepped over it. (2) The characters to the left of $\mathcal{L}$ leftwards up to but not including the nth space are returned as the value of this command and $\mathcal{L}$ is stepped over them. The value appears with the words (separated by spaces) in reverse order, each word spelled forwards.

$,^{n}<>$
n,<> } item words left of $\mathcal{R}$. Like $,^{n}<$ except (a) process starts to left of $\mathcal{R}$; (b) neither $\mathcal{L}$ nor $\mathcal{R}$ is repositioned.

SEARCH STRING MODIFICATION

;= unary operator replace center. Old center of search string (between $\mathcal{L}$ and $\mathcal{R}$) is deleted and operand is inserted there. Value is entire (modified) search string.

; unary operator replace left. Old left part of search string (left of $\mathcal{L}$) is deleted and operand takes its place. Value is new search string.

; unary operator replace right. Right of $\mathcal{R}$ is replaced by operand; value is search string.

:,sym define as search string. The symbol sym becomes defined as a universal symbol with a symbol value identical to the current search string. The positions of $\mathcal{L}$ and $\mathcal{R}$ are also saved as part of the definition. (That would not be the case if the symbol were defined in terms of something whose "value is the search string", since $\mathcal{L}$ and $\mathcal{R}$ are not inherent in that value.) This operator does not affect the current value, or modify the search string.

:;sym level define as search string. Like :, but defines as level symbol.

.:sym item restore search string. The search string is set to the symbol value of the symbol sym. If the active definition of sym was performed by :, or :; then $\mathcal{L}$ and $\mathcal{R}$ are set to their positions saved in the definition. Otherwise $\mathcal{L}$ and $\mathcal{R}$ are set to the left end of the search string. This command has the value of the new search string.

Note: The commands :, and :; provide the only means of saving search pointers in a definition, and .: provides the only way such saved pointers may be accessed. Any use of a symbol defined by :, or :; except by the .: command gets the symbol value as if the pointers were not there.

EXAMPLES: An elementary use of the search string is to edit function definitions.
Suppose .FACT2⇒ ↑+N⌣1;F⌣[⌣N#>∅⌣F)⌣N$*F⌣1:-N⌣]⌣
and ↑+N should be ↑;N instead. To edit this,
"↑+N⌣←.FACT2⌣NO)⌣"↑;N⌣;=,FACT2⇒ ↑;N⌣1;F⌣[⌣N#>∅⌣F)⌣N$*F⌣1:-N⌣]⌣
The ⌣NO)⌣ in case the search fails may if one is confident be condensed to ⌣⌣ .

As another example, to change all instances of the string FOO to BAR in the definition of BLETCH: ←.BLETCH⌣⌣[X"FOO⌣:!"BAR⌣;=]X⌣,BLETCH

INPUT AND OUTPUT

% unary operator print. When performed, causes the current value to be printed out. Returns that as its value.

#% like % but inserts carriage return and line feed in typeout when number of characters on current output line reaches line length.

$? unary operator set line length, for #% operator. If current value is a number not less than 1, line length is set to integer part of that number. Returns current value.

&	item <u>read word</u>. When performed, takes typed-in characters through the first ⌣ (space) and has the value of that string excluding the space.
.&	item <u>read character</u>. Takes any one typed-in character and returns it as value.
#&	like & but ignores any carriage returns and line feeds in the string read in.

ERRORS

A few cases of inscrutability in a string being processed, due most likely to a typing error, for which no particular default action is obvious, cause an error condition: processing is terminated, an error printout occurs, and then the program awaits a fresh typed-in string to evaluate. The error printout has three parts: (a) a code describing the fault detected; (b) the number of the evaluation level where the error occurred; (c) a "backtrace" giving: first, for the top level, the symbol whose evaluation was in progress, with the character preceding it; next, the symbol being evaluated on the second level, etc., through the level where the error occurred, giving for that level the character string which caused the error (or in a few cases, the characters just after those causing the error). In this backtrace, a character whose ASCII code is less than $4\emptyset_8$ is printed as ↑ followed by the character with $1\emptyset\emptyset_8$ added to its ASCII code; except that the character whose code is Ø is not printed at all.

Error Codes:

UCQ	unclosed quote. A " was encountered which had no terminating ⌣ (space).
UCS	unclosed superquote. A [[had no matching]] .
UCP	unclosed parenthesis. A (had no matching) .
SCE	storage capacity exceeded. The typed-in string, symbol definitions, all levels' current values and search strings, and the partial result of the operator being performed, all taken together exceed available memory.
NSD	non-symbol definition. An attempt was made to define something which is not a symbol.
ASD	at sign definition. An attempt was made to redefine the permanent symbol @ .
DIS	decimal point in symbol. The meaningless format sym. was encountered.
PIS	parenthesis in symbol. The meaningless format sym(was encountered.
IOP	illegal operator. A character with no meaning was encountered (not quoted).
IDO	illegal double operator. A meaningless concatenation of two characters was encountered. (Many such combinations, however, are not detected and their first characters are ignored.)
EAS	evaluated argument searches. Some form of search command occurred in a word on this level which is being evaluated by ↑ on a lower level.
ATL	argument on top level. A ↑ was processed on level 1.
UDT	undefined tag. A goto (not a dispatch) referred to a tag not defined in that string.
UUO	unused opcode. Internal error in the STRING program.
POV	pushdown overflow. Usually means maximum level depth exceeded.

Note: The following Error Codes relate to the particular input-output control features of the PDP-6 STRING program.

\\\	G character evaluated.
LIU	(Time-sharing version only) line printer in use, assignment not done.
NFB	no free blocks. DECtape used for output is full.
NFF	no free files. Not possible to initialize DECtape for output.

Error Codes (continued):

FNF	file not found. Input file not found on specified device.
UNA	unit unable. Specified DECtape drive not turned on, not selected, multiply selected, or has no tape.
BDD	bad directory, on specified DECtape.
TMD	too many directories in core. One must be killed to make room for a new one.
UNF	(Time-sharing version only) unflappable. Specified DECtape unit's file directory may not be killed at this time.

INPUT-OUTPUT CONTROL

The following input-output control features are those implemented in the PDP-6 STRING program; where possible they were chosen to resemble those in other MAC PDP-6 system programs. Their usage and effects are the same in both time-sharing and non-time-sharing versions of the program.

$ℓ binary operator _designate input file_. The left operand is examined and the last digit therein is taken as a DECtape unit number. (In the time-sharing version, additionally the unit number Ø refers to device DDØ and 9 refers to DD1.) If there is no digit in the operand, the most recent unit number given is assumed. The value of the right operand is then interpreted as a file name (two subnames separated by a space) according to the algorithm used by MACDMP. Subsequently, when DECtape input is turned on, input characters will come from the designated file. This operator returns the left operand as its value.

$% unary operator _initialize output file_. Gets unit number from left operand as does $ℓ . Returns operand as value. Turns on DECtape output and directs it to specified device.

;% binary operator _close output file_. Ignores left operand except to return it as value. Takes right operand as file name; DECtape output since last ;% or $% is filed with that name.

:% binary operator _delete file_. Gets unit number and file name like $ℓ , and deletes that file from specified device. Value is left operand.

,% unary operator _kill file directory_. Gets unit number, deletes that unit's file directory from memory. (In time-sharing version, this command _must_ be given prior to demounting tape.)

Input characters are taken for the typed-in string to be evaluated, and for performance of ℓ , #ℓ , and .ℓ . They are not echoed until they are taken for one of these purposes by the program. The program provides a large buffer, located between the keyboard and the program, for characters typed in. If a character is typed in when this buffer is full, that character is lost.

Certain characters when typed in on the keyboard are filtered out and are not put into the buffer for the program.

⌊@	ignored
⌊B	"Begin" turn on line printer output
⌊E	"End" turn off line printer output
⌊R	"tape" turn on DECtape output
⌊T	"not tape" turn off DECtape output
⌊Q	"x-on" turn on DECtape input; this character is treated specially. It is put into the type-in buffer, but when taken out of the buffer is performed and not passed on to the program.
⌊S	"x-off" turn off DECtape input.
⌊V	turn on Teletype/user's console output
⌊W	turn off Teletype/user's console output

```
⌊O    "Out" transfers to DDT.
⌊N    "oN"  }
⌊F    "oFf" } control routing of echo output to devices other than user's console
⌊P    control character quote.  Causes the next typed-in character to go into
           the buffer without taking special action.  Exceptions:
                ⌊@     still ignored
                ⌊Q     this is entered in a special way such that it gets passed
                            on to the program without special action
                carriage return    carriage return and line feed are echoed;
                                        nothing is put into the buffer
                ALT MODE/ESC       this is entered in such a way that it will
                                        not terminate the typed-in string
                RUBOUT             just cancels the  P .
⌊G    "bell" quit.  (This makes use of the "console output disable flag" which
           is turned off whenever the program is ready to accept an input string.)
           Characters in the type-in and type-out buffers are all erased.  Then
                (a) if the value of a typed-in string, or an error printout,
                     is currently being output on some device other than, or
                     in addition to, the user's console: echo a backslash on
                     the console, and disable console output (turn the flag
                     on).  When the output is finished, echo another backslash
                     to the console and turn the flag back off.
                (b) if the console output disable flag is on: erase all charac-
                     ters in other devices' output buffers, echo a backslash on
                     the console, and output two backslashes on all devices
                     selected for error output; then turn flag off and await
                     a new typed-in string.
                (c) otherwise, output two backslashes to the console and all
                     devices selected for error output; then await a new typed-
                     in string.
```

Input characters are taken initially from the user's keyboard. When a ⌊Q is read from the type-in buffer, and the $& operator has previously been performed to select an input DECtape file, then characters are taken from that file until the end-of-file is reached or a ⌊S is typed in; then input reverts to the user's keyboard.

Output characters are generated to echo typed-in characters, as the value of the typed-in string, in the performance of % and #%, and for error printouts. Output characters from all sources are routed to devices according to Fig. 1.

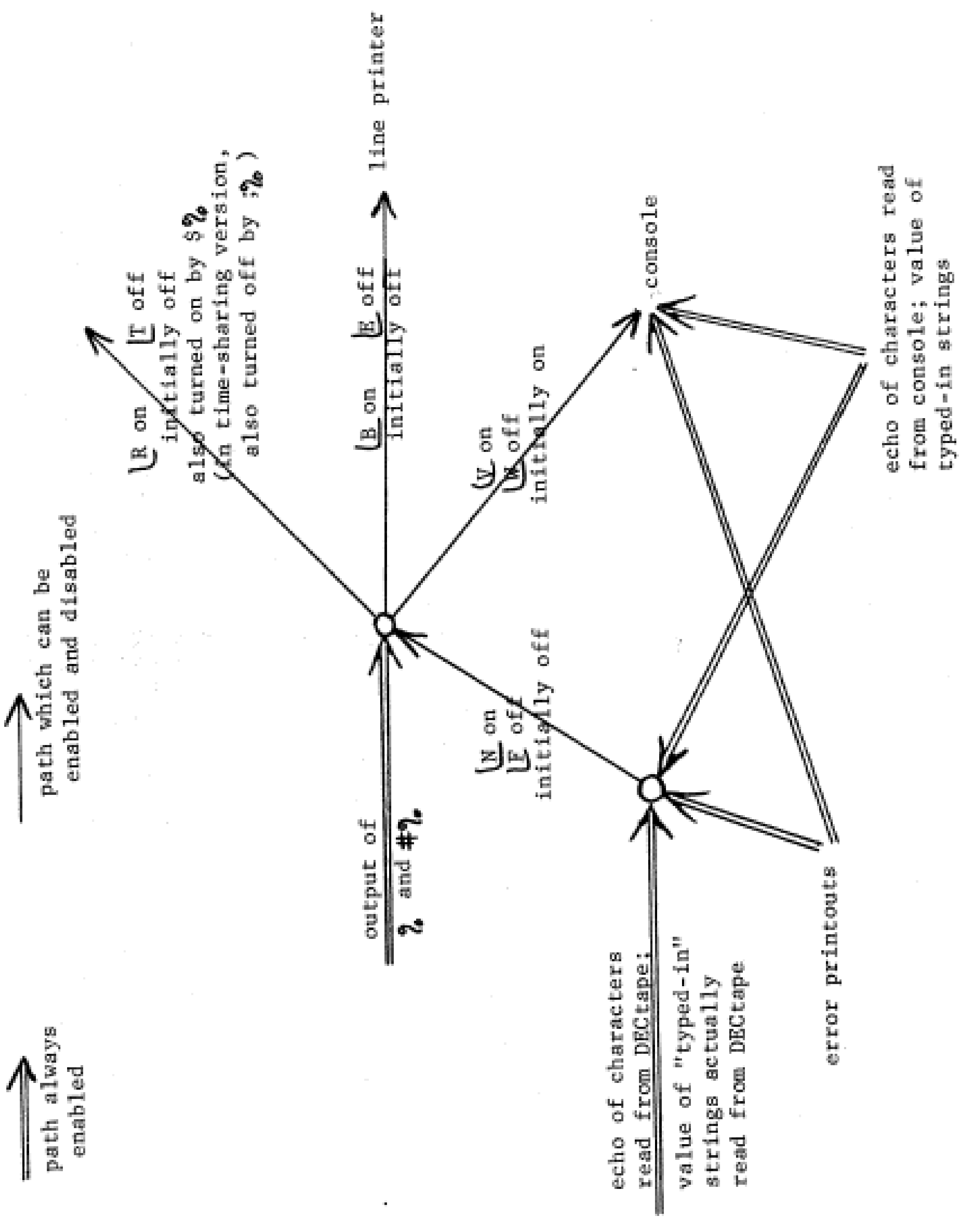
path always enabled
path which can be enabled and disabled
output of ?, and #?,
echo of characters read from DECtape; value of "typed-in" strings actually read from DECtape
error printouts
N on
F off
initially off
R on
T off
initially off
also turned on by $?
(in time-sharing version, also turned off by ;?)
B on
E off
initially off
line printer
V on
W off
initially on
console
echo of characters read from console; value of typed-in strings

Fig. 1. Character Output Paths