UCID - 30100
COMPUTER
DOCUMENTATION

LAWRENCE LIVERMORE LABORATORY
University of California/Livermore, California

TRIX — AN INTERACTIVE, INTERPRETIVE LANGUAGE FOR MANIPULATING STRINGS OF CHARACTERS

Hank Moll

November 16, 1974

Prepared for U. S. Atomic Energy Commission under contract no. W-7405-Eng-48

SUPERSEDURE

This document supersedes the following documents:

Octopus Communique No. 715, September 18, 1973

Octopus Communique No. 888, September 16, 1974

CONTENTS

TRIX – AN INTERACTIVE, INTERPRETIVE LANGUAGE FOR MANIPULATING STRINGS OF CHARACTERS

ABSTRACT

TRIX is a language for manipulating strings of characters. It consists of functions and a syntax for assembling these functions into named procedures called macros. A program written for the TRIX interpreter is called a dialect. The TRIX user writes a dialect (program) by writing a series of macros as a disk file, and then reads-in (interprets) the dialect and executes the macros to carry out operations on a text file. Dialects run under TRIX are used for text editing, where text can be source code (e.g., a Fortran disk file) or English-language text. This report describes in detail the basic TRIX functions, the TRIX syntax, and brief examples of macros, showing applications to text-editing procedures. TRIX is an executable controllee (written in assembly language) available on the CDC 6600 and 7600 computers of the Livermore time-sharing system.

INTRODUCTION

TRIX is designed to manipulate strings of characters. TRIX is interactive and interpretive. That is, commands are entered at a terminal (or from a disk file called a dialect) and are interpreted one at a time as they are received.*

TRIX is available as an executable controllee on the CDC 7600 and CDC 6600† computers of the Livermore time-sharing system. It may be executed by entering the input line TRIX / t v, whereupon TRIX will prompt for input, and the user will interact with TRIX in the pure-TRIX mode. Most often the user will execute TRIX with the input line TRIX AC / t v, which will make available to him not only the basic TRIX commands but also the large number of functions available in the AC dialect.[3]

Command Stack and Storage Stack

Commands received by the interpreter are placed on a push-down stack called the "command stack." The command stack, a sequence of characters, is scanned from left to right. The scanner evaluates the command stack piece by piece, removing each piece in the process. When the command stack becomes null, a command called the "idle macro" is placed on the command stack and scanned. The idle macro will prompt the user for the next command and thereby reinitiate the process.

---

*The concepts underlying TRIX, recursive string language and pattern matching, are described in Refs. 1 and 2. The reader should acquaint himself with the contents of at least these two references before proceeding further.

†Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Atomic Energy Commission to the exclusion of others that may be suitable.

Another push-down stack, called the "storage stack," is used by the interpreter to hold temporary items and various pointers.

Placing a string on the command stack means concatenating it to the left end of the command stack. Thus it is next in line to come under the scanner. Placing a string on the storage stack results in the string being stored in the "list space" and a pointer to it pushed down onto the storage stack.

The command and storage stacks are programmed devices that simplify internal processing, and conceptual devices for explaining the recursive process fundamental to the interpretation of the commands.

Macros and Data Strings

TRIX is a macro language. That is, procedures may be assigned names, and these names may appear in commands to invoke (call) the procedures. We call these named procedures "macros." Macros are usually made up of TRIX commands and therefore may invoke other macros. In particular, a macro may invoke itself. This is called recursion, and it is the basic and most important concept in the language.

Invoking a macro means placing its value (expansion) on the command string, thus bringing it under the scanner.

Macros can be viewed as user-defined functions. They augment the 40 or so system-defined functions described in Appendix B.

In general, any named string is called a macro. The string itself is called the macro body. At naming time, certain substrings of the macro body may be defined as parameters to be replaced by arguments at evocation time.

Distinguishing between Macros and Data Strings

TRIX commands are composed of strings of characters. How does the scanner distinguish which substrings are macros and which are data strings? That is, which are verbs and which are nouns?

In general, data strings are quoted with angle brackets (<>). This requirement is awkward and in many cases unnecessary since the context in which the strings appear in the command provide the same information as the brackets would.

The context is supplied by a format imposed upon a command (when it is defined) and by the use of punctuation marks (e.g., commas) as delimiters and/or operators. The rules of context provide for defaults in the command expression that simplify usage at the expense of making the initial learning process more difficult.

Example

The user inputs the following series of commands to TRIX and follows it with a linefeed:

XYZ=<HOORAY>↑ ABC=< #PS($XYZ)>↑ ABC↑

This defines XYZ to be the name of the data string HOORAY, defines the user-defined function ABC as the data string, #PS($XYZ), and invokes the function ABC.

The scanner, starting from the left, recognizes XYZ as a token, removes it from the command stack, and stores it on the storage stack. The following equal sign (=) identifies XYZ as a macro name. The equal sign is also recognized as the left delimiter of the replacement field and is removed from the command stack. The < is recognized as an open quote and is removed. The string up to the matching close quote, >, is taken as a literal, and is removed and stored on the storage stack, and the close quote is removed. The ↑ is recognized as the right delimiter of the replacement field and is removed.

The scanner stops at this point and calls the macro-definition subroutine, which removes the name and body from the storage stack and makes an internal assignment, adding the macro to the list space, and returns to the scanner.

The scanner recognizes ABC as a token, removes it from the command stack, and follows the procedure described above.

We have now defined two macros, XYZ and ABC, and the scanner is positioned at the space preceding the final ABC. The scanner sees and removes the space and sets the value flag to "command." The scanner recognizes the ABC as a token and stores it on the storage stack. The following ↑ flags the end of the command.

The scanner calls the evaluator, which looks at the token on the top of the storage stack, recognizes it as a macro name, and calls the macro expander. The expander expands macro ABC and, finding the value flag at "command," pushes the value of ABC onto the command stack, concatenating it to the ↑ we left there. The scanner is now looking at the # of the #PS; it recognizes #PS as a function, removes it from the command stack, and stores #PS on the storage stack.

The (is recognized as a begin-argument string and is removed. The $ is removed, and the value flag is set to "data." The token, XYZ, is removed and placed on the storage stack. The) is recognized as "argument complete" and calls the evaluator.

The evaluator removes the top token from the storage stack, identifies it as a macro name, calls the expander (which expands the token), and (finding the value flag set to "data") pushes the expansion onto the storage stack. The argument has now been evaluated and is the string HOORAY.

The scanner, which is still positioned at the), recognizes the) as "argument string complete," removes it, and calls the evaluator. The evaluator removes the top token from the storage stack, which is #PS(HOORAY), recognizes it as a function-argument string pair, and calls the #PS function. The #PS function prints the string HOORAY and returns a null value.

At this time, the scanner, finding the command stack empty, sets the value flag to "command," and calls the expander to expand the idle macro. The idle macro is placed on the command stack, scanned, evaluated, etc., and it prompts the user for the next command.

Figure 1 shows this process. When an #ED precedes the input line, the sequence of operations is typed on the teletypewriter.

```
TRIX AC / 1 1
.XYZ=<HOORAY>↑ ABC=< #PS($XYZ)>↑ ABC↑
HOORAY.
.#ED XYZ=<HOORAY>↑ ABC=< #PS($XYZ)>↑ ABC↑
=HOORAY
XYZ
= #PS($XYZ)
ABC
ABC
XYZ
#PS(HOORAY)
HOORAY.
.END

 ALL DONE


TRIX / 1 1

OK
XYZ=<HOORAY>↑ ABC=< #PS($XYZ)>↑ ABC
HOORAY
OK
#ED XYZ=<HOORAY>↑ ABC=< #PS($XYZ)>↑ ABC↑
=HOORAY
XYZ
= #PS($XYZ)
ABC
ABC
XYZ
#PS(HOORAY)
HOORAY
OK
#EX

 ALL DONE
```

Fig. 1. Executions of the example under both TRIX and TRIX AC.

Dialects

A dialect is a set of macros (named procedures). Dialects thus consist of user-defined functions. Each user-defined function consists of a name (the macro name) and its definition. The set of macros in a dialect performs specific tasks; the set of macro names form a tongue sufficient to describe these tasks. Writing dialects is central to using TRIX.

Dialects may be kept on disk and loaded at run time, thereby obviating the necessity to redefine the macros each time TRIX is executed. Instead, one uses the TRIX system macro #RD or the TRIX AC macro RD to read the dialect from a disk file. In the above example we made use of the TRIX AC macro END to end the execution of TRIX. The following sequence shows the definition and use of a disk-file dialect called TEST.

```
TRIX AC / 1 1
.C(TEST) O(TEST) AL0
0 LINES. (80A)
&XYZ=<HOORAY>↑
&ABC=< #PS($XYZ)>↑
&.
.NF
2 LINES. (80A)
.T
1 XYZ=<HOORAY>↑
2 ABC=< #PS($XYZ)>↑
.END

 ALL DONE
FILES / 1 1
      27R TEST

 ALL DONE
TRIX AC / 1 1
.RD(TEST)
.ABC
HOORAY.
.END

 ALL DONE
```

Strings

A string is any sequence of characters. A named string is called a macro. Strings contained between angle brackets <> are called literal strings. Literal strings may contain angle brackets in matching pairs. Unmatched angle brackets within a literal must be written $< or $>; the $ will be deleted. Angle brackets protect the literal string from evaluation. Back-slashes appearing in a literal string must be written $\. Dollar signs, standing for the character $, may appear in literal strings. In cases of ambiguity, (namely, immediately preceding an angle bracket) use a double dollar ($$).

The operation of concatenation between strings is implied by juxtaposition of the strings in the command. Unquoted blanks are ignored, ^A^B^C^ (where ^ indicates the space bar) concatenates to ABC. <^A^><^B^><^C^> concatenates to <^A^^B^^C^>. Note that <^$^> is the three-character string ^$^, while <$$> is the single character $. Note that <> is the null string.

String Names

A string name is composed of 10 or fewer alphabetic or numeric characters and may appear anywhere in a command. There must be at least one leading alphabetic

character. The numeric characters, if any, are the rightmost characters of the name. Medial blanks are not allowed. Note that "string name" is just a term used to describe any alphameric string of less than 11 characters. It may or may not be the name of some actual string. If it is the name of some string, it is also called a macro, otherwise, it may be referred to as an undefined string name.

The string %X, where X is an alphameric string name less than 10 characters long, is also a string name. Names of this type are used internally in dialects, thereby freeing for external use the pure alphameric form. Three-character string names of the form %AB, where A and B are alphameric, are reserved for use by the interpreter. The following are string names: ABC, A1, %XX, and %A1. The following are not: A^C, 123, 1A, and ABCDEFGHIJKL.

Function Names

A function name consists of two alphabetic characters preceded by a # character and may appear anywhere in a command. Again, a function name is a term used to describe strings of this form. Not all strings of this form have corresponding functions defined; however, all functions have names of this form. Functions are predefined and cannot be modified by the user. Appendix B describes the TRIX functions.

Command Format

A command has three fields: subject, pattern, and replacement. The subject field is always the first field. There may be several pattern and replacement fields, or either or both may be null. The back arrow (←) signals the beginning of a pattern field. The equal sign (=) signals the beginning or continuation of a replacement field. An up arrow (↑) terminates the command, and each command must be so terminated. Several commands may appear on the same input line, and commands can be continued to the following line(s). A back-slash (\) signals the beginning of a comment field, which continues to the linefeed and is ignored. Dialect commands are disk images of terminal commands. The end-of-card string corresponds to the linefeed. The bell will ring when more input is needed to complete a command, as is the case when a command is continued through several input lines or when the terminal ↑ is left off.

The command format implicitly calls for the replacement, in the subject, of a specified pattern. For example, S←P=R↑ reads: Replace the first occurrence of pattern P in subject S with R. S=R↑ reads: Replace S with R, or assign the name S to string R. S←P↑ reads: Find the first occurrence of pattern P in subject S. S↑ reads: Execute command S. In the above forms, S is called the subject field, P is called the pattern-field, and R is called the replacement field. These fields may contain any sequence of functions, macro calls, literal strings, text references, or algebraic expressions. The last two entities will be defined later.

There may be several pattern and/or replacement fields. For example, S←P1=R1←P2=R2↑ reads: Search subject S for pattern P1 and replace with R1, and search subject S for pattern P2 following pattern P1 and replace with R2. Likewise, S←P1←P2←P3=R1↑ reads: Search subject S for pattern P1, search subject S for pattern P2 following pattern P1, and search subject S for pattern P3 following pattern P2, and replace patterns P1, P2, P3 with R1.

The expression MAC=<XYZ>↑ reads: Assign the name MAC to string XYZ. It is a macro definition. The expression MAC(<Y>)=<XYZ> reads: Assign the name MAC to the string XYZ with parameter Y. Macro MAC may be called with an argument that will replace the substring Y in the string XYZ.* For example:

HELLO(<JOHN DOE>)=<MY NAME IS JOHN DOE.>↑

becomes, when called by the expression ↑HELLO(<JOE DOAKS>)↑,

MY NAME IS JOE DOAKS

This example turns over a rock, beneath which one will discover many fascinating subtleties.

---

*The function #NS (see page 24) can also be used to define macros. It does not allow for the assignment of parameters as arguments.

MACROS

A macro is a named string of which certain substrings may be designated as parameters. A macro name has the same form as a string name. A macro is simply a device to simplify procedures. Any named string is a macro, even if parameters are not designated.

Defining a Macro

A macro definition consists of the macro name in the subject field followed by a string called "the macro body," in the replacement field. The pattern field is null. The substrings to be designated as parameters are separated by commas, enclosed in parentheses, and immediately follow the macro name. The parentheses may be omitted only if these argument strings are null. Evaluation of the parameters takes place before the macro is defined.

Any command consisting only of (1) a subject that evaluates to a string name and (2) a replacement, is a macro definition. If a macro already exists by this name it is replaced by the new one. The #NS function can be used to define parameterless macros. In either case, a pointer is associated with the macro and is set to one. The nth character of a string is said to follow pointer position n.

Calling a Macro

A macro call is a reference to a previously defined macro. Arguments to replace parameters are separated by commas, enclosed in parentheses, and immediately follow the macro name. If the argument string is null, the parentheses may be omitted. If the arguments are numeric and the macro name does not end with a numeric, the parentheses may be omitted. Extra arguments are ignored, and missing ones are assumed null. Macro-call arguments are evaluated before they are inserted into the macro body.

Naming Macros

Names ending with a number are ok but may cause confusion. Suppose a name ends with a number, say AB7. In this case 7 is part of the name. But the string AB7 could also be a call for a macro named AB, with the 7 being a numeric argument, which need not be enclosed by parentheses. The user can either avoid these situations or take advantage of them, providing he knows how they are handled by the interpreter, and that is thus: If the name of a previously defined string is equal to the alphabetic (left) part of this name, then any numeric (right) part will be treated as a numeric argument. Otherwise, both parts together form the name.

System Macros; Treatment of Errors

Errors arising during the operation of TRIX will result in the insertion and evaluation of the appropriate error macro at the current position in the command. These macros can be redefined by the user. Therefore the user can tailor error response to his particular needs. In most cases the system macro simply prints a message describing the error. There are a number of error conditions, each with a unique name, which is also the name of the error macro. The user can redefine a system macro with the regular macro definition command or the #NS function. Parameters cannot be assigned for system macros. The system macros are listed in Appendix A.

FUNCTIONS

A function name consists of two alphabetic characters preceded by a #. Its arguments, separated by commas and enclosed in parentheses, immediately follow. If the argument string is null or numeric, the parentheses may be omitted. A function may appear in any field of a command, and a command may consist of a function only. Function arguments are evaluated before they are used by the function. Extra arguments are ignored, and missing ones are assumed null or zero unless stated otherwise. The functions are listed in Appendix B.

TEXT

A text is a rectangular array of characters contained in a disk file. Text files fall into three categories: line-oriented text, such as input to compilers, word-oriented text such as reports and documents, and a third group consisting of text files not in the above two groups, for example, binary files. Text files are also categorized by line length and the packing device used. Text files "opened" by TRIX are mapped into a uniform internal format designed to render subsequent editing simple and efficient. Files to which changes have been made are mapped back to their original format upon termination.

Text Lines

A line in the internal format is a string of m characters followed by n blanks, where $m+n$ is a multiple of 10, and n is less than 10 and may be zero. Notice that each line occupies an integral number of machine words. This device simplifies the finding of the location of specific lines within the text file since their location may be computed instead of searched for. m is called "the number of characters per line," or "the length of the line." A text in the internal format is a sequence of disjointed, equal-length lines.

Text References

A text reference is a reference to any portion of the text and may appear anywhere in a command. Listed below are some of the special forms a text reference may take. Below, m and n are integers, or are strings that evaluate to integers.

Line-Number Formats

n;m refers to lines n through m inclusive. If n > m then n + m replaces m in the expression.

n; becomes n;n and refers to line n, as does n;0.

;n refers to the next n lines.

Special Symbols

[evaluates to the current line number. [is the line cursor. After each text reference, [will evaluate to the next line following the reference. [is interpreted as a numeric character.

] evaluates to the last line number of the text. [is constrained to be less than or equal to].] is interpreted as a numeric character.

[] refers to the entire text, while [;] is the text from the current line thru and including the last line.

In the subject field, a text reference serves to delineate an element of text as the subject for the pattern match. In the pattern or replace fields, or in an argument string, a text reference is replaced by the text string it refers to and is treated as a literal string.

STRING ARITHMETIC

Strings can be manipulated by "arithmetic" operators. We treat three operators: the semicolon (;), the colon (:), and the apostrophe ('). The operands are strings or arrays.

The use of ; (to specify lines) has been discussed in the preceding section, Text References.

The use of : (to specify columns) will be discussed in the following section, Arrays.

The use of ' (to specify blank-filling) will be discussed below under Tabs.

Once one has specified lines, columns, blanks, etc., he can concatenate them together to form new strings. (See Concatenation, below.)

Typical applications appear in Appendix C, Example 4. We leave it to the reader to read the following subsections and then study the possibilities of string arithmetic on his own, using Example 4 of Appendix C as the jumping-off point.

Arrays

An array is a pair consisting of a string and a dimension c, called the number of columns per row. It will be useful to deal with strings as arrays and vice versa. Text, as defined, is an array of c columns and r rows, where c is the number of characters per line and r is the number of lines in the text. A string can be thought of as a one-row array. Since arrays are stored as strings, their only use is to generate strings, but that's useful enough. Arrays are particularly useful as arguments of the output functions, #PS and #DS. We introduce the following notation:

The string n:m, where n and m evaluate to positive integers, will format a preceding array into an array consisting of columns n thru m inclusive. Several strings of the form n:m may appear sequentially, the resulting arrays merging to form one array. For example:

```
ABCDEFGHIJ 1:2 7:8
```

Selects columns 1 to 2 and 7 to 8 and evaluates to:

```
ABGH
```

The string :m, where m evaluates to a positive integer, will format a preceding string or array into an array of m columns taken from the string or array, m characters at a time, blank filled if necessary. The form :m is equivalent to m:. For example:

```
ABCDEF :2
```

Evaluates to:

```
AB
CD
EF
```

The colon by itself, properly delimited, will format a preceding string into a one-row array and a preceding array into a string.

Arrays, formed by these three methods, that appear sequentially in a command will be merged into one array. Some arrays may be blank-filled from the bottom to make them row equivalent to the final array. The arrays so formed replace what they were formed from in the command and are not evaluated further.

Concatenation

If after evaluation two strings appear sequentially in a command they will be concatenated to form one string. Likewise, after evaluation two adjacent arrays will be merged to form one array. Finally, after evaluation a string adjacent to an array will be treated as a one-row array and merged with the array. In this case it will be extended by itself to make it row equivalent to the array.

Tabs

Strings of the form n' or 'n, where n evaluates to an integer, serve as tabs. The value is a string of blanks sufficient to space from the current character position in the preceding literal string or array, to the nth position (in the case of an array its row length is extended to n characters). If the preceding literal string is null, then the value is n blanks. If the current character position exceeds n-1, then the value is null. A single quote all by itself, evaluates to a string of blanks from the current character position to the end of the line. In this respect, ' serves as the end-of-line flag.

ARITHMETIC EXPRESSIONS

An arithmetic expression may appear in any field of a command, either independently or as an argument. It may be a combination of items from the following four groups: (1) paired parentheses, (2) numbers, (3) functions or macros that evaluate to numbers, and (4) the symbols + - * / " and ;. The arithmetic expression must, of course be algebraically meaningful. Its value is the algebraic result of the arithmetic operations. Medial blanks are not allowed in an arithmetic expression. Blanks and all meta characters except ()[]#% \ serve to delineate arithmetic expressions in a command.

A number is defined to be 9 or fewer decimal digits and may be preceded by a sign. Normally the expressions are evaluated in an integer mode. However, a mode is provided that allows decimal points, recognizes the notation +n.nnnnE+nn, and does decimal arithmetic. See functions #ED8 and #ED9 on page 38.

COMMANDS

Command Field Functions

Text replacement, insertion, and pattern matching is accomplished as in SNOBOL by the command format. The command S←P=R↑ reads: Replace the first occurrence of the pattern P in subject S with replacement R. The subject field must evaluate to a text reference. (In the future, string names will also be allowed.) The subject is the portion of the text file that is to be searched for the pattern. The replacement field must evaluate to a literal string. The pattern field describes a selection process,

not just a literal string to be matched. The pattern field usually contains information as to how to proceed after the match failed or succeeded. The pattern field is evaluated from left to right. When the scanner encounters an ↑, ←, =, or #SF function, the subject is searched for a string as described by the evaluated part of the pattern field. This will generate a success or fail condition that can be tested by the #SF function. The scanner will then continue evaluating the command string where it left off.

Following a successful pattern match, [will evaluate to the line number of the line containing the first character of the matched pattern, system macro %CN will evaluate to the column number of the first character of the matched pattern, and system macro %PN will evaluate to the entire pattern matched. [is not changed, and system macros %CN and %PN are set to the null string, when a match fails.

An example follows:

```
TRIX AC / 1 1
.O(AA,10)!T
2 LINES. (10A)
1     1234567890
2     ABCDEFGHIJ
.1,2←DEF=XYZ↑
.#RT([)↑
2
.#RT(%CN)↑
4
.#RT(%PN)↑
DEF
.T
1     1234567890
2     ABCXYZGHIJ
.END

 ALL DONE
```

Here, the user opened a file AA, typed it out, and then searched lines 1 through 2 of this file for the first occurrence of pattern DEF, asking that if the pattern was found that it be replaced with XYZ. Using the #RT system function, he has typed out the values of the line at which the pattern was found ([), the column in which it began (%CN), and the pattern that was replaced (%PN). These values are 2, 4, and DEF respectively. Then he typed out the modified file to show that the replacement did indeed take place.

The text is edited according to the context of the command. The subject is searched for the pattern, which if found, is replaced by the string in the replacement field. If the replacement string is null the pattern string is deleted from the text. Sometimes it is desired to insert the replacement string either before or after the pattern. This is indicated by immediately preceding or following the back arrow, which

marks the pattern field, with a comma. The rule is: If the comma precedes the back arrow, the replacement string is inserted before the pattern; If the comma followed the back arrow, the replacement string is inserted following the pattern string.

Any command consisting of: (1) a subject that evaluates to a text reference, (2) a null pattern, and (3) a replacement, is interpereted as having a pattern equal to the subject. In this case, since the pattern is the subject, no match is required to locate the position for replacing the pattern. This device will simplify edit input. For example, the commands to interchange lines m and n are: m;=n;↑ n;=m;↑.

In all cases, to promote efficiency, the text file is not updated until an update function is encountered by the processor. This means you cannot make changes to changes without an intervening update. Should this situation occur unintentionally, only the first of the overlapping changes is made. The following functions update the text file: #FU, #FC, #RP, and #EX.

Pending updates are saved in core. This introduces a limitation since core is not infinite. About 5000 replacements can be held in core providing the user employs the device of "continuing" the replacement field. An unquoted equal sign in the replacement is interpreted as signaling that the replacement string up to this point is complete and may be pushed onto the replacement queue. Queues, being open ended, imply that the replacement string, if continued, is also open ended.

According to the definition, a text is a string with a line structure imposed upon it. This line structure is invariable under the editing operations. After a text string is deleted and the replacement string inserted, any fractional lines that result are filled out with blanks. Sometimes the inserted string will cause non-blank characters to extend beyond the end of the line. These extra characters are truncated, blank extended and form a new line following the original one. If "text" mode, as opposed to "line" mode (see #ED option 17) has been selected, the truncation process backs up to a blank, thereby ensuring that words are invariant, that is, not split by the editing process.

Command Evaluation

A command consists of literal strings, string names, functions, text references and algebraic expressions, all embedded in a context provided by the field format and the meta characters (see Summary of Punctuation). Each command is scanned from left to right. Starting from the left, each part of the command is removed from the command string and evaluated. Depending upon the context, its value is either active or neutral, that is, it is either pushed onto the command stack or pushed onto the storage stack. If it is pushed onto the command stack it will become the next part of the command to be scanned. On the storage stack it is simply stored for possible future internal reference. This process is repeated until eventually the command string becomes null. When this happens the idle macro is pushed onto the command string. It will load in or prompt for the next command.

Text references, literal strings, and integers always have a neutral value and are placed on the storage stack. Algebraic expressions always have an active value, and this value replaces the expression in the command. For example, 12*(1+3); evaluates to the command 48; and eventually refers to line 48. The disposition of the value of string names and functions is more complicated, but most important. Read the next six paragraphs very carefully.

The value of a function is active if the # of the mnemonic is immediately preceded by a blank character, or if it appears in a context that is constrained to be numeric (for example, in an arithmetic expression or as the numeric argument of some other function or operator). Suppose ABC is the name of a 10-character string. Then the command #SP(ABC,#GC(ABC)), with #GC constrained to be numeric, evaluates to #SP(ABC,10), which evaluates null after setting the pointer of macro ABC to 10.

The value of a function is neutral if the # of the mnemonic is immediately preceded by a dollar sign, or in all other cases not specifically mentioned. Going back to the above example, the value of function #GC would still be active in the command #SP(ABC,$#GC(ABC)), since a numeric context takes precedence.

Every string name has a value. The value may be the string name itself, taken as a literal, or it may be, in the case where the string name is a macro, the body of that macro.

An undefined string name has itself as a value. Its value is always neutral. Undefined string names appear in commands either by mistake or as intended literals, the quotes not being necessary. An undefined string name appearing in a numeric context is an error and will be flagged by the scanner.

The value of a defined string name, a macro, is the string consisting of the macro body with the appropriate parameter substitutions. This value is active if the name was immediately preceded by a blank character, began a command, or appeared in a numeric context. The value is neutral if the name was immediately preceded by a dollar sign, or was immediately followed by an equal sign, or in all other cases not specifically mentioned above.

For example: We have two macros A and B, where A=<B> and B=5. Macro A appears in a numeric context in the command: A;. Hence its value, the one-character string B, is active and replaces it in the command, which becomes: B;. For the same reason, the value of macro B, the one-character string 5, replaces string name B in the command. The command now reads, 5; and is in its final form, referring to line 5.

The above rules appear awkward and complicated. They are. But they admit an economy that will be appreciated when you write dialects.

The Use of Blanks and Dollar Signs

A string of unquoted blanks is equivalent to a single blank. Blanks serve as delimiters and can be used to improve readability. Blanks also serve to indicate context. A blank immediately preceding a function or macro call assigns an "active"

context to the resulting value. In this context, the blank provides an active indirect reference. Blank was chosen for this role because in most cases it has the side effect of improving readability. This is true since the "neutral" occurrence of functions and macros is usually limited to argument strings where the function or macro is immediately preceded by a comma or left parenthesis. A blank here would not enhance readability as much.

The dollar sign serves to indicate context. A dollar sign immediately preceding a function or macro call assigns a "neutral" context to the resulting value. A dollar sign preceding a macro name provides a neutral indirect reference to the macro body. Should the macro body itself be a macro name, a double dollar, $$, provides an indirect reference two levels deep. In general, a string of n dollar signs provides an indirect reference n levels deep.

Command Interrupt

TRIX may be interrupted at any time, under any circumstance, by entering (CTRL-E)I. The command and storage stacks will be cleared and the idle macro will be loaded. Another way to achieve the same result is to break, set the program pointer (P-counter in exchange package) to 4, and restart %ATRIX.

SUMMARY OF PUNCTUATION (META CHARACTERS)

<>	contains and inhibits evaluation of a literal string
()	contains an argument string
←	signals the beginning of the pattern field
=	signals the beginning or continuation of the replacement field
\	signals the beginning of the comment field
↑	terminates a command
.	is the decimal point in decimal mode. See #ED8 and #ED9
;	text-line operator
,	separates arguments within parentheses and, in conjunction with the back arrow, signals text insertion
#	begins a function name
%	begins a system macro name
"	evaluates to a linefeed, carriage-return
?	evaluates to an end-of-message
[]	refers to the entire text file
:	column operator
'	tab operator
$	When it precedes a string name, it results in the value of the string name being pushed onto the storage stack. It is the escape character in a literal string for unmatched angle brackets, for \, and for itself when necessary.

Blank When preceding a string name, a blank results in the value of the string name being pushed onto the command stack. It may not be contained in an arithmetic expression or text reference. Otherwise, it may be used anywhere to improve readability. A sequence of blanks reduces to a single blank except in a literal string. Blank has a null value.

Other All other punctuation marks are ignored.

In the above discussion, the term "string name", refers in particular to macro calls and functions. As long as the correct context is maintained, literals, functions, macros, text references, arithmetic expressions, and other commands or parts of them may appear anywhere in any capacity in a command. The only exception is that the functions #VS, #CR, #MC, #AL, and #TS are ignored unless they appear in a pattern field.

FILE MANIPULATION

When a text file is opened, its contents are copied into a working file. The format of the working file is chosen to simplify subsequent editing procedures, and it usually differs from that of the original file. All changes requested for the original file are made to the working file. The original file remains untouched until the user requests that the working file be repacked to the original format and replace the original file. The working file remains as the working file for the newly updated original file. In practice the original file is updated only at termination; thus it remains unchanged for a possible restart should some drastic error occur during the editing procedure.

The editing commands of the interpreter recognize only one file format, that of the working file. However, the "open" and the "repack" commands recognize four different formats: Monitor, Squoze, ASCII, and other. Thus, in practice, any file, regardless of format, may be edited.

Selected information concerning the opened text file is kept in a list. Thus the costly I/O involved in opening a text file can be avoided on subsequent reopenings of the same file, since the necessary pointers and associated information may be retrieved from the list.

Numerous files may be open simultaneously. However, there is only one I/O buffer, and that is associated with the currently opened text file, i.e., the file named in the most recent "open" command. All editing functions of the interpreter operate on the currently open text file. At termination time, all working files to which changes have been made are repacked and may replace their corresponding original text files.

With one I/O buffer, merging information from several text files can be inefficient. However, this problem can be avoided nicely by using the #WQ function and merging the selected text into an open-ended queue. The queue buffer resides in list space and automatically spills over onto disk as it fills. Since a queue is a potential disk file, the queue name must not conflict with any existing file name. The two queue

functions, #WQ and #RQ, are designed to merge text-file data simply and efficiently, and should be used whenever possible.

Creating a text file results in an empty file by that name being created. This empty file is then opened; hence all of the above pertains. A newly created file becomes the currently opened text file. Notice that text added to a created file is actually added to the working file and won't show up in the original until a repack is issued or TRIX is terminated. Numerous files may be created, but only one, the currently open text file, can be accessed.

APPENDIX A

SYSTEM MACROS

Error Macros

When certain error conditions are detected, the appropriate error macro is pushed onto the command stack. The error macros may be redefined by the user.

```
%XA=<#PS("<ILLEGAL CHARACTER IN ARITHMETIC EXPRESSION          >"?)#SK>
%XB=<#PS("<DISPLAY NOT RESERVED OR AVAILABLE                   >"?)#SK>
%XC=<#PS("<ISOLATED RIGHT ANGLE BRACKET ENCOUNTERED            >"?)#SK>
%XD=<#PS("<HAVE EXCEEDED MEMORY LIMIT                          >"?)#SK>
%XG(X)=<#PS("<CANT OPEN OR CREATE FILE X                       >"?)#SK>
%XH=<#PS("<ILLEGAL FORMAT DIMENSION                            >"?)#SK>
%XI=<#PS("<MISSING RIGHT PARENTHESIS, WILL SUPPLY ONE.         >"?) ) >
%XJ=<#PS("<TOO MANY DIGITS IN INTEGER INPUT                    >"?)#SK>
%XK=<#PS("<UN-PAIRED OR MISSING PARENTHESIS                    >"?)#SK>
%XL=<#PS("<ILLEGAL FUNCTION NAME                               >"?)#SK>
%XM=<#PS("<TOO MANY CHARACTERS IN STRING NAME                  >"?)#SK>
%XN=<#PS("<DISK READ ERROR                                     >"?)#SK>
%XO=<#PS("<DISK WRITE ERROR                                    >"?)#SK>
%XP=<#PS("<NON-NUMERIC FUNCTION ARGUMENT                       >"?)#SK>
%XQ=<#PS("<DIVIDE BY ZERO                                      >"?)#SK>
%XR=<#PS("<ILLEGAL TAB DIMENSION                               >"?)#SK>
%XS=<#PS("<ILLEGAL SUBJECT FORMAT                              >"?)#SK>
%XT=<#PS("<ILLEGAL LINE NUMBER                                 >"?)#SK>
%XU=<#PS("<CANT OPEN DIALECT FILE                              >"?)#SK>
%XV=<#PS("<PATTERN FIELD ERROR                                 >"?)#SK>
%XW=<#PS("<MACRO CALL LOOP                                     >"?)#SK>
%XX=<#PS("<UNDEFINED FUNCTION ARGUMENT                         >"?)#SK>
%XY=↑ (Is called by the (CTRL-E)I interrupt.)
%YA=<#PS("<UNDEFINED MACRO AS SUBJECT                          >"?)#SK>
%YC(Z)=<#PS("<Z OCCURRENCES OF OVERLAPPING PATTERNS DURING UPDATE  >"?)>
%YD(Z)=<#PS("<Z LINE OVERFLOWS DURING UPDATE                   >"?)>
```

Notice that a string of blanks and right parentheses followed by an up arrow is equivalent to an up arrow, provided that the system error macro %XI is changed to read: %XI=<)>.

Idling Macros

%RT=< #RT("<OK>")>

%RD=< #RD>

Other Useful Macros

%00	evaluates to the null string
%12	evaluates to the string 0123456789
%AB	evaluates to the string ABCDEFGHIJKLMNOPQRSTUVWXYZ
%AT	evaluates to the AT character @
%CN	evaluates to the column number of the matched pattern
%EP	evaluates to exclamation point !
%GC	evaluates to the value of the latest #GC function
%GS	evaluates to the value of the latest #GS function
%FU	evaluates to the name of the last working text file
%ID	evaluates to the date of the current version of TRIX
%IM	evaluates to the current idle macro: %RT or %RD
%ME	evaluates to the name of this drop file
%NA	evaluates to the string "#$%&'()*+,-./:;<=>?[\]↑←!
%PN	evaluates to the matched pattern
%SK	evaluates to the skipped command string
%SM	evaluates to the value of the latest #SM function
%SP	evaluates to the value of the latest #SP function
%BY(A)	The moses macro. String A evaluates to a valid execute line and is run as a controllee with all bypasses open. It can be used to make a copy of the executing TRIX for later recovery, should that be necessary. For example, %BY(<COPY >$%ME< x / t v>)↑ will copy the current running TRIX to a file named x. If arguments t and v are omitted, the current values will be used.

APPENDIX B

FUNCTION DEFINITIONS

In the definitions given in this appendix, the arguments to those functions are identified as letters. Function arguments are separated by commas and enclosed in parentheses, and they immediately follow the function mnemonics. If the argument string is null or numeric, the parentheses may be omitted. All unquoted functions are evaluated. The value of the function is pushed onto either the storage stack or the command stack, depending upon the context in which the function appears in the command. A function has a null value unless stated otherwise.

Although the function arguments are given in the definitions as letters (A, B, C, etc.), for these letters you will substitute macro names, literal strings, arithmetic strings, numbers, other letters, or other TRIX functions. The items you substitute may or may not be preceded by $ signs or spaces, and may or may not be quoted with angle brackets.

The decision as to the exact form in which to type an argument depends on a clear understanding of your algorithm and the rules of TRIX, some of which are given on page 10. In addition, there are rules of context (mentioned on page 1) that make it possible, for example, to leave off the angle brackets on literal strings in some contexts. Figure 2 gives some examples that, if studied carefully, will demonstrate some of the rules. Take heart; most of the rules become clear after a little practice at the teletypewriter.

```
TRIX AC / 1 1
.#NS(Y,1)
.#NS(X,Y)
.#PS(X)
X.
.#PS($X)
Y.
.#PS($$X)
1.
.#PS(X+1)
2.
.#PS(X+Y)
2.
.#PS($X+$Y)
2.
.#NS(Z,X+Y+1)
.#PS(Z)
Z.
.#PS($Z)
3.
.#NS(Q,<X+Y+1>)
.#PS($Q)
X+Y+1.
.#NS(W,$Q)
.#PS($W)
X+Y+1.
.#PS($$W)
3.
.Q=< #PS(1)>↑
.#PS($Q)
 #PS(1).
.#PS( Q)
1.
.END

 ALL DONE
```

Fig. 2. Demonstrating the rules of TRIX by teletypewriter interaction.

Text-File Functions

The text is contained in a single file or in a family of files. The text is converted to an internal format and kept in a sequence of working files. Upon termination, these working files are either destroyed or converted back to the original format and replace the original text file(s). In the latter case, the machine designator, the date, the time, the user number, and the checksum are added following the end of file. This information may be retrieved with one of the #ED options. See #ED14 and #ED20 on page 38.

#IO(A,B,C,D) I/O Buffer: Make the input-output buffer A words long. B is the minimum buffer size, C is the maximum, and D is the factor described below in #FU. The initial values of A,B,C, and D are 20774, 256, 20000, and 3. The current values are returned, space delimited, if the argument string is null.

#FU(F,L,X,R) File Open: Update the current text file, if any, and then open file F. Each line of text in file F is L characters long. If argument X is null, then file F will be converted to the above mentioned format. If arguments L and X are null, then file F will be converted and L will default to 80 or 120, whichever is correct. If file F does not exist, and argument R is not null, then a file F will be created as per #FC below. If all arguments are null, then just the update takes place. If a file is opened, then [is set to one and] is set to the number of the last line in the file.

If file F is to be converted from packed ASCII, then the input-output buffer size is modified and is equivalent to the product of factor D, described in #IO above, and the size of file F, subject to the max and min restrictions. Notice that if A=B=C in #IO above, then the buffer size is not changed by this function. %FU will evaluate to the name of the last working file. The working files have names of the form: %A001, %A002, etc. In case of conflict, names of the form: %B001, %B002, etc., will be used.

File F is understood to be the first in a family of files if argument L is understood to be negative.

If the third argument is null, and editing takes place, then the updated text file will be converted to the original format and will replace file F upon termination.

#FC(F,L,X,R) File Create: Update the current text file, if any, and then create and open text file F. Text entered into this file by the user will have a line length of L characters. The default for argument L is 80. If the third argument is null, file F will be converted to packed ASCII upon termination. Otherwise, it will be left as unpacked ASCII. F will be read-only if the last argument is R. [is set to one and] is set to zero.

#FM(F,M) File Mode: At repack time change the mode of file F to M, where M=1 for packed ASCII, M=4 for Monitor Squoze, and M=6 for Monitor M. No update is made.

#RP(F,L,P,X) Repack File: Update the current text file, if any, and then repack the working files associated with file F to a file with the original format. Name this new file P. The working files are now associated with file P; file F is closed. Line length in file P will be L characters. Make file read-only if X is null. The defaults for P and L are F and the line length of file F. The default for F is the currently open text file. When repacking a file upon itself, the original file is preserved under the name %RESTORE.

#CF(F,X) Close File: Close file F and destroy the associated working files %A001, ..., etc. No updates are made. Destroy file F also if X is non-null.

Macro Functions

Macro functions perform operations on macros. The first argument of a macro function must evaluate to a macro name.

#NS(A,S) Name String: The string S is given the name A. Such a named string is called a macro. A pointer is associated with macro A and is set to one. The nth character of a string is said to follow pointer position n. If a macro named A already exists, it is replaced by the new macro.

#SP(A,N,X) Set Pointer: The pointer associated with macro A is set to point to character N. N is positive or negative depending on whether the count is to the right or left of the current position or starts from the right or left ends of the macro. The count begins from the right or left end of the macro if argument X is null. Otherwise, the count is relative to the current pointer position. This function has a null

value except when arguments N and X are null. In this case the value is the current pointer position. This value is also assigned to system macro %SP. For example:

#SP(A,1)	sets pointer to the first character.
#SP(A,-1)	sets pointer to the last character.
#SP(A,-3,X)	sets the pointer 3 places left of the current pointer position.

#GC(A,N,X) Get Character: The value of this function is the next N characters from macro A starting from, and including, the character at the current pointer position. The count is to the right or left depending on whether N is positive or negative. If argument X is null the pointer is moved, right or left, to the first position beyond the Nth character, otherwise, the pointer is not moved. If arguments N and X are null the value is the character length of macro A. The value of this function is also assigned to system macro %GC.

This function may be used to determine if a macro by the name A is defined. If both arguments N and X are null, the value of this function is null if A is not defined.

#SM(A,S,T,F) Search a Macro: Search macro A for the first occurrence of string S following the current pointer position in macro A. Name the substring contained between the pointer and the matched string %SM, and move the pointer to the first position beyond the matched string. If a match is not found, %SM becomes the substring contained between the pointer and the end of the macro and the pointer is not moved. If the match was successful, then the value of this function is string T. Otherwise, its value is string F. The default for T and or F is <%SM>.

#GS(A,T,F) Get Symbol: A blank-delimited substring, containing no blanks, is called a symbol. The next symbol following the current pointer in macro A is named %GS. The pointer is moved to the first character beyond the symbol. If no symbol was found, or if the pointer was already past the end of the macro, the symbol is the null string, and the pointer is moved past the end of the macro. The value of this function is string T if the resulting symbol is non-null; otherwise, the value is string F. Defaults for T and F are <%GS> and the null string.

#DM(A,B...) Delete Macro: Delete macros A, B, ... A null argument deletes all pattern-replacement pairs awaiting an update. At most 30 arguments.

Terminal Input and Idling Functions

Input functions read strings from either a disk file or the teletypewriter. This string becomes their value. A disk file containing command strings is called a dialect. A dialect may call other dialects. A dialect contains teletypewriter input images. Hence, a dialogue can be had with a dialect file in the same manner as with a user at a terminal.

#RT(P,X) Read the Teletypewriter: The value of this function is the next teletypewriter message. If there is no message waiting, the prompt string P is concatenated with an end-of-message (EOM) string and sent as a prompt message. The presence of argument X, called the escape string, results in the queueing of input messages. The procedure is as follows: The prompt P is sent to the user's terminal. The user's message is read and pushed onto the queue. This procedure repeats until a user message equivalent to the escape string is received. This last message is also pushed onto the queue. The message at the head of the queue, the first one received, is called the next teletypewriter message. It is the value of this function. Subsequent #RT functions will read messages from the queue until the queue is empty. Messages can be queued at the rate of about one per two seconds. The terminal key ESC is recognized as a logical line feed (EOM). Input messages with embedded logical line feeds are treated as a message queue. Subsequent #RT functions will read messages from this queue until it is empty. The logical line feed can be inhibited. See #TC, page 28.

#WT(S,X) Write the Teletypewriter: Push string S onto the input message queue. String S is a sequence of input messages that may be internally generated. An input message is less than 148 characters long and must end with an EOM. Argument X specifies whether the input is from the teletypewriter (null) or from a dialect (non-null). This function is useful for handling user dialogue, especially iterative dialogue.

#RD(D,L) Read a Dialect in: If the argument string is null, read the next message from the dialect file. If the argument string is not null, then dialect file D either exists as a private file or is contained in library file L. In either case the idling function is set to %RD and the first message from dialect D is read. When the last dialect file becomes empty, the idling function is set to %RT. A dialect can be envisioned as a queue

of teletypewriter input messages. Edit function nineteen, #ED19, is useful for locating errors caused by commands contained in the dialect.

Arguments for a #RD function may appear in the execute line (space delimited). The default value for the second argument is A200. The dialect file must be packed ASCII.

There are two idling functions: %RT and %RD. The idling functions can be modified by redefining system macros %RT and %RD. For example, a print function could be included so that each request for a new command is signaled by some form of prompt message. System macro %IM will invoke the current idle macro.

Terminal Output Functions

Output functions send strings or arrays to either the teletypewriter, the TMDS, or an output file. The arguments of the output function are evaluated before being sent to the output device, enabling the user to utilize the full power of the language to generate output strings.

#PS(A,B,C ..) Print String: Print the string or array A on the teletypewriter. Below that, print the string or array B, and so on. All arguments of this function, except the last one, are merged with a linefeed, carriage-return string before being printed. At most 30 arguments.

#DS(A,B,C ..) Display String: The display string function is analogous to the #PS function described above, but is intended for the TMDS. If the TMDS is not being used, then it is the same as the #PS function. At most 30 arguments.

#FS(F,X) File String: Output from all subsequent #DS functions is diverted and written to file F. The TMDS need not be connected. If argument X is non-null, output from all #PS functions is also written to file F, and in addition, will not be sent to the terminal unless X is the two-character string ON. A null argument string restores subsequent output to the original output terminals. Text written to file F is concatenated to text, if any, from previous calls. File F is a queue. See #WQ on page 25. For this reason, if file F is opened as a text file, it may no longer serve as an output file.

The two meta-characters double-quote (") and question-mark (?) evaluate to 18-bit binary strings that are respectively: the internal linefeed, carriage-return, and the end-of-message. To promote efficiency, all output is held up until a string containing an end-of-message is received or the buffer is filled; at which time, transmission

occurs. This holds for all three output devices. In addition, an end-of-message string in any teletypewriter output message also empties the TMDS buffer. Notice that the standard idling function, %RT in Appendix A, generates such a string.

Display Functions

The TMDS output automatically advances line-by-line down the scope. Scope lines may be reserved so that separate information can be displayed simultaneously but independently. When the information to be displayed exceeds the allocated space, pressing the linefeed on the teletypewriter will display the next group of lines starting at the top of the allocated space. (Information in other areas is not affected.) Typing anything else will terminate the current display and will be treated as regular input. The display in each area remains until overwritten or cleared.

#SL(N,M) Scope Limits: The scope line limits for subsequent output are set to N and M. The line numbers remain in effect until changed. If the argument string is null, the standard values (1,42) or (1,51) are used. Subsequent display will begin with column 1 of the first row of this area.

#SD(N,...) Select Display: The integer argument(s) select one of the following actions:

N=0 erase display delimited by current #SL function. (This also is the default for a null N.)

N=1 N between 1 and 999 inclusive reserves display N. N between 1000 and 1999 links display N-1000. N between 2000 and 2999 unlinks display N-2000.

N=-1 will release the display.

N=-2 reverse background.

N=-3 select low-capacity characters (64 by 42).

N=-4 select high capacity characters (85 by 51). Options -3 and -4 should be separated by an ERASE character.

N=-5 select teletypewriter; do not release TMDS.

#TC(N,V,W,X,Y,Z)

Terminal Configuration: This function permits the user to describe the configuration of a specific terminal. Arguments not specifically mentioned are null.

N=0 Evaluates to the TMDS channel number, or to 0 if the TMDS channel is not connected. If negative, it denotes the position in the TMDS queue.

N=1 Output to the terminal will wrap-around after column V. Output to the display will wrap-around after column W. Output to the disk file will wrap-around after column X. An indent of Y characters is provided after each automatic wrap-around. Defaults for V, W, X and Y are the previous values. The initial values are 72, 85, 120 and 0.

N=2 Select the triple V as the "logical line feed". The @ character may be used to escape the triple V. The initial value is the triple ESC. A logical line feed may appear in the execute line. Turn off the logical line feed if V is null.

N=4 Handling of special characters by the terminal input-output functions: #RD, #RT, #PS, and #DS. Defaults for V, W, X, Y and Z are the previous values. Should all five arguments be null, the value is a five-symbol string containing the current settings. The initial values are all OFF. Options W and X may not both be ON at the same time. In fact, turning on one will turn the other off automatically.

V=ON Accept control characters in input strings.

V=OFF Remove control characters from input strings. Notify user.

W=ON Convert upper-case characters (triples) in input strings to the ↑ format. That is, an upper-case character is represented by an ↑ followed by the lower-case correspondent. Convert the five special characters, @ ' ~ ¦ and ! , to the ampersand quartets: &ATX, &LQX, &CDX, &FUX, and &EPX. This option allows one to send the 7-bit ASCII character set. The character ↑ becomes ↑↑. However, the user must enter the & as &&Q. Sets option X to OFF.

W=OFF Pass upper-case characters in input strings as triples. Do not convert the five special characters.

X=ON Recognize the pair ↑\ as a "case reverse" switch in input strings. Each input string begins in lower case. Any case reverse extends to the next case reverse or to the end of the input string. All characters in the input string are considered lower case except those immediately preceded by an up-arrow. The case reverse pair is removed from the input string. For example, input line ↑\ ABC^↑D^2R↑\ is stored internally as ↑A↑B↑C^D^^2↑R. The up-arrow character is represented in the input string as ↑↑. The user must enter the five special characters @ ' ~ ¦ and ! as the ampersand quartets: &ATX, &LQX, &CDX, &FUX, &EPX. The & is entered as &&Q. Sets option W to OFF.

X=OFF The pair ↑ is not recognized as the "case reverse" switch. The ↑ and & characters have no special meaning.

Y=ON Upper-case characters in the ↑ format appearing in output strings destined for the terminal will print as upper case. The five ampersand quartets: &ATX, &LQX, &CDX, &FUX, and &EPX, will print as @, ', ~, ¦ , and !. The strings ↑↑ and &&Q will print as ↑ and &. This option allows the user to receive the 7-bit ASCII character set.

Y=OFF No case or special character format is recognized. Output strings destined for the terminal will print as 6-bit ASCII characters.

Z=ON Upper-case characters in the ↑ format appearing in output strings destined for the TMDS will display as upper-case. The five ampersand quartets: &ATX, &LQX, &CDX, &FUX, and &EPX, will display as @, ', ~, ¦ , and !. The strings ↑↑ and &&Q will display as ↑ and &. This option allows the user to display the 7-bit ASCII character set.

Z=OFF No case or special character format is recognized. Output strings destined for the TMDS will be displayed as 6-bit ASCII characters.

Control Functions

#EQ(A,B1,T1,B2,T2, ... Bn,Tn,F)

Equal-Else: If string A is equal to (same as) string B1, then string T1 is the value of this function. Otherwise, if string A is equal to string B2, then string T2 is the value. If string A is not equal to any of the strings Bn, then string F is taken as the value.

#GE(N,M,T,F) Greater Than or Equal: The value of this function is the string T if N is numerically equal to or greater than M. Otherwise, the value is string F. N and M must evaluate to numeric values.

#IF(A,B,T,M,F) If: If each character contained in string A is also contained in string B, then the value of this function is string T. The value is string F if no character in string A is also in string B. Otherwise, the value is string M. The null character is always contained in string B.

#SF(S,F) Success-Fail: The value of this function is string S if the last pattern match was successful; otherwise the value is string F.

#DO(A,I,J,K) Do String: Do string A, ((J-I)/K)+1 times. Replace occurrences of %DO in string A with: I the first time, I+K the second time, I+2K the third time, ..., etc. I, J and K are, or evaluate to, integers, either positive or negative, and K must be non-zero. The default for K is +1 or -1, whichever is indicated by I and J. If both J and K are null, the arguments used are: 1, I, +1, or -1. If all three are null, we have the "do forever" case. A DO loop may be terminated from within, by string A, with the issuance of the function #DX. For example, to count the number of commas in string ABC:

```
COUNT=0↑ #SP(ABC,1)↑
#DO(< #SM(ABC,<,>,<COUNT=COUNT+1↑>,<#DX>)>)↑
```

(Here, the user first sets COUNT to 0 and sets the pointer in macro ABC to the first position. Then he searches macro ABC for commas. Each time a comma is found, COUNT is incremented by 1 and the #SM command is executed again, starting at the current pointer position. When the end of macro ABC is reached without finding a comma, control goes to #DX, which when executed terminates the DO loop.)

#DO functions may be nested to any level. For example, to generate a blank-delimited string, Y, of all combinations of the first five integers taken two at a time:

```
X(N)=< #DO(<$#EQ(N,%DO,,<N%DO  >)>,5)>↑
Y= #DO(< X(%DO)>,5)↑
```

Since the replacement string on the second line is not protected by angle brackets, it is evaluated and not just taken as a literal. (In the process of generating the replacement string the value of Y is generated.) The replacement string has to be completely evaluated before you can assign the value of Y to it. In the process of evaluation, the #DO function in the replacement string evaluates X five times. Each time macro X is evaluated, its argument (N) is the current value of %DO in the outside loop. The first time X is executed, N is 1; the second time, it is 2; and so on. Each time macro X is executed, it executes $#EQ(N,%DO,,<N%DO >) five times. If N is the same as the value of %DO in the inner loop, nothing is done. However, if N is not equal to %DO, the current value of N is concatenated with %DO and a space, and that in turn is concatenated with the current value of the DO function, to eventually be returned as Y. See Fig. 3 for an edit of the execution of this example.)

```
TRIX AC / 1 1
.X(N)=< #DO(<$#EQ(N,%DO,,<N%DO >)>,5)>↑
.Y= #DO(< X(%DO)>,5)↑
.Y
.#RT($Y)↑
12 13 14 15 21 23 24 25 31 32 34 35 41 42 43 45 51 52 53 54
.#ED Y= #DO(< X(%DO)>,5)↑
#DO( X(%DO),5)
#DO
X(1)
#DO($#EQ(1,%DO,,<1%DO >),5)
#DO
#EQ(1,1,,11 )
#DO
#EQ(1,2,,12 )
#DO
#EQ(1,3,,13 )
#DO
#EQ(1,4,,14 )
#DO
#EQ(1,5,,15 )
#DO
#DO
X(2)
#DO($#EQ(2,%DO,,<2%DO >),5)
#DO
#EQ(2,1,,21 )
#DO
#EQ(2,2,,22 )
#DO
#EQ(2,3,,23 )
#DO
#EQ(2,4,,24 )
#DO
#EQ(2,5,,25 )
#DO
#DO
X(3)
#DO($#EQ(3,%DO,,<3%DO >),5)
#DO
#EQ(3,1,,31 )
#DO
#EQ(3,2,,32 )
#DO
#EQ(3,3,,33 )
#DO
#EQ(3,4,,34 )
#DO
#EQ(3,5,,35 )
#DO
#DO
X(4)
#DO($#EQ(4,%DO,,<4%DO >),5)
#DO
#EQ(4,1,,41 )
#DO
#EQ(4,2,,42 )
#DO
#EQ(4,3,,43 )
#DO
#EQ(4,4,,44 )
#DO
#EQ(4,5,,45 )
#DO
#DO
X(5)
#DO($#EQ(5,%DO,,<5%DO >),5)
#DO
#EQ(5,1,,51 )
#DO
#EQ(5,2,,52 )
#DO
#EQ(5,3,,53 )
#DO
#EQ(5,4,,54 )
#DO
#EQ(5,5,,55 )
#DO
#DO
=12 13 14 15 21 23 24 25 31 32 34 35 41 42 43 45 51 52 53 54
Y
.
```

Fig. 3. Execution and edit of the second example on page 31.

The value of the #DO function is the accumulative values of its first argument.

#DX — DO Exit: Leave the current #DO loop after this iteration.

#(A) — Null: The null function. The value of this function is A. This function is used as parentheses are used in an arithmetic expression, namely, to define the order in which operations are performed.

#SK(S,X) — Skip: The remaining command string to the right of this function is placed in macro %SK and replaced with string S. The active and neutral strings are cleared. In addition, the input message queues, both teletypewriter and dialect, are cleared if argument X is null. The default for argument S is a single up arrow.

#EX(X) — Exit: If the argument is null, then update, repack, and close all text files; destroy all working files; and terminate. If the argument is not null, then terminate only.

Pattern-Field Functions

The functions available for examining subject strings for occurrences of specified substrings (pattern matching) are described below. The functions have meaning only when appearing in the pattern field of a command. They may be used separately or in conjunction with literal strings to describe patterns to be matched or selection processes. Before any matching is attempted, the expressions in the pattern field are evaluated. The resulting pattern structure defines a pattern-matching or selection procedure, which is then performed.

#TS(S) Terminal String: A pattern match fails if the string S is encountered before the pattern is matched.

#AL(A,B,C,...) Alternate Patterns: Strings A,B,C,... are alternate patterns. In the pattern field this function will match strings A or B or C or ... . In the event that two alternate patterns match the same subject sub-string, the longer of the two is selected as the matching pattern. For example, the following pattern will match: MARK, REMARK, MARKS, REMARKS, MARKED and REMARKED.

```
←#AL(RE,< >)<MARK>#AL(S,ED,< >)↑
```

The interpreter searches the subject for a blank or the string RE, immediately followed by the string MARK, immediately followed by a blank, the string ED, or the string S.

#VS(B,N,S,X) Variable String: The variable string is constrained to be N characters long. Each character of the variable string must also be a character of string S. If N is null, the variable string may be any length; and if S is null, it may contain any character. If B is not null, then a macro identical to the variable string is defined and named B. Variable-string functions are very useful for describing text patterns, as well as selecting and naming substrings from the text. The arguments of the #TS, #AL functions may not contain variable strings.

The end(beginning) of line is frequently used as a delimiter. The difficulty is that there is no character there to match. So we introduce two special cases to the above rules: #VS(B,1,S,L) is considered a match if it falls on the last character of some line. This character is not considered as part of the pattern for subsequent replacement, and B is set null. For the first character of a line, the analog is #VS(B,1,S,F).

In the examples below, the text consists of two 10 character lines: XYZBBB-ABC and DEF+ABCMMM.

[]←#VS(M)<ABC>↑
The match is successful. The pattern is XYZBBB-ABC, and M is XYZBBB-.

[]←<A>#VS(M)#VS(N,2)<F>↑
The match is successful. The pattern is ABCDEF, and M and N have the values BC and DE.

[]←<A>#VS(M,1)#VS(N,2)<F>↑
The match fails. The pattern string, %PN, is null.

[]←<Y>#VS(M,,$%AB<->)#VS(N,1,<+>)<ABC>↑
The match is successful. M is ZBBB-ABCDEF, N is + and the pattern is YZBBB-ABCDEF+ABC.

[]←#VS(,1,<+-*/>,L)<D>↑
The match is successful. The pattern is D.

#CR(N,M) Column Restriction: This function appears only in the pattern field and restricts the subsequent search as follows: The subject must have a line structure, the pattern is M characters long, and any match must begin in column N. This function provides a very fast search and may not be used with the #VS or #AS functions. N+M must be less than (N/10)*10+21, where the divide is an integer divide.

#MC(N) Move Cursor: Move the cursor marking the starting character for the next pattern-match N places. Move it left for N negative, or right for N positive. If the argument is null, the value of this function is the cursor position relative to the beginning of the text file. The initial value of the cursor is one.

Miscellaneous Functions

#WQ(N,S,X,F) Write Queue: Push string S onto queue N with option X. A queue name is less than 11 characters long and may contain any character. There are seven options. The push is performed before the option. Queues used for temporary storage should be destroyed (use option D) when no longer needed. The length of queue N is limited by the hardware to (2**24)*10 characters. Queues are very useful data structures because they are essentially open ended. The user may assume that the queue is

contained in core. However, when core gets full, queues will be spilled out to disk to make room. Subsequent I/O dictated by this device will be handled efficiently by the interpreter.

X=D delete queue N.

X=K keep queue N as disk file N at termination time.

X=L the character length of queue N is returned as the value of this function. The value is null if queue N doesn't exist.

X=F flush the core buffer of queue N to disk file N. An end-of-file string is added to the queue. A subsequent flush will overwrite any existing end-of-file.

X=P push an "end-of-record" string onto queue N. A device to generate packed-ASCII queues.

X=R replace the working files for file F with queue N.

X=B reset the "bottom" of queue N to one. See #RQ.

#RQ(N,C,X,Y) Read Queue: The name "queue" is a misnomer since this storage device may be used as either a queue, a stack, or just as a linear store. Read C characters from queue N with option X. There are three options. Option T results in the queue being shortened by C characters and should not be followed by options B or R. Options B and R are simple reads and do not change the size of the queue. A "bottom" pointer is incremented for each "read from bottom".

X=T read and remove from top of queue N.

X=B read from bottom of queue N.

X=R read starting with character number Y from queue N.

#SC(S,T) System Call: Macro S has a bit pattern that is the same as that of some system call. The beta address is set by TRIX. The beta words are a substring of macro S. beta is characters 11 thru 20, beta+1 is characters 21 thru 30 etc. Information returned by the system will replace substrings of macro S. The function evaluates to system macro %SC if the call fails, otherwise, the value is null. The user may set system macro %SC.

T is a four-digit octal integer specifying whether the call is to be processed by FLOE or by EEA. The default for T is 1004. See LTSS-10.[4]

#AD(X,Y) ASCII-Decimal: Either X or Y is null. If Y is null, the value is the floating-point binary representation of the ASCII decimal X. If X is null, the value is the fixed-point binary representation of the ASCII

integer Y. In either case, the value string is 10 characters long and is not evaluated further.

#AO(X,Y) ASCII-Octal: Either X or Y is null. If Y is null, the value is the 10-character representation of the 20-character ASCII octal string X. If X is null, the value is a 20-character ASCII octal representation of the 10-character string Y. In either case, the value is not evaluated further. For example:

```
#AO(<41424344454647505152>)  becomes  ABCDEFGHIJ
#AO(,<ABCDEFGHIJ>)  becomes  41424344454647505152
```

#BY(C,M,T,V,X) Bye: Initialize controllee C with time T and value V. Send it message M. If T and V are null, use the current values. All message bypasses are set. If X is present, do not print the controllee's ALL DONE.

#SS(S,D,K) Sort Strings: The substrings of string S that are delimited by string D, are sorted according to the 64-character key K, and delimited by string D, and the sorted strings replace S in the command. The default value for D is one or more blanks and for K is the ASCII set in the usual order.

#TM(S) Trim: The value of this function is string S with the trailing spaces removed.

#ED(N,W,X) Edit: This function incorporates a number of useful devices for debugging and checking for user errors. Unless specified, arguments W and X are null.

N=0 or 10 List on the teletypewriter (N=0) or TMDS (N=10) each macro call or function following this command on the input line, along with its arguments, just before evaluation on the teletypewriter or TMDS. The trace mode is indispensable for debugging dialects. The trace is turned off by the idle macro. To continue the trace past the idle macro, redefine the idle macro to include the trace: %RD=<#ED10 #RD>↑, etc.

N=1 Evaluates to a "linefeed, carriage-return" delimited string containing all macro names.

N=3 Returns as value the latest undefined macro. See %YA on page 19.

N=5 Display the "list space" on the teletypewriter (or TMDS, if connected).

N=6 Evaluates to the CPU, SYSTEM, and I/O time used, the number of disk reads and writes, the average length per read and write, the approximate number of suspends to disk, the number of packets sent to the PDP-10(TMDS), the number of logical linefeeds, the number of updates (long and short), and the field length. Counters for all of the above are decimal integers, are blank-delimited and are reset to zero. Time is in microseconds.

N=8 Perform decimal arithmetic.

N=9 Perform integer arithmetic.

N=10 See N=0 Above.

N=11 Turn off active-neutral string error print.

N=12 Turn on active-neutral string error print.

N=13 Evaluates to the current "machine, date, time" string.

N=14 Evaluates to "machine, date, time, user-number, cksum" string for currently opened text file.

N=17 Update mode. This edit function requires three additional arguments. The first is T or L, for text mode or line mode. The second is the actual line length, and the third is a single character, which is the line-overflow flag for line mode. This last argument is not needed for text mode.

N=18 Update query. Returns the number of overlapping patterns in any outstanding text-file update. However, no update is performed.

N=19 When you are reading from a dialect file, #ED19 will print the current card number and contents in event of an error. Otherwise, it does nothing.

N=20 Evaluates to a string containing information on the files that are currently open. The data consists of the file name, working-file name, line width, the read-write status, mode (ASCII, MONITOR, etc.), and security level. The data is blank-delimited and ends with a comma.

N=21 Evaluates non-null if there are outstanding updates between text lines W and X inclusive, where W and X are the second and third arguments.

N=22 Update switch. If the second argument is OFF, all updates are inhibited. A second argument ON allows updates as per context. This is the default value.

APPENDIX C

EXAMPLES

1. List the Line Numbers of All Lines in a File That Contain a Given Pattern

Open a file named ABC. Make a list consisting of the line numbers of all the lines in file ABC that contain the pattern XYZ. Name this list LIST. Delimit the entries in this list with blanks. The following three commands suffice.

```
P1=<←<XYZ> #SF(<#NS(LIST,$LIST< >[) P1>)>↑
#FU(ABC)#NS(LIST)↑
[] P1↑
```

First define a macro, P1, which will expand to the pattern field required for the search. Notice that it begins with a ← and that the #SF function is used to trigger recursion, ensuring that all such patterns, not just the first one, are found. Next, open the file and initialize the list. The list is referred to in P1; hence a string by that name must exist (a null one will suffice). In the last command, [] sets the subject to be the entire file. Macro P1, being preceded by a space, is expanded and placed on the command string. It is recognized as a pattern field describing a pattern XYZ. The scanner is left pointing to the #SF function and the subject is searched for the pattern. When the search is completed, the scanner takes up where it left off. The #SF function is evaluated. Since it was preceded by a blank, its value is placed on the command string. The value is null if the match failed, and the command scanner continues, finding the terminal up-arrow. If the match succeeded, then a space < > and the line number of the pattern just matched, [, is concatenated to macro LIST, and macro P1 is expanded and placed on the command string. The command scanner sees the back-arrow, signaling a pattern field. Since the subject wasn't respecified, the search continues, starting from the first character following the previous match. In this manner the entire text is scanned. If the pattern appears more than once in a given line, the line number is duplicated in the list. If this is to be avoided, change the recursion to respecify the subject to begin with the line following the match.

```
P1=<←<XYZ> #SF(<#NS(LIST,$LIST< >[)↑[+1;] P1>)>↑
```

The list of line numbers is in monotonic increasing order. To put them in monotonic decreasing order, change the order of concatenation.

2. Remove All Blank Characters from a Macro

Remove all blank characters from macro ABC. The following command suffices.

```
X = #DO(< #GS(ABC,<$%GS>,<#DX>)>)↑
```

Here we take advantage of the "do forever" feature of the #DO function. Symbols are removed from macro ABC and concatenated to form the body of macro X. The absence of symbols to be removed terminates the DO loop. The following example illustrates a different approach to the same problem. Notice that where this is a sequential solution, the next example is an associative solution.

3. Count the Number of Blank Characters in a Macro

Count the number of blank characters in macro ABC. The following three commands suffice:

```
X(< >)=$ABC↑
Y=$X↑
COUNT=#GC(X)-#GC(Y)↑
```

The first command defines a macro X to have a body equal to that of macro ABC with a single blank as a parameter. The second command defines a macro Y to have a body equal to that of macro X with the null string substituted for all occurrences of the parameter. Notice that macro Y is macro X with all the blanks removed. Therefore, the number of blank characters in macro ABC is the difference in the length of macros X and Y. This difference is assigned the name COUNT by the third command.

4. An Experiment in String Arithmetic

The potential and versatility of string arithmetic is exceeded only by the paucity of the rules governing its use. There are too many possible cases to reconcile into a few concise rules, so, experiment to find out what works, and use it.

1. Internally, strings are concatenated before, and arrays are merged after, the operations implied by the colon or quote are performed.

2. The null function (page 33) may be used as parentheses to describe the order of operations, as in algebra.

3. For notational convenience, colons in the form n:m may be strung together with literal strings and tabs, the whole thing operating on the same subject somewhere to the left. However, colons in the form n: :n or : take as the subject the first array or string to the left.

4. The user must discover how to achieve the more elaborate results by experimentation. See the example on the right.

```
TRIX AC / 1 1
.0(AA,10) T
2 LINES. (10A)
1 1234567890
2 ABCDEFGHIJ
.#PS(1; 1:3 :)
123.
.#PS(1; 1:3 1:)
1
2
3
.#PS(1; 1:2 <ABC> 8' 7:8)
12ABC   78
.#PS(1; 1: 2; 1:)
1A
2B
3C
4D
5E
6F
7G
8H
9I
0J
.#PS(1; 1: 2; 1: :)
1ABCDEFGHIJ
2ABCDEFGHIJ
3ABCDEFGHIJ
4ABCDEFGHIJ
5ABCDEFGHIJ
6ABCDEFGHIJ
7ABCDEFGHIJ
8ABCDEFGHIJ
9ABCDEFGHIJ
0ABCDEFGHIJ
.#PS(#(1; 1: 2; 1:) :)
1A2B3C4D5E6F7G8H9I0J.
.#PS(1; : <ABC> 10:)
1234567890
ABC
.END

 ALL DONE
```

Explanations for Example 4

#PS(1; 1:3 :)	Takes columns 1 through 3 of line 1 and formats them 3 character string.

#PS(1; 1:3 :)	Does the same as above, but formats columns 1 through 3 of line 1 into an array of 1 column, 1 character at a time.
#PS(1; ':2 AB€ 8' 7:8)	Takes columns 1 through 2 from line 1, concatenates it with ABC, blank fills the result out to 8 columns, and concatenates it with columns 7 through 8 of line 1.
#PS(1; 1: 2; 1:)	In effect, this takes columns 1 through 10 of line 1, formats them into an array of 1 column (the items being taken 1 at a time), then takes columns 1 through 10 of line 2, and formats them into an array of 1 column (the items being taken one at a time), and finally concatenates the first array with the second array (see note 1 above).
#PS(1; 1: 2; 1: :)	In effect, this takes columns 1 through 10 of line 1, formats them into an array of 1 column (the items being taken 1 at a time), then takes columns 1 through 10 of line 2 and formats them into an array of 1 columns taken 1 at a time, and finally merges the first array with the second array into a string.
#PS(#(1; 1: 2; 1:) :)	This is a shuffle. It takes one element from each set and interleaves them. It is perhaps the most valuable of the examples on this page. (See note 2.)

REFERENCES

1. D. J. Farber, R. E. Griswold, and I. P. Polonsky, J. Assoc. Computing Machinery 11, 21 (1964).
2. C. N. Mooers, Commun. ACM 9, 215 (1966).
3. A. Cecil, H. Moll, and J. Rinde, TRIX AC: A Set of General-Purpose Text Editing Commands, Lawrence Livermore Laboratory, Rept. UCID-30040 (1974).
4. P. Du Bois et al., CDC 7600 System Calls and I/O Requests, Lawrence Livermore Laboratory, Rept. LTSS-10 (1974).

INDEX

#TM(S)	Trim	37
#TS(S)	Terminal String	34
#VS(B,N,S,X)	Variable String	34
#WQ(N,S,X,F)	Write Queue	35
#WT(S,X)	Write Teletypewriter	26

CRS/lt