DISCUSSION

Salton opened the discussion with the comment that a system such as this is inherently not extendable. The system will operate nicely with various kinds of "fish," but will run into trouble when "whales" appear, since the system will not know how to deal with aquatic mammals. He compared the system to the "Baseball" system, in which one cannot go beyond a limited range of questions, e.g., one has trouble if a new team appears. Young objected to this view, saying he believed the two systems were quite different, and that the present system was in principle infinitely extendable and very general. He said he could conceive of a system one level above this one which could deal with generalizations and which would make handling propositions easier than with the present special programming.

Burger said that work with higher level relationships was planned, but that the immediate extensions would be more trivial, in the way of inserting a great deal of knowledge about the world, of the kind any child has (e.g., "Walls are vertical.").

Gorn observed that the present system is based upon two forms of the verb "is" and the verb "has," and that more relationships were needed. Burger said the system was not in principle so limited. Gorn then asked how they would deal with the statement " 'Word' is a word." Burger had no immediate answer, though he thought they would eventually be able to deal with such cases.

Responding to a question from Cheydleur, Burger said they had about 50 different functions.

Mooers asked if there were any inherent features of Lisp which limited their work. Burger said that a fundamental limitation was the limitation to use of core storage alone in the SDC version of Lisp. They could not use disks. The second limitation was the inability to break out individual characters from the "atoms" of Lisp. This prevents an easy and direct way of treating the similarity between "dog" and "dogs." At present this relationship has to be put in as a separate piece of information.

Mitchell then mentioned that for a very much larger data base a limitation will be the time required to pass the dictionaries through the machine. He said one of the big improvements in speed due to syntax-directed compilers was that they had less need to refer to a very large dictionary. Burger admitted that this was a very important problem, and one which is being considered. What can be done outside of Lisp is being studied, since a problem in using an auxiliary memory with a list-structure system like Lisp is that interrelationships between lists are broken up if just one section is brought from the auxiliary memory. So far, a good solution to the problem has not been found.

TRAC, A Procedure-Describing Language for the Reactive Typewriter

Calvin N. Mooers
Rockford Research Institute Inc., Cambridge, Massachussetts

A description of the TRAC (Text Reckoning And Compiling) language and processing algorithm is given. The TRAC language was developed as the basis of a software package for the reactive typewriter. In the TRAC language, one can write procedures for accepting, naming and storing any character string from the typewriter; for modifying any string in any way; for treating any string at any time as an executable procedure, or as a name, or as text; and for printing out any string. The TRAC language is based upon an extension and generalization to character strings of the programming concept of the "macro." Through the ability of TRAC to accept and store definitions of procedures, the capabilities of the language can be indefinitely extended. TRAC can handle iterative and recursive procedures, and can deal with character strings, integers and Boolean vector variables.

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965; a supplementary paper, not on the announced program.

The present work was supported in part by the following grants and contracts: Advanced Research Projects Agency contract SD-295, Information Sciences Branch of the Air Force Office of Scientific Research contracts AF-AFOSR 376, 377 and 461-64, and Division of General Medicine, National Institutes of Health, U.S. Public Health Service grant GM 10416.

Introduction

The Trac (Text Reckoning And Compiling) language system is a user language for control of the computer and storage parts of a reactive typewriter system. A reactive typewriter is understood to be one of a number of teletypewriters simultaneously connected online by wire to a memory and computer complex which permits real-time, multiple access (time-shared) operation. In the philosophy of the reactive typewriter, the man at the typewriter keyboard is the focal point of the system. The connected storage and computer devices are considered to be peripheral service units to the reactive typewriter.

The design goals for the Trac language and its translating system included: (1) high capability in dealing with back-and-forth communication between a man at a keyboard and his work on the machine, so as to permit him to make insertions and interventions during the running of his work; (2) maximum versatility in the definition and performance of any well-defined procedure on text; (3) ability to define, store and subsequently use such procedures to extend the capabilities of the language; and finally (4) maximum clarity in the language itself so that it could be easily taught to others. A discussion of these design goals, and of the design decisions which went into the language, may be found in a companion paper [1]. The TRAC language has now been programmed for several computers.[1] It has shown a high degree of stability during the past year of experimental use.

The present TRAC language is machine-independent and is closed with respect to operations performed upon sets of characters from a typewriter keyboard. The present language should be precisely designated as "TRAC 64."[2] Later versions of TRAC are expected to have the ability to deal with strings of machine-coded words and with subroutines, and will thus be self-implementing.

The TRAC language was developed after a study of a number of procedure-describing languages, but only after it was concluded that each of these had features which were believed to be unsuitable or unduly constraining for the purposes contemplated by TRAC. In particular, the languages IPL-V, LISP and COMIT were carefully examined. In brief, IPL-V appeared to be too closely oriented to computer programming. LISP had severe restrictions due to its "atomic" symbols, certain conceptual confusions and too great an orientation to mathematical logic. COMIT, while suitable in many respects, had the rigidity associated with a compiler. When work was begun on TRAC (1960) none appeared to have the capabilities desired, though they were a definite source of inspiration.

The prime stimulus to the present TRAC language came from two important unpublished papers by Eastwood and McIlroy [3] and McIlroy [4]. The first paper described a macro assembly system having run-time definition-making and decision-making capabilities. The second paper showed how this system could perform very general manipulations on symbol strings. TRAC is a refinement and extension of the macro approach of these papers. Therefore it can be said that the present TRAC system consists of a machine-independent language together with a generalized macro text processor which runs interpretively to provide versatile interaction capabilities at run time. A recent, independently developed system by Strachey [5] has a number of remarkable similarities to TRAC.

TRAC Syntax

A TRAC string may contain a substring enclosed by a matching pair of parentheses, such as (···) where the dots indicate a string. The matching parentheses indicate the scope of some particular action. There are three cases, epitomized by #(···), ##(···) and (···).[3] The first two formats indicate the presence of a TRAC "primitive function." The format #(···) denotes an "active function," while the format ##(···) denotes a "neutral function." This distinction is clarified below. The string interior to either kind of function is generally divided into substrings by commas as in #(, ,) where these substrings constitute the arguments of the function. Parentheses in the format (···) have roughly the same role as paired quotation marks, and, in particular, whatever string is inside the paired parentheses is protected from functional evaluation.

TRAC strings are dealt with by the TRAC processor according to a scanning algorithm which works from left to right and performs the evaluation of nested expressions from inside outward. In the expression

```
#( ,#( , , #( ) , ##( ) ))
4   3      1      2
```

the functions are evaluated in the order indicated. As each function is evaluated, it is replaced in the TRAC string by the string (possibly null) which is its value. The evaluation of an active function is followed directly by the evaluation of any function in its value string not protected by matched parentheses. The value string of a neutral function is not further evaluated.

Examples of Functions

An example of the "define string" primitive function is the expression #(*ds,AA,CAT*). This causes recording of the string *CAT* in the memory and places the name *AA* of the string in a table of contents. The string can be called out of memory by the "call" function #(*cl,AA*). The result of the call function is to place the string value of the call, namely *CAT*, in the former location of #(*cl,AA*), with an expansion or a closing up of the surrounding strings. The call function is in the class of functions having a "value string." The define string function is an example of a function having a "null value"; i.e., no string is left behind in its place after its evaluation.

Evaluation of the "read string" function #(*rs*) causes the processor to accept input from the typewriter. Its value is the string as received from the typewriter up to a terminating "meta character" which is usually taken to be the apostrophe. The meta character can be changed. The "print string" function #(*ps,X*) causes printing out of the argument string, here represented by the symbol *X*. It has null value. The nested expression #(*ps*, #(*cl,AA*)) will cause *CAT* to be printed out.

In the beginning, and at the completion of every processing cycle, the TRAC "idling procedure" #(*ps*, #(*rs*)) is automatically loaded into the TRAC processor. It is therefore seen that all strings and programs are effectively loaded into the interior of the idling procedure, and furthermore, all TRAC computations are made on functions nested within the argument string of some other function.

TRAC Algorithm

The TRAC algorithm governs the precise manner in which TRAC expressions are scanned and evaluated by the TRAC processor. At the beginning, the unevaluated strings are in the "active string" and the "scanning pointer" points to the leftmost character in this string. As characters have been treated by the scanning algorithm, they may be added to the right-hand end of a "neutral string," which is so called because its characters have been fully treated by the algorithm and are thus neutral, like alphabetic characters. The algorithm follows.

[1] The computers are the Digital Equipment PDP-1, PDP-5, the General Electric Datanet-30 and the Scientific Data Systems SDS 930.

[2] The present paper is a revision and extension of [2].

[3] With the model 33 Teletype, the "up arrow" character is preferred instead of the sharp sign.

1. The character under the scanning pointer is examined. If there is no character left (active string empty), go to rule 14.

2. If the character just examined (by rule 1) is a begin parenthesis, the character is deleted and the pointer is moved ahead to the character following the first matching end parenthesis. The end parenthesis is deleted and all nondeleted characters passed over (including nested parentheses) are put into the neutral string without change. Go to rule 1.

3. If the character just examined is either a carriage return, a line feed or a tabulate, the character is deleted. Go to rule 15.

4. If the character just examined is a comma, it is deleted. The location following the right-hand character at the end of the neutral string, called the "current location," is marked by a pointer to indicate the end of an argument substring and the beginning of a new argument substring. Go to rule 15.

5. If the character is a sharp sign, the next character is inspected. If this is a begin parenthesis, the beginning of an active function is indicated. The sharp sign and begin parenthesis are deleted and the current location in the neutral string is marked to indicate the beginning of an active function and the beginning of an argument substring. The scanning pointer is moved to the character following the deleted parenthesis. Go to rule 1.

6. If the character is a sharp sign and the next character is also a sharp sign, the second-following character is inspected. If this is a begin parenthesis, the beginning of a neutral function is indicated. Two sharp signs and the begin parenthesis are deleted and the current location in the neutral string is marked to indicate the beginning of a neutral function and the beginning of an argument substring. The scanning pointer is moved to the character following the deleted parenthesis. Go to rule 1.

7. If the character is a sharp sign, but neither rule 5 or 6 applies, the character is added to the neutral string. Go to rule 15.

8. If the character is an end parenthesis, the character is deleted. The current location in the neutral string is marked by a pointer to indicate the end of an argument substring and the end of a function. The pointer to the beginning of the current function is now retrieved. The complete set of argument substrings for the function have now been defined. The action indicated for the function is performed. Go to rule 10.

9. If the character meets the test of none of the rules 2 through 8, transfer the character to the right-hand end of the neutral string and go to rule 15.

10. If the function has null value, go to rule 13.

11. If the function was an active function, the value string is inserted to the left of (preceding) the first unscanned character in the active string. The scanning pointer is reset so as to point to the location preceding the first character of the new value string. Go to rule 13.

12. If the function was a neutral function, the value string is inserted in the neutral string with its first character being put in the location pointed to by the current begin-of-function pointer. Delete the argument and function pointers back to the begin-of-function pointer. The scanning pointer is not reset. Go to rule 15.

13. Delete the argument and function pointers back to the begin-of-function pointer for the function just evaluated, resetting the current location to this point. Go to rule 15.

14. Delete the neutral string, initialize its pointers, reload a new copy of the idling procedure into the active string, reset the scanning pointer to the beginning of the idling procedure, and go to rule 1.

15. Move the scanning pointer ahead to the next character. Go to rule 1.

The TRAC processor will accept any string of symbols. Nonexistent functions are given a null value. Omitted arguments are given a null value, while extra arguments are ignored. Omitted right parentheses will cause the processor to terminate its action and reinitialize itself at an unexpected point, while extra right parentheses are ignored and deleted at the end of a procedure. When the processor becomes too full, perhaps due to an infinite iteration or recursion, a diagnostic is typed out to indicate that fact and the processor is re-initialized by going to rule 14. The break key stops any action and causes re-initialization.

The TRAC Functions

Input-Output. All functions are shown in their active representation, which is the form most often used. As shown, the argument strings are presumed not to contain functions or other active matter.

#*(rs)* "read string" (one argument). (Note that the mnemonic for the function name is counted as the first argument.) The value is the string as read from the teletypewriter keyboard up to the point of occurrence of the meta character, which is deleted.

#*(rc)* "read character" (one argument). The value is the next character, which may be any character (including the meta character) received from the teletypewriter.

#*(cm,X)* "change meta" (two arguments). This null-valued function changes the meta character to the first character of the string symbolized by *X*. Upon starting, the TRAC processor is loaded with a standard meta character, usually the apostrophe.

#*(ps,X)* "print string" (two arguments). This null-valued function prints out on the teletypewriter the string represented by *X*.

Define and Call Functions

#*(ds,N,X)* "define string" (three arguments). This is a null-valued function. The string symbolized by *X* is placed in storage and is given the name symbolized by *N*. The name is placed in a name list or table of contents to the "forms" in storage. A "form" is a named string in storage. If a form is already in storage with name *N*, this form is erased. The name *N* may be a null string.

#*(ss,N,X1,X2,···)* "segment string" (three or more arguments). This is a null-valued function. The form named N is taken from storage and is scanned from left to right with respect to string $X1$. If a substring is found matching $X1$, the location of the match is marked. The matching substring is excluded from further action, thus creating a "segment gap." The rest of the form is scanned with respect to $X1$ to create any additional segment gaps. These segment gaps are all given the ordinal value one. The parts of the form not taken by segment gaps are now scanned with respect to string $X2$, with the creation of segment gaps of ordinal value two. This action is repeated with all of the remaining argument strings. At the end, the marked form, along with its pointers and ordinal identifiers for the segment gaps, is put back into storage with the name N. The untouched portions of the string in the form are called "segments." It is seen that the segment string function creates a "macro" in which the arguments $X1$, $X2$, etc., indicate the dummy variables. The segment string function subsequently can be applied with other arguments to the same form, with the result of new segment gaps being created with ordinal value one, two, etc., and being inserted among those already there. A null string for one of the arguments X causes no action for this argument.

#$(cl,N,X1,X2,\cdots)$ "call" (two or more arguments). The value is generated by bringing the form named N from storage and filling the segment gaps of ordinal value one with string $X1$, the gaps of ordinal value two with string $X2$, and so on for all the segment gaps in the form.

The following specialized calls read out a part of a form. They treat the segment gaps as if the gaps were filled with the null string. These calls (*cs*, *cc*, *cn*, and *in*) preserve the neutral-active function distinction only for the strings coming from the form named N. Since the alternative value of these functions, symbolized by Z, may be a call to a procedure, the alternative value is always treated as if the function were active.

All the call functions (*cl*, *cs*, *cc*, *cn*, and *in*) read the text of a form beginning at the location indicated by a "form pointer" which is part of the apparatus of the form. Initially the form pointer points at the first character of the form. The call function does not change the form pointer.

#(cs,N,Z) "call segment" (three arguments). The value of this function is the string from the current location of the form pointer to the next segment gap of the form named N. If the form is empty, the value is Z. The form pointer is moved to the first character following the segment gap.

#(cc,N,Z) "call character" (three arguments). The value is the character under the form pointer. If the form is empty, the value is Z. The form pointer is moved one character ahead (segment gaps are skipped).

#(cn,N,D,Z) "call n characters" (four arguments). This function reads from the form named N from the point indicated by the form pointer and continuing for a number of characters specified by the decimal integer number at the tail end of the string symbolized by D. Segment gaps are skipped. If the decimal number is positive, this function reads the string to the right of the pointer; if negative, to the left. The strings so read are preserved in their character sequence. If no characters are available to be read, the value is Z. The form pointer is moved (right or left) to the next unread character.

#(in,N,X,Z) "initial" (four arguments). Starting from the form pointer, the form named N is searched for the first location where the string X produces a match. The value is the string from the pointer up to the character just before the matching string. If a match is not found, the value is Z. The form pointer is moved to the character following the matching substring, or is not moved if there is no match.

#(cr,N) "call restore" (two arguments). This null-valued function restores the form pointer of the form named N to the initial character.

#$(dd,N1,N2,\cdots)$ "delete definition" (two or more arguments). This null-valued function deletes the forms named $N1$, $N2$, etc., from memory and removes their names from the list of names.

#(da) "delete all" (one argument). This null-valued function deletes all the forms in memory, and removes their names.

Arithmetic Functions. TRAC does integer arithmetic, taking decimal arguments. The decimal numeric digits are looked for at the tail ends of the argument strings. The prefix string of the first argument string is preserved and is appended to the answer, while the prefix string of the second argument is ignored. Negative quantities are indicated by the minus sign "$-$", and initial zeros are ignored. Whenever the integer values become so large as to overflow the capacity of the arithmetic processor, the overflow value Z of the function is taken. The overflow value is always treated as if it were produced by an active function. The arithmetic functions are: #$(ad,D1,D2,Z)$ "add", #$(su,D1,D2,Z)$ "subtract", #$(ml,D1,D2,Z)$ "multiply" and #$(dv,D1,D2,Z)$ "divide". They all take four arguments. In these functions, $D2$ is subtracted from $D1$, and $D1$ is divided by $D2$, with the answer being the largest integer contained in the dividend.

Boolean Functions. Boolean TRAC functions operate on strings of bits (of value 0 or 1), i.e., on Boolean vectors. The bit strings are represented by octal digits, with each digit representing three bits. Thus the bit strings have lengths in multiples of three. The octal digits are looked for at the tail end of the 01 and 02 strings, and any non-octal prefix matter is deleted The functions are: #$(bu,$01,02) "Boolean union," #$(bi,$01,02) "Boolean intersection," #$(bc,$01) "Boolean complement," #$(bs,$ $D1$,01) "Boolean shift" and #$(br,D1,$01) "Boolean rotate." The bit strings are right justified. In the Boolean union the shorter string is filled out with leading zeros, while in the Boolean intersection, the longer string is truncated at the left. In the complement, shift and rotate, the length of the bit string remains the same. Shift is to the left by the number of places specified by the decimal $D1$ (with leading nondecimal matter being deleted) when

*D*1 is positive, and to the right when *D*1 is negative. The new positions created by the shift are filled with zeros. Rotate is also to the left or right, with positive or negative values of *D*1. The digits displaced from one end of the vector are added to the place created at the other end.

Decision Functions

#(*eq*,*X*1,*X*2,*X*3,*X*4) "equals" (five arguments). This is a test for string equality. If *X*1 is equal to *X*2, the value is *X*3; otherwise it is *X*4.

#(*gr*,*D*1,*D*2,*X*1,*X*2) "greater than" (five arguments). This is a test of numerical magnitude. If the integer decimal number at the tail of string *D*1 is algebraically greater than the number at the tail of *D*2 the value is *X*1; otherwise it is *X*2.

External Storage Management Functions

#(*sb*,*N*,*N*1,*N*2,···) "store block" (three or more arguments). This null-valued function assembles the group of forms named *N*1, *N*2, etc., and stores them as a block in an external storage area. The form names, segment gaps, etc., are all preserved. When the forms have been put into the external storage, they are erased from form storage. A new form is created with name N and with a string which is the address of the block in external storage.

#(*fb*,*N*) "fetch block" (two arguments). This null-valued function is the converse of the store block function. The name *N* is the name of the block to be fetched. The function restores to form storage all the forms in the block, complete with names, segment gaps, pointers, etc. It does not erase the block in external storage, nor the form named *N*.

#(*eb*,*N*) "erase block" (two arguments). This null-valued function erases the form named *N* and also the group of forms in the block in external storage.

These functions permit forms to be moved to and from the main memory and also protect the stored forms from accidental erasure. They also permit one to build a "storage tree." By this technique, a group of forms can be stored under a group name, a set of group names can be stored under a section name, and so on.

Diagnostic Functions

#(*ln*,*X*) "list names" (two arguments). The value of this function is the list of names in the name list, i.e., the names of all the forms in form storage. Each name in the value string is preceded by string *X*. If *X* is the character pair "carriage return, line feed" protected by double parentheses, the names will be listed in a column.

#(*pf*,*N*) "print form" (two arguments). This causes the typing out of the form named *N* with a complete indication of the location and ordinal values of the segment gaps.

#(*tn*) "trace on" (one argument). This null-valued function initiates the trace mode in which, as the computation progresses, the neutral strings for each function are typed out. Typing the backspace key causes evaluation of the function, and presentation of the neutral strings of the next. Typing anything other than backspace causes initialization. Carriage return may be used instead of backspace.

#(*tf*) "trace off" (one argument). This is a null-valued function which terminates the trace mode without initialization. Both trace on and trace off functions may be placed anywhere in a procedure.

Examples of TRAC Procedures

1. The distinction between active and neutral functions is usually puzzling. In essence, the value from an active function is rescanned, while in the neutral function it is not. The following example shows the action of the protective parenthesis, the neutral and the active forms of the function. Consider that both #(*ds*,*AA*,*CAT*)′ and the simple program #(*ds*,*BB*,(#(*cl*,*AA*)))′ have been presented to the processor. Then,

```
#(ps,(#(cl,BB)))',      #(ps,##(cl,BB))',   #(ps,#(cl,BB))'
```

prints out, respectively:

```
#(cl,BB),               #(cl,AA),           CAT
```

2. When the processor is quite full, it is often desirable to delete all forms but one of a particular name. The procedure #(*ds*,*N*,##(*cl*,*N*)#(*da*)) will accomplish this. Here ##(*cl*,*N*) reads the form *N* into the processor, and it is held in the neutral string while all the forms in memory are erased. The form is then redefined with its original name. In this example, segment gaps are lost.

3. This and the following example illustrate the extension of TRAC capabilities through defining and storing of suitable procedures. The calculation of the factorial of a number can be done by simple recursion:

```
#(ds,Factorial,(#(eq,1,X,1,
(#(ml,X,#(cl,Factorial,#(ad,X,-1))))
)))#(ss,Factorial,X)'
```

Then the call #(cl,Factorial,5)′ produces the result 120.

4. Many users will prefer to have TRAC supply its own sharp signs and parentheses when calling a procedure. The following will do this:

```
#(ds,English,(#(ps,
#(cl,#(rs))(
))#(cl,English)))'
```

To start this action, we use #(*cl*,English)′, and then if one types in Factorial,5′ the response is 120 followed by carriage return, line feed. The action is terminated by #(*dd*,English)′.

Acknowledgments. Special thanks are due to my collaborator L. Peter Deutsch for his assistance in the development and implementation of TRAC.

REFERENCES

1. MOOERS, C. N. TRAC, a text handling language. Proc. ACM 20th Nat. Conf. Cleveland, Aug. 1965, pp. 229–246.
2. TRAC—a procedure defining and executing system. Mem. V-157, Rockford Research, Cambridge, June 1964.
3. EASTWOOD, D. E., AND MCILROY, M. D. Macro compiler modification of SAP. Mem., Comput. Lab., Bell Telephone Labs., Murray Hill, N.J., Sept. 3, 1959. (Unpublished)
4. MCILROY, M. D. Using SAP macro instructions to manipulate symbolic expressions. Mem., Comput. Lab., Bell Telephone Labs., Murray Hill, N.J., 1960. (Unpublished)
5. STRACHEY, C. A general purpose macrogenerator. *Comput. J.* *8*, 3 (1966).