379
N81
NO. 4880

DESIGN AND IMPLEMENTATION OF A TRAC PROCESSOR FOR FAIRCHILD F24 COMPUTER

THESIS

Presented to the Graduate Council of the
North Texas State University in Partial
Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Ping Ray Chi, B. A.
Denton, Texas

August, 1974

Chi, Ping R., Design and Implementation of a TRAC Processor for Fairchild F24 Computer, Master of Science (Computer Sciences), August, 1974, 77 pp., 6 illustrations, 2 appendices, bibliography, 7 titles.

TRAC is a text-processing language for use with a reactive typewriter. The thesis describes the design and implementation of a TRAC processor for the Fairchild F24 computer.

Chapter I introduces some text processing concepts, the TRAC operations, and the implementation procedures. Chapter II examines the history and characteristics of the TRAC language. The next chapter specifies the TRAC syntax and primitive functions. Chapter IV covers the algorithms used by the processor. The last chapter discusses the design experience from programming the processor, examines the reactive action caused by the processor, and suggests adding external storage primitive functions for a future version of the processor.

Bill Scott
May 20, 1974

TABLE OF CONTENTS

Page

LIST OF ILLUSTRACTIONS iv

Chapter

I. INTRODUCTION 1

Text Processing
TRAC Operations
Implementation of TRAC Processor
Summary of Chapters

II. REVIEW OF LITERATURE 6

III. DESCRIPTION OF TRAC 12

Syntactic Structure
Basic Primitive Functions
Editing of Partial Text String
Text Deletion
Diagnostic Facilities
Supplemented Primitive Functions
Secondary Storage Primitive Functions

IV. ALGORITHMS OF TRAC PROCESSOR 33

Scanner
Pushdown Automaton
Decoding of Primitive Functions
Dynamic Storage Allocation
Execution of Some Primitive Functions

V. CONCLUSIONS AND RECOMMENDATIONS 63

Linked Lists
Assembler and Simulator
Response Time
Implementation of Secondary Storage Primitive Functions

APPENDIX A. SAMPLE TRAC PROGRAMS 74

APPENDIX B. GLOSSARY OF TERMS 75

BIBLIOGRAPHY 76

LIST OF ILLUSTRATIONS

Figure	Page
1. System Flow of TRAC Processor	33
2. Scanner for TRAC	39
3. Flowchart of Decoding Routine	44
4. Diagram of Linked Allocation	48
5. Storage Organization of TRAC Processor	49
6. Structure of Form Pointer and Form Store	55

CHAPTER I

INTRODUCTION

Text processing plays a distinctive role in the field of information systems because of its generally non-arithmetic characteristic. The object of this thesis is to describe the design and implementation of a processor for TRAC, a conversational-text-handling language system for use with a reactive typewriter. TRAC (text reckoning and compiling) is the trademark and service mark of Rockford Research Institute Incorporated in connection with its standard computer controlling language (3).

Text Processing

Text is any combination of alphabetic, numeric, and special characters arranged in strings and other structures. Natural languages, computer programs and their data, personal letters, and even this sentence are all considered as text. Besides TRAC, there are several other languages designed specifically for processing string-oriented text, such as COMIT, IPL-V, and SNOBOL (3). However, apparently none of them was originally intended only for interactive on-line application.

TRAC Operations

The TRAC processor is able to accept and execute TRAC source programs. These programs can be procedures to operate on the text from the teletypewriter. The operations (2) include accepting, naming and storing a character string, modifying a string, concatenating strings, creating a macro skeleton from a string, treating any string at any time as an executable procedure, or as a name, or as text, and printing out any string known to the processor. Besides, the TRAC language allows the user to do integer arithmetic and Boolean operations, and provides diagnostic facilities.

Implementation of TRAC Processor

The TRAC processor is implemented on model 1 of the Fairchild F24 computer available in the Department of Computer Sciences, North Texas State University. This 4k memory computer system has a KSR teletype with paper-tape reader/punch, a card reader, and digital-to-analog converters as I/O devices.

The processor was written in Fairchild assembly language and was tested under a Fairchild assembler (1) and simulator (4) on the IBM OS/360 system of the University. During the test the Teletype input is simulated by punched card; the Teletype output is simulated by the printer. The results obtained from the test show only the simulated

reactive action, since neither the card reader nor the printer is responsive. The test result is then verified by the actual implementation of the processor in Fairchild F24 computer system.

The TRAC processor can be loaded into the Fairchild memory as a software text-processing package by the following steps (4):

1. The object code of the assembler is generated in punched card form.

2. The program loading routine is loaded into the F24 memory from paper tape.

3. The TRAC processor punched object deck is then read by the F24 card reader into the computer memory.

A copy of the TRAC processor object code is also available on paper tape punched out by the Teletype after the cards have been read into memory (5). This paper tape may later be reloaded into memory.

Summary of Chapters

The review of original definition and related material of the TRAC language system is the subject of the next chapter. The history, goal, characteristics, development, and implementation of the language are summarized.

The prospective users of the TRAC processor are expected to understand the structure of the TRAC language

by being exposed to the syntax and primitive functions briefly discussed in Chapter III.

Chapter IV describes the procedures of designing the processor, *i.e.*, the detection of syntactic units, the handling of various primitives, and the allocation of memory storage.

Chapter V examines the designing experience learned from the processor, offers some conclusions, and suggests future development for this conversational-text-processing system.

CHAPTER BIBLIOGRAPHY

1. Grimes, Glen T., "Design and Implementation of an Assembler for the Fairchild F24 computer," unpublished master's problem in lieu of thesis, Department of Computer Sciences, North Texas State University, Denton, Texas, 1973.

2. Mooers, C. N. and Deutsch, L.P., "TRAC, a Text Handling Language," Proceedings, Association for Computing Machinery Twentieth National Conference, (August, 1965), 229-246.

3. Sammet, J. E., Programming Language: History and Fundamentals, New Jersey, Prentice-Hall Inc., 1969.

4. The F24 simulator on the IBM 360 and the F24 program loader routines were provided by Prof. Dan W. Scott.

5. The memory-dump program was provided by Prof. Dan W. Scott.

CHAPTER II

REVIEW OF LITERATURE

TRAC was specified by C. N. Mooers in 1960 and first implemented by L. P. Deutsch in 1964 (5). One of their stated goals in designing this conversational text-handling language was that TRAC should be able to accept, name, manipulate, store, delete, and retrieve any Teletype character or string of characters.

In addition, TRAC was created to satisfy the following objectives (5):

1. It should allow the user to move any named string into a secondary storage device such as tape, disk, drum, to retrieve it at will, and to control the organization of the strings within the storage.

2. It is desired that TRAC would be operated as a component of an executive program to serve many users concurrently.

3. It should be easy for a user to recover from keyboard errors.

4. It should produce a simple and precise syntax independent of a line format on a page.

5. The format of TRAC input data should be

identical to the TRAC program; i.e., they are all strings of characters.

Mooers and Deutsch, at the same time, gave the motives for initiating the TRAC language. They found the existing string-processing languages, such as COMIT, LISP, IPL-V, unsatisfactory. They said that COMIT was rigid because procedures could not be modified at the keyboard during run time; LISP's details of syntax were inelegant in practice; IPL-V was like assembly language: mechanistic and not user-oriented. The main inspiration, however, came from the study about macro-assembly system by McIlroy and Eastwood (2, 3).

Although TRAC is a text-processing language, it is characterized as a language system built with macro capability. This means that portions of the user's program may be defined with formal parameters, stored, and when supplied with actual values for the parameters, can be called and operated upon when needed.

In the same paper (5), Mooers and Deutsch discussed the syntactic phase of TRAC. They defined the control characters and various functions used in the language. The algorithm for evaluation of a TRAC source program was briefly covered by them. It was described in detail later by Mooers (4).

Sammet (6), in 1969, gave a general introduction to the TRAC language. She said that TRAC was still too new for its long-range significance to be determined, but Mooers' work on TRAC extensions would make the language more powerful. Wegner (8) covered TRAC in his book, with emphasis on its macro attribute. Essentially their discussions were based upon the original TRAC language and no attempt was made to extend the ability of the language. In fact, Mooers vigorously opposes any modification of TRAC, as being a standard language.

Nevertheless, one of the distinguishing attributes of TRAC is its extendability. Levine (1) discussed three types of extended functions which would increase the usability of TRAC. They are RESISTORS functions, input/output, and graphic functions.

RESISTORS is the abbreviation of a club named the Radically Emphatic Students Interested in Science, Technology or Research Studies. It is a group of high school students of New Jersey interested in the TRAC language. RESISTOR functions allowed interchange of TRAC programs between computers and off-line storage for installations without mass-storage devices. Input/output functions were ideal for where the TRAC "select devices" primitive function would not be applicable. Graphic functions could write straight lines, initialize display pointers, and print x and y coordinates of a given point in the graph.

The graphic functions were reported by Teriault (1) in more detail.

According to Mooers, TRAC was designed to be machine-independent. This philosophy, Klein in 1964 (1) said, had been achieved by the implementation of TRAC at that time on the following machines: PDP-5, PDP-8, PDP-10, Honeywell DDP-516, IBM 360, Hewlett-Packard 2116. He studied the possibility of automatic and efficient translation of TRAC between machines. He predicted problems which would arise from the translation process,and, at the same time, suggested the methods to solve them.

The discussion of TRAC implementation from the hardware system point of view was explored by Wickham and Hamming (9). The configuration of the system they designed to support an on-line TRAC processor was composed of a central supervisory processor, the TRAC processor, some teleprocessing computers to handle simultaneous users, and a large storage. The concluded that the general structure of the TRAC language can place unusual demands on the resources of a computer system.

One of the most successful implementations of the TRAC language system was done by University Computing Company in 1969 for the FASBAC system. Their version of TRAC was called CASH (7). It embodied most of the functions and conventions of TRAC and extended the functions to some extent. CASH has a complete set of

file-handling primitive functions for processing information to and from secondary-storage devices; this was a practical deficiency in the original TRAC. The manual of CASH, from the user's point of view, provides clear and precise explanations of each TRAC primitive function, accompained by various examples which resolve several ambiguities of the TRAC language.

CHAPTER BIBLIOGRAPHY

1. Bosack, L., Eichenberger P., Klein B. Kuhn., Levine J., Theriault, D., and Young J., "TRAC language: Construction, Use and Philosophy," presented at the DECUS Symposium, Wakefield, Massachusetts, May 13, 1969.

2. Eastwood, D. E., and McIlroy, M. D., "Macro Compiler Modification of SAP," unpublished memorandum, Bell Telephone Laboratories, Computation Laboratory, 1960.

3. McIlroy, M. D., "Macro Instruction Extensions of Compiler Language," Association for Computing Machinery Communications, 3 (April, 1960), 214-220.

4. Mooers, C. N., "TRAC, a Procedure-Describing Language for the Reactive Typewriter," Association for Computing Machinery Communications, 9 (March, 1966), 215-219.

5. Mooers, C. N., and Deutsch, L. P., "TRAC, a Text Handling Language," Proceedings, Association for Computing Machinery Twentieth National Conference, (August, 1965), 229-246.

6. Sammet, J. E. Programming Languages:History and Fundamentals, New Jersey, Prentice-Hall Inc., 1969.

7. Scott, Dan W., CASH Language Primer, University Computing Company, Dallas, 1969.

8. Wegner, P., Programming Languages, Information Structure and Machine Organization, New York, Mcgraw-Hill, 1968.

9. Wickham, K., and Hemming C., "The TRAC Processor," unpublished paper for a course given by Dan W. Scott at Southern Methodist University, Dallas, December 1969.

CHAPTER III

DESCRIPTION OF TRAC

TRAC processor accepts text strings in the 7-bit American Standard Code for Information Interchange, abbreviated as ASCII. From the alphanumeric and special characters of ASCII the TRAC language defines its valid characters, control characters, strings of characters, primitive functions, and the arguments within the functions.

Syntactic Structure

The basic structure of TRAC language is defined in three types of string expressions designated by #(...), ##(...), and (...). The sharp sign and parentheses are syntactic control characters. The dots enclosed in parentheses are character strings which can be denoted by PF, $A_1, A_2, \ldots, A_k$, where PF is a mnemonic for some primitive function, $A_1, A_2, \ldots, A_k$ are arguments for the primitive function, and commas separate the arguments.

The first type of string expression #(PF,...) is called an active function. It returns a character string value after being evaluated. This value is to be further evaluated, and replaces the current string expression.

The second type of string expression ##(PF,...) is called a neutral function. It, like an active function,

returns a character string value after evaluation, but this value is not to be further evaluated.

The value returned by these two types of functions could be null. The null value is a string of no length.

The third type of string expression, (...), is called a quote function. The text string within parentheses can be an active or neutral function. It can also be any ordinary character or string of ASCII characters, other than certain exceptions for parenthesis characters. The evaluation of this quote-mode function results in copying the text string, without the enclosing parentheses, as the value. In fact the text string is protected from being evaluated.

Actually the format of text strings is more complicated than what has been said. The text may contain another function; it may consist of a pair of protected parentheses, or it may be a combination of both, and other variations. In general, the arguments of string expressions may also be string expressions.

For example, a TRAC program may have its text strings arranged in the following formats:

```
#( , ,##( ), ),
##( ,#( ), )
#( ,##( ), ,#( )),
```

where the specific mnemonics of primitive functions and arguments are left blank. From these examples it is

apparent that there is no restriction on the number of nested parentheses of a TRAC program.

The scanning of this nested structure is from left to right and from inside outward. It can be described by the following example:

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

The neutral function underscored by number 1 is first to be evaluated. The returned value becomes the second argument of the active function underscored by number 3. The active function underscored by number 2 is the next one to be evaluated. It returns a value which is to be further scanned and evaluated. The value obtained from further evaluation will be the third argument of the active function underscored by number 3, which in turn is the next function to be evaluated. The returned value occupies the position of the second argument of the active function underscored by number 4. This active function is the last one to be processed.

When a string is immediately followed by another string, it is said that they are concatenated. Concatenation of strings is quite simply indicated by their adjacency; *e.g.*, the concatenation of the four strings A, #(RS), ##(PS,TRAC), (Y) is written as the single string A#(RS)##(PS,TRAC)(Y). The scanning and evaluation of

concatenated strings follow the rule for simple strings except that the returned string values are also concatenated.

The scanning and evaluation of the string expressions of TRAC programs must terminate. This is indicated by an end-of-string symbol "'" at the end of the text string. This apostrophe character is called by Mooers the meta character. It may be dynamically redefined to be any other valid character in ASCII. The redefining primitive function is called "change meta character" and will be discussed in a succeeding section concerning text input and output.

Basic Primitive Functions

The TRAC language has six basic and important primitive functions to perform input and output of text strings, to define and to name the strings, to call, and to segment the strings.

Text Input and Output

The input operation is handled by the primitive function #(RS), where RS is the primitive-function mnemonic for "read string". This primitive function will read a string of characters from the teletypewriter up to an end-of-string symbol.

The output operation is taken care of by the primitive function #(PS,...), where PS is the

primitive-function mnemonic for "print string" and the three dots are the argument to be printed on the teletypewriter. For example, the execution of #(PS,THIS IS A TRAC STATEMENT)' will have the character string THIS IS A TRAC STATEMENT printed.

The nesting of input and output primitive functions #(PS,#(RS))' is called the idling string or idling routine. This string initiates the whole TRAC processing. It is initially loaded into a scratch area of TRAC processor memory. This area is called the active string. The execution of this string causes a text string ended by an end-of-string character to be read from the teletypewriter and to replace #(RS) as the second argument of the print string primitive function. If this input text includes functional statements, they are performed. At the end of the performance, if there is a non-null value returned from the functional statements, it is printed out by the execution of a print string primitive function. The print string primitive function itself returns a null value, so the active string becomes empty. At this time the TRAC processor again loads a new copy of the idling string into the active string and is ready for more input from the teletypewriter.

The primitive function "read string" performs inputting of character string of arbitrary length terminated by the meta character. TRAC also provides a

primitive function to read a single character typed on the teletypewriter. It is denoted as #(RC) and it means to "read one character", with no end-of-string character used to terminate the string.

It has been said that the character apostrophe indicates the end of text string. It can be changed into any other ASCII character, by the primitive function "change meta character", denoted by the mnemonic CM. For example, #(CM,##(RC))' will have the character entered on the Teletype to be the end-of-string indicator, and this character is printed on the teletypewriter.

Text Definition and Calling

Any input text in TRAC can be named and defined by assigning a name. This is done by the primitive function "define string" denoted by #(DS,N,A), where DS is the primitive-function mnemonic, S is the text string which is to be stored in memory, and N is the name assigned to the text string. For example, #(DS,AA,TRAC) defines a string named AA and associates it with the string value TRAC. It will be recalled that the arguments of DS may also be string expressions. If the name of the string AA is not changed and the three dots ... are used to represent string expression, then #(DS,AA,TRAC) can be specified in a more general format #(DS,AA,...).

In the TRAC language, the memory locations reserved specially for the strings defined by DS are called form store. Each named string defined in the form store is called a form. Form and defined string are used interchangeably.

The form AA defined by #(DS,AA,...) can be called upon by the TRAC primitive function #(CL,AA). The CL means "call string". The execution of #(CL,AA) copies the string named from form store and the resulting value replaces the string expression #(CL,AA).

Text Segmentation and Calling

The definition of a named string in form store can be modified by inserting, replacing, and deleting characters within the form. Before any modification is performed, the target character or characters of the form-store string must first be marked. This is done by the execution of the primitive function #(SS,N,X_1,X_2,...X_n), where SS is the primitive-function mnemonic for "segment string", N is the name of the string in form store, and X_1,X_2,...X_n are text arguments with which the text string is to be compared. The matched characters are marked and gaps in the form are created. The markers, in ascending ordinal value, are chosen from characters outside the ASCII character set to avoid confusion and duplication. Also, SS may be repeatedly applied to a given form.

For example, the TRAC statement #(SS,AA,C) will cause the character C in named string AA defined previously to be deleted throughout the string AA and in each case to be replaced by a mark of ordinal value. If it is then desired to change the value of string AA from TRAC to TRADE, the execution of #(CL,AA,DE) will accomplish the job. The form named AA is fetched from store and is searched to find the markers of ordinal value 1. The locations where the markers reside are replaced by the characters DE. That means that the contents of form AA have been changed from TRA@ to TRADE, where the character @, for convenience, indicates the marker of ordinal value 1. (The markers used in the TRAC processor are not printable.)

Note that #(CL,AA) has fewer arguments than #(CL,AA, DE). The former calls the string AA without editing; the latter calls with replacing. From this example, the format of the primitive function "call string" may be extended to #(CL,N,X_1,X_2,...,X_k), where the number of arguments is not limited, but cannot usefully exceed the maximum number of arguments used by SS on that form.

The values of arguments in the primitive functions SS and CL could be null. The detailed explanation of null arguments is covered by Mooers (1) and Scott (2).

Editing of Partial Text String

The SS and CL deal with the whole text string. The segmentation and retrieval are accomplished in terms of the entire string value. Beyond that scope, the TRAC language allows the user to manipulate portions of string. As mentioned before, each defined string is stored in form store. In order to perform partial string operations a pointer is needed to point at a given character in the sequence of characters in the string. This pointer is called the position pointer and is set at the first character of its string in form store by the define string primitive function. Its value is changed when the action of partial string editing is finished.

The first primitive function in this class is "call character", denoted as #(CC,N,A), where CC is the primitive-function mnemonic, N is the name of a defined string in form store, and Z is the value to be returned if the position pointer has reached the end of the string and there are no more characters to be called. After execution of the CC primitive function, the position pointer is advanced to the next character in the string. If a marker for segment gap is encountered, it is skipped and the next character in sequence is fetched. Using the previously defined string AA as an example, AA has value TRADE and the form pointer points at the first character T. The execution of #(CC,AA,EMPTY)' will give a character value

T and the position pointer is moved to point at the next character, R.

Not only can the user call a single character of a defined string, but he can call several characters. This is done by the primitive function #(CN,M,D,Z), where CN is the primitive-function mnemonic for "call N characters", M is the name of a string defined in form store, D is a string of digits, a decimal integer specifying the number of characters to be called, and Z is the value to be returned if the position pointer goes beyond the end of string. For example, #(CN,AA,2,E)' will call the previously defined form AA and return RA as the function value.

The integer N may also be negative. That means the characters to be called are those -N character positions to the left of the current position pointer. If -N characters are not available, Z is the function value. Thus the position pointer used by CN may be moved to the left or the right after evaluation.

Another primitive function to manipulate string within the form is "call segment". It has the format #(CS,N,Z), where CS is the primitive-function mnemonic, N is the form to be called, and Z is the function value if the position pointer has already reached the end of the string. This primitive function causes the defined form N to be searched until a segment gap is found. The

characters beginning with the one pointed to by the position pointer up to the one preceding the gap marker, become the returned function value. Then the position pointer is advanced to the character following the gap marker. If a segment gap is not found, a null string value is returned.

The next primitive function is called "initial". It is defined by #(IN,N,X,Z), where IN is the primitive-function mnemonic, N is the name of the string in form store, X is any character string, and Z is the function value returned in case the position pointer of the form is set at the end of the string. The "initial" primitive function will look for the first occurrence of character string X in form named N. If the match is found, the characters between the current position pointer of the form and X become the function value and the pointer is advanced to the first character beyond the matched string X. If no match is found, the string Z is the function value and the position pointer is not moved.

It has been mentioned that the value returned by a primitive function goes to either active string for re-scanning or neutral string,depending upon the mode (# or ##) of the function. For the primitive functions "call character", "call N character", "call segment", and "initial", however, there is an exception. That is, if Z is the returned value when the form pointer initially

points beyond the last character of the string, the value Z is to be placed in the active string and rescanned whether or not the mode of the function is neutral. This exception is defined by the original TRAC (1).

The execution of the primitive functions described in this section causes the form pointer to be moved. This pointer, however, may be reset to the first of the string in form store by performing the "call restore" primitive function designated by #(CR,N), where CR is the primitive-function mnemonic, N is the name of the form with which the form pointer is to be restored. For example, #(CR,AA)' will reset the form pointer of form AA to the first character T. Thus the definition of AA becomes TRADE again.

Text Deletion

The TRAC language can define a text string; it also can delete a text string. This is made possible by two primitive functions: "delete all" and "delete definition". The first deletion function is denoted by #(DA), where DA is a primitive-function mnemonic and the function is without arguments. The execution of this primitive function results in all the string definitions in form store being deleted. The second deletion function is denoted by #(DD,N_1,N_2,...,N_k). DD is the primitive-function mnemonic and it means to delete the form definitions specified by the arguments N_1,N_2,...N_k. For

example, if the form AA is to be deleted, it is accomplished by issuing #(DD,AA)'.

Additional Primitive Functions

In addition to primitive functions strictly for text string processing, TRAC furnishes a limited number of primitive functions to do comparison of strings, integer arithmetic, and Boolean operation. These functions are briefly described as follows.

Decision-Making Commands

The TRAC language provides two primitive functions for comparison of arbitrary strings and arithmetic values. The string comparison is done by the primitive function "equal" denoted by #(EQ,S1,S2,YES,NO), where EQ is the primitive-function mnemonic, and S1 and S2 are two strings. If S1 and S2 are identical character by character, the argument YES will be the value (segment gaps are ignored in the comparison). If S1 and S2 are not identical, the argument specified by NO is the function value.

The integer comparison is made by primitive function "greater than", designated by #(GR,I1,I2,YES,NO), where GR is the primitive-function mnemonic and I1 and I2 are character strings defining integer values. If the value of I1 is greater than the value of I2, then the argument specified by YES will be the function value. If the value

of I1 is equal to or less than the value of I2, then the argument specified by NO is the function value.

For example, the evaluation of #(EQ,14,14,EE,NN) results in a string value EE stored in the active string. The execution of #(EQ,30,5,100,99) returns string value 99 in the active string. The performance of #(GR,89,0,YY,NN) results in the value YY stored in the active string. The processing of #(GR,99,99,AA,BB) ends up with the string BB in the active string.

Boolean Operations

The TRAC language can handle Boolean operations on vectors of 1's and 0's. Instead of representing the Boolean elements in pure binary form, TRAC uses a sequence of octal-digit characters. The octal digits are defined as the value of a group of three binary digits. For example, the binary number 101 is equivalent to 5 in base eight and 110101 is equal to 65 in base eight. There are five TRAC primitive functions for performing Boolean operations. They are union, intersection, complement, shift, and rotation of bit sequence.

The union and intersection actions are denoted by #(BU,O1,O2) and #(BI,O1,O2) respectively. BU and BI are the primitive-function mnemonics for "Boolean union" and "Boolean intersection" and O1 and O2 are octal digits. The value after evaluation of the function is also an

octal digit character string. For example, #(BU,5,5) gives an octal character string value 5 and #(BI,14,10) returns an octal character string value 10.

The complement operation is represented by #(BC,01), where BC is the primitive-function mnemonic for "Boolean complement", 01 is the only argument, a string of octal-digit characters. The resulting value after executing the function is the bit-by-bit complement or inverse of the bits of the value of 01. For example, #(BC,5) gives a character-string value 2 base eight (i.e., 010 in binary).

The shift and rotate primitive functions have a choice for the direction of shifting and rotation. That is, octal digits representing binary digits can be shifted left or right. They have formats: #(BS,D,01) and #(BR,D,01). BS is the primitive-function mnemonic for "Boolean shift"; BR means "Boolean rotate"; 01 is the character string of octal digits whose bit value is to be shifted D positions. If the integer D is positive, the bit-shift operations are done to the left. If D is negative, the bit shift operations are done to the right. Any leading nondecimal character in the argument string D other than a minus sign is ignored. For example, #(BS,2,31)' returns value 44, #(BS,-5,654)' gives 015; #(BR,3,61)' gives 16; #(BR,-4,53)' returns 56.

Arithmetic Primitive Functions

TRAC provides facilities to do simple integer addition, subtraction, multiplication, and division. These are dentoed by #(AD,D1,D2,Z), #(SU,D1,D1,Z), #(ML,D1,D2,Z) and #(DV,D1,D2,Z,R). In these primitive functions the first arguments specify the primitive-function mnemonic, D1 and D2 are strings of decimal characters, and Z is the function string value to be returned if arithmetic computation results in overflow (i.e., an incorrect result). The condition of overflow is raised when the calculation causes a number greater than the decimal numer $2^{23}-1$ or 838608,since the Fairchild computer has a 24-bit accumulator where the computation is performed. The argument R of the DV primitive function is used to store the remainder after the division operation. This is not defined in the original TRAC language. Note that any leading nondecimal character in D1 is attached to the first of the result string; in D2 leading non-decimal characters are ignored during the evaluation.

The following examples of the four arithmetic operations are self-explanatory: #(AD,15,4,OVER)' gives 19; #(SU,100,99,ONE)' gives 1; #(ML,EA4,67,@*)' gives EA268; #(ML,100000,100000,BURP)' gives BURP; #(DV,33,Z4, QT,RM)' gives 8. In decimal division, only the integer portion of quotient 8 is kept as the returned value. The

remainder 1 is stored in the argument specified as RM, which is defined to be a form in form store.

Diagnostic Facilities

TRAC supplies several primitive functions as debugging aids for the user. One of the primitive functions defined in the TRAC language will list all the names of strings defined in form store. In the listing, each of these names is preceded by some characters chosen by the user. This primitive function is denoted by #(LN,C), where LN is the primitive-function mnemonic for "list name", C is any valid character or string of characters to be attached to each of the names. Assume that two forms have been defined: AA and BB. The statement #(LN, ---) results in the character string ---AA---BB printed on the Teletype. If the string --- is replaced by the two carriage-control characters, carriage return and line feed, the names of the forms will be listed columnwise.

Furthermore, TRAC can also print the string values of forms defined in form store. This primitive function is "print form definition" and is denoted by #(PF,N), where PF is the primitive-function mnemonic and N is name of the form to be printed. If there is any marker for segment gap in the string, it is indicated by ordinal value in the output. Not only can this function list the content of form, but it specifies the computer memory address

where the form is located in the form store. Note that this address is not the relative position of the form name N in TRAC program.

By execution of primitive function "trace on", the neutral strings for each function are typed out on the Teletype. In the Teletype output the step-by-step evaluation of each primitive function is presented with all the intermediate results. This trace can be terminated by executing the primitive function #(TF). The mnemonic means "trace off". These two tracing primitive functions can be turned on or off any time during the execution of TRAC program. Initially the trace is assumed to be off.

Supplemented Primitive Functions

All the primitive functions discussed so far are defined in original TRAC language (1). In this TRAC processor, three more additional primitive functions are added to the system. They are "decode character", "encode character", and "implied call".

The primitive function "decode character" is denoted by #(DC,X). The value of this primitive function is the character string representing the value (base ten) of the ASCII representation of the first character of the form whose name is indicated by X. Taking the form named AA as an example, the execution of #(DC,AA)' will return value 84,which is the decimal equivalent of ASCII code for the

character T in the form string value TRADE. It is sometimes useful to convert alphanumeric names into numbers which can be utilized for external storage addresses.

The primitive function "encode character" is the converse of "decode character". The value after encoding of a numeric value is a character. The format is #(EC,D) where EC is the primitive-function mnemonic and D is a string representing the number to encoded. For example, #(EC,49), after execution, returns the ASCII character 1 (the character 1 has the binary value 0110001 in ASCII).

If the first argument of a string expression does not match any primitive function that has been defined in the TRAC language, it is assumed to be an "implied call". The mnemonic is the name of the form and this form is to be searched in form store. The value will be definition of the form with segment gaps filled out as in CL.

During the scanning and evaluation of TRAC primitive functions presented in this chapter, it is possible that the processor is in infinite iteration or loop due to the user's error. The break key on Teletype can stop the action and cause the reinitialization of the processor. As a matter of fact, any action, not necessarily a loop, at any time can be stopped by merely a touch of the break key. This break key is defined by Mooers. For this processor, the break key is replaced by the key ← on Teletype because when the break key on the F24 Teletype

is punched, the content entered into the accumulator is not stable.

Auxiliary Storage Primitive Function

All but three of the primitive functions defined in original TRAC language (2) are included in the F24 TRAC processor. These three primitive functions not implemented perform storing, fetching, and erasing forms between the main storage and the auxiliary storage devices.

The primitive function "store block", denoted as #(SB, N,N1,N2,...Nk), stores the forms N1,N2,...,Nk as a single block of record in secondary storage devices. When the forms have been put into the external storage, they are erased from form store in memory. A new form named N is created with its string value as the address of the block in external storage. The forms stored in external storage in the block whose address is in the form named N can be retrieved by the primitive function "fetch block", denoted as #(FB,N), and can be erased by the primitive function "erase block", denoted as #(EB,N).

This chapter has described the syntax elements and various primitive functions of the TRAC language. Several TRAC examples obtained from the Teletype outputs are included in Appendix A to demonstrate the behavior of the language.

CHAPTER BIBLIOGRAPHY

1. Mooers, C. N., "TRAC, a Procedure-Describing Language for the Reactive Typewriter," *Association for Computing Machinery Communications*, 9 (March, 1966), 215-219.

2. Scott, Dan W., *CASH Language Primer*, University Computing Company, Dallas, 1969.

CHAPTER IV

ALGORITHMS OF TRAC PROCESSOR

This chapter illustrates the TRAC processor algorithms which were used in this implementation of TRAC on the F24 computer. This description includes the steps for scanning the TRAC programs, the techniques for dynamic-storage allocation, and the specific features for handling some primitive functions. The system flow of the processor is shown in Figure 1.

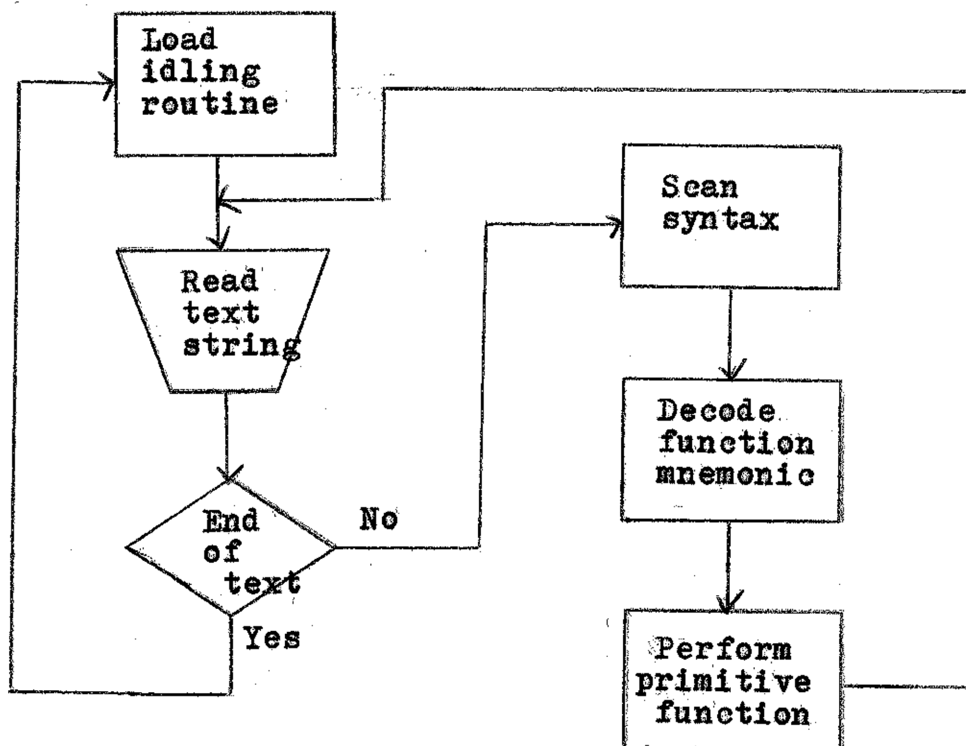

Fig. 1--System flow of TRAC processor

Scanner

The unevaluated TRAC text strings are loaded in the active string at the beginning. The active string is a reserved area of the TRAC processor memory. The functions of the scanner are to recognize the syntactic atoms of the loaded text string, to detect the control characters, i.e., comma, left and right parentheses, sharp, tabulation, carriage return, and line-feed characters, and to take proper action depending upon the control character scanned. Furthermore, the scanner distinguishes active function from neutral-function values.

A pointer is maintained within the scanner. This pointer moves from left to right in the active string during the scanning process. When the current active string is empty, i.e., there are no characters remaining in the active string, the pointer is reset to point at the new active string initiated by the idling string.

As characters of text string in active string are treated by the scanner, some of them may be appended to the right end of the neutral string, which is another area of memory reserved for storing strings of characters. It is called neutral because characters in the string have been operated (scanned) and thus remain neutral.

Mooers (6) described the TRAC processing algorithm in early 1966. The scanner is essentially based upon his description but modification has been made in order to be

compatible with the linked-list structure of the TRAC processor. This linked organization will be described later in this chapter.

The modification is as follows: during the scanning, extra consideration is given when a comma is encountered in the active string. It has been mentioned that the arguments of a string expression are delimited by comma, and the arguments could be characters, active functions, neutral functions, or quote functions. For example, in #(SU,#(AD,3,5,SSS),7,NUM), the commas within #(AD,3,5,SSS) delimit simple character arguments; the comma between SU and #(AD,3,5,SSS) separates a primitive-function mnemonic and an active function. The address of the comma succeeding SU is first saved and then linked with the returned string value 8 resulting from evaluation of #(AD,3,5,SSS). In another example, #(DS,AA,(ZZZZ)), the comma after AA is first saved and then is linked with the string ZZZZ of quote function. Though it is said that the address of comma is saved, it is actually the address of delimiter for the comma in neutral string being remembered. This will be seen in steps 5, 6, 7 of the scanner.

A step-of-step description of scanner is presented as follows:

1. Test active string. If it is empty, go to step 11; otherwise, get current character from the active string and go to step 2.

2. If the character being examined in the active string is a left parenthesis, go to step 3. Otherwise, go to step 4.

3. The scanning pointer is moved ahead to the next character in the active string until the matching right parenthesis is found. Then the character string between the matching parentheses is appended to the right end of the neutral string. Go to step 10. Step 3 thus recognizes the quote-mode function.

4. If the character being examined in the active string is either a carriage return, a line feed, or a tabulation, go to step 10; otherwise, go to step 5.

5. If the character being examined in the active string is not a comma, go to step 6. If it is a comma, the location to the rightmost character of the present neutral string is marked by a delimiter and a flag is set to indicate a comma is found. Go to step 10.

6. If the character being examined in the active string is not a sharp sign, go to step 8. If it is a sharp, the scanning pointer is moved ahead to scan the next character. If the next character after the sharp is not a left parenthesis, go to step 7. If it is a left parenthesis, the beginning of an active function is indicated. The sharp and the left parenthesis are ignored 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. Then test the flag for comma. If it is not on, bypass the next sentence. If it is on, the address of corresponding delimiter in neutral string is saved as a linking address and the flag is reset. The scanning pointer is moved to the character following the discarded parenthesis. Go to step 10.

7. If the character succeeding a sharp sign is not a sharp sign, go to step 9. If it is also a sharp sign, the scanning pointer is moved to point at the next character in the active string. If the next character is not a left parenthesis, go to step 9. If it is a left parenthesis, a neutral function has been encountered. At this time, the current location in the neutral string is marked to denote the beginning of a neutral function and the beginning of an argument substring. Then test the flag for comma. If it is not on, go to step 10. If it is on, the address of corresponding delimiter in neutral string is saved as a linking address and the flag is reset. Go to step 10.

8. If the character being examined in the active string is not a right parenthesis, go to step 9. If it is a right parenthesis, it triggers the execution of the function. The current location in the active string is marked as the end of an argument substring and the end of a string expression. Go to the function-decoding routine, which is discussed in a later section.

9. Move the character to the right end of neutral string. This character is not a candidate for being an indicator for either active or neutral function. Go to step 10.

10. Advance the active string scanning pointer one character position and go to step 1.

11. Load a new copy of idling routine into the active string. Reset the scanning pointer at the beginning of the active string. The current neutral string is deleted (the scanning pointer for neutral string is moved during the function evaluation).

The flowchart of the scanner is shown in Figures 2.1, 2.2. For simplicity, in the flowchart, the word "advance" means to advance the scanning pointer of active string by one character position; the word "CC" means the current character of the active string.

Pushdown Automaton

During the scanning procedure, a pair of properly matched parentheses in the text string must be found to indicate that the primitive function is ready to be executed. The scanner uses a pushdown stack to accomplish this. This pushdown stack is indexable and is initially empty. Each time a left parenthesis is encountered in the text string, the address of next available storage in the neutral string is stored (pushed) on the stack. The first memory cell in the neutral string stores the first character of the

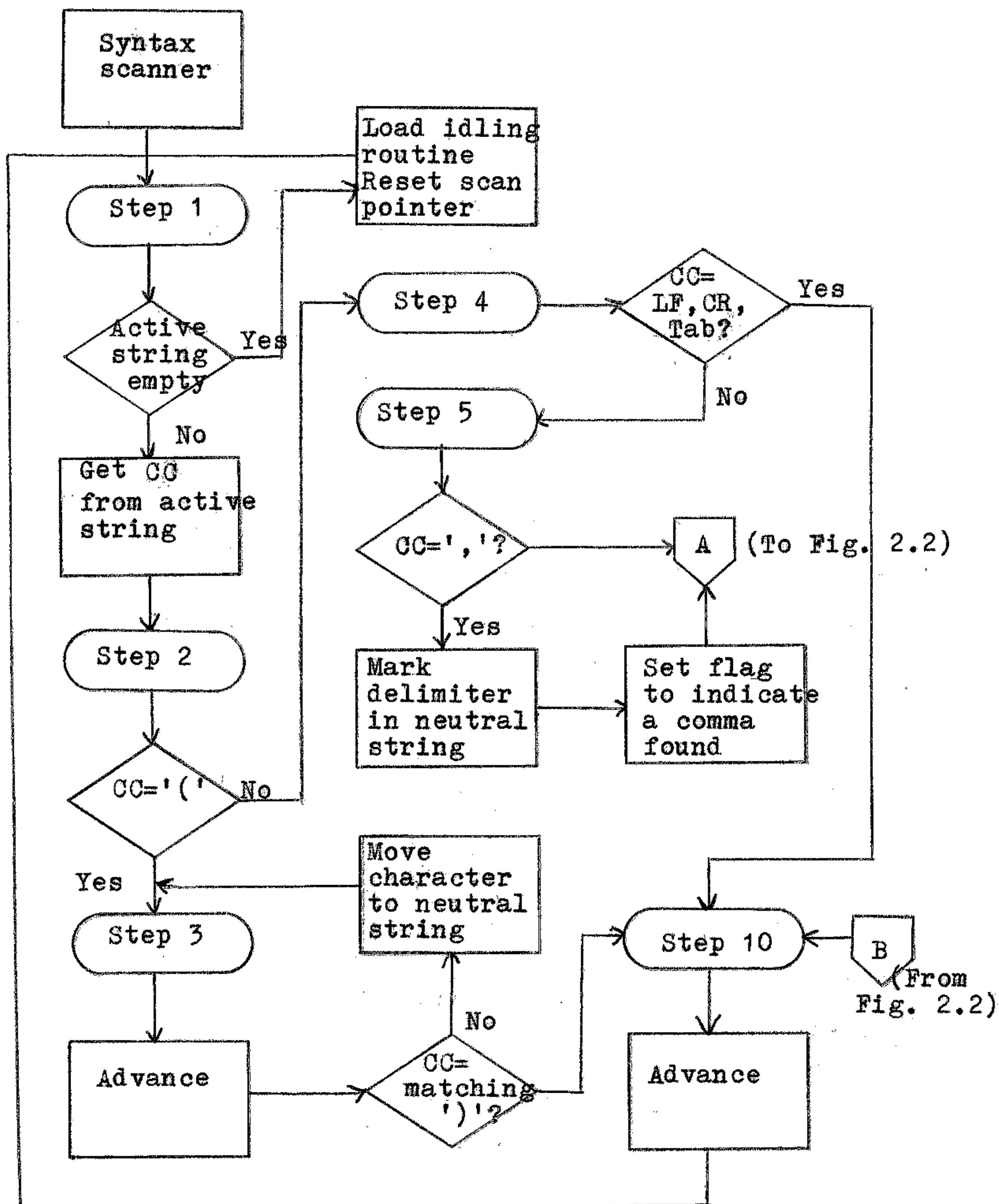

Fig. 2.1--Scanner for TRAC

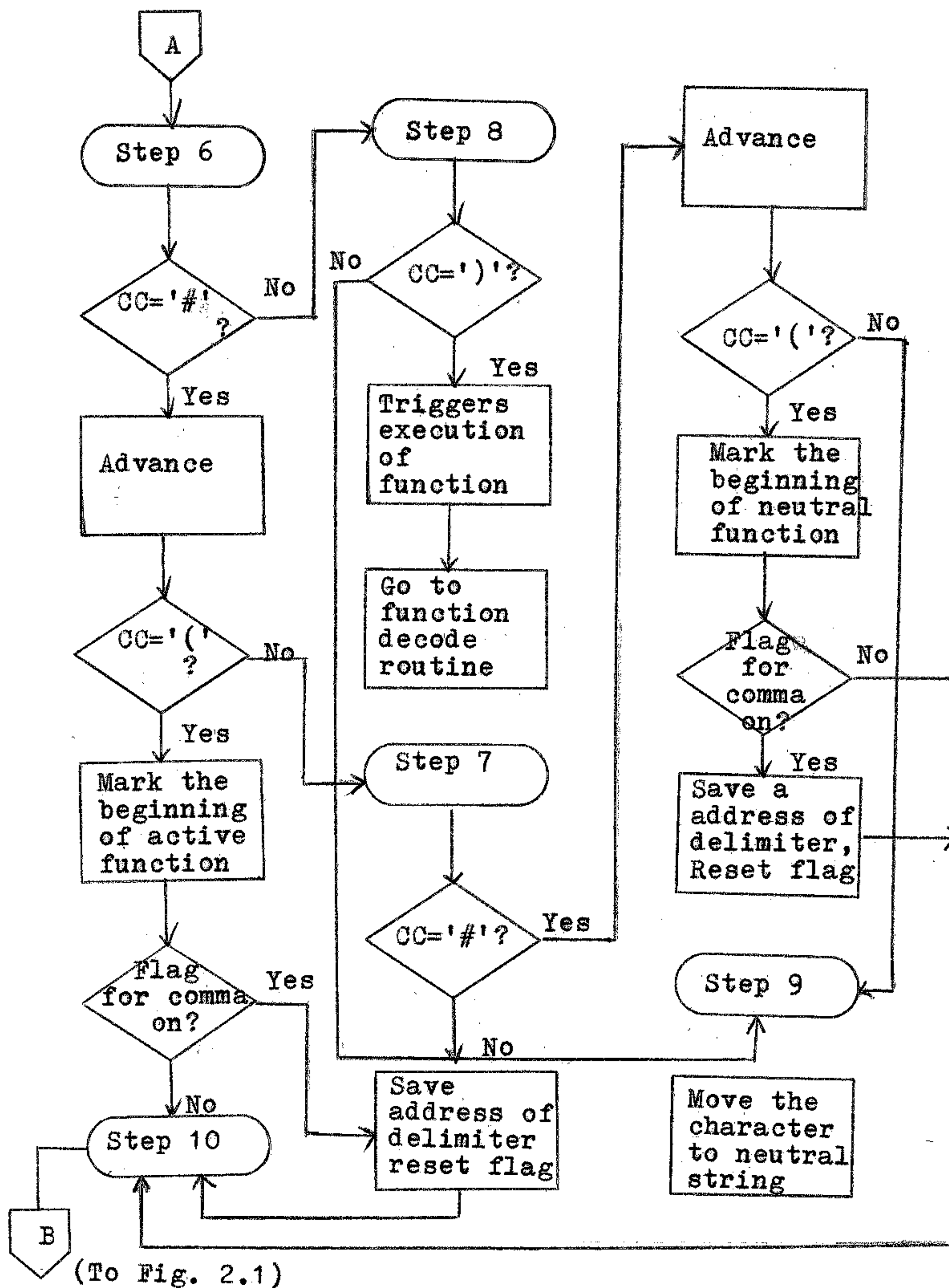

Fig. 2.2--Scanner for TRAC

primitive-function mnemonic for a primitive function or, in case of "implied call", the first character of the name of a defined string in form store.

The indexable address in the pushdown stack is decreased by one whenever a matching right parenthesis is being scanned. When the stack becomes empty, it indicates the end of performance of a primitive function. At this time the scanner continues looking at the next character in the active string. The description of storage allocation for the pushdown stack to keep track of matching parentheses is discussed in a later section.

Decoding of Primitive Functions

When a complete string expression has been recognized and is ready for execution, the two-letter mnemonic for the specific primitive function is decoded. The TRAC processor is able to handle thirty-two primitive functions, as described in Chapter III.

Among them, twenty-nine are originally defined in TRAC language (6); three are added primitive functions in this processor. They are divided into eight categories according to the first character of the mnemonic.

The first category includes those primitive names starting with the letter R. They are primitive functions for input, RS (read string) and RC (read character).

The second category of primitive functions begins

with the letter D. They consist of DS (define string), DA (delete all forms), DD (delete definition), DC (decode character), and the arithmetic operation DV (divide).

The mnemonics starting the letter C belong to the third category. They are CL (call string), CC (call a character), CS (call segment), CN (call N characters), and CM (change meta character).

The mnemonics beginning with the letter S are classified as the fourth category. They are SS (segment string) and SU (subtract).

The fifth category embodies the Boolean primitive functions. They are BU (Boolean union), BI (Boolean intersection), BC (Boolean complement), BS (Boolean shift), and BR (Boolean rotate).

The sixth category includes two primitive mnemonics starting with the letter E. They are EQ (equal) and EC (encode character).

The mnemonics beginning with the letter P are the seventh category. They are PS (print string) and PF (print definition).

The last category includes two primitive functions in which the first character of mnemonics is unique among the primitive functions. They are GR (greater than) and IN (initial).

If the first character of mnemonic does not conform to any of the above eight categories or the first character

does agree but not the second, it is treated as an implied call primitive function. The mnemonic in this case is the name of the defined string in form store to be called upon. For example, #(DS,G,YYY)#(G)' will return string value YYY on Teletype; #(DS,GZ,ZZZ)#(GZ)' will response string ZZZ on Teletype. Furthermore, if both characters of the mnemonic belong to one of the defined categories but the character after the two-letter mnemonic is not a comma, that means the length of the mnemonic is greater than three; then all the characters of the mnemonic ended by a comma or a right parenthesis are also considered as the argument of an "implied call". However, carriage return, line feed, and tabulate are ignored and will not be counted as part of the mnemonic. For example, #(DS,GRX,XXX)#(GRX)' will return a string value XXX.

The flowchart of the decoding routine is illustrated in Figures 3.1, 3.2, 3.3.

Dynamic Storage Allocation

The allocation of storage is a significant and vital portion of the task for developing any compiling or interpretive type of processor. There are various methods to handle the fixed amount of available memory so as to obtain the maximum efficiency of storage utilization (2, 5). For the TRAC processor, the allocation of storage is manipulated dynamically in combination with lists of pointers.

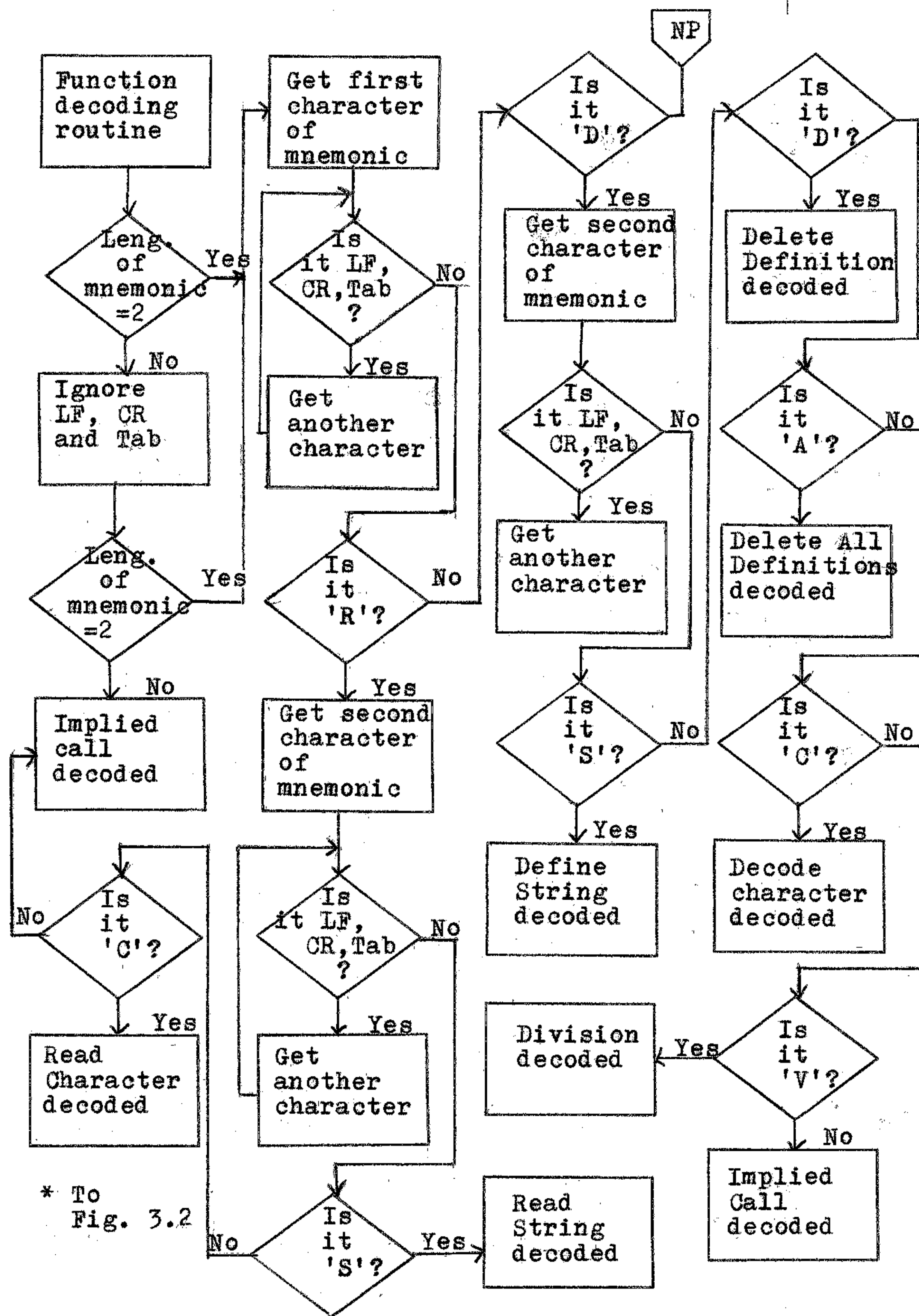

Fig. 3.1--Flowchart of decoding routine

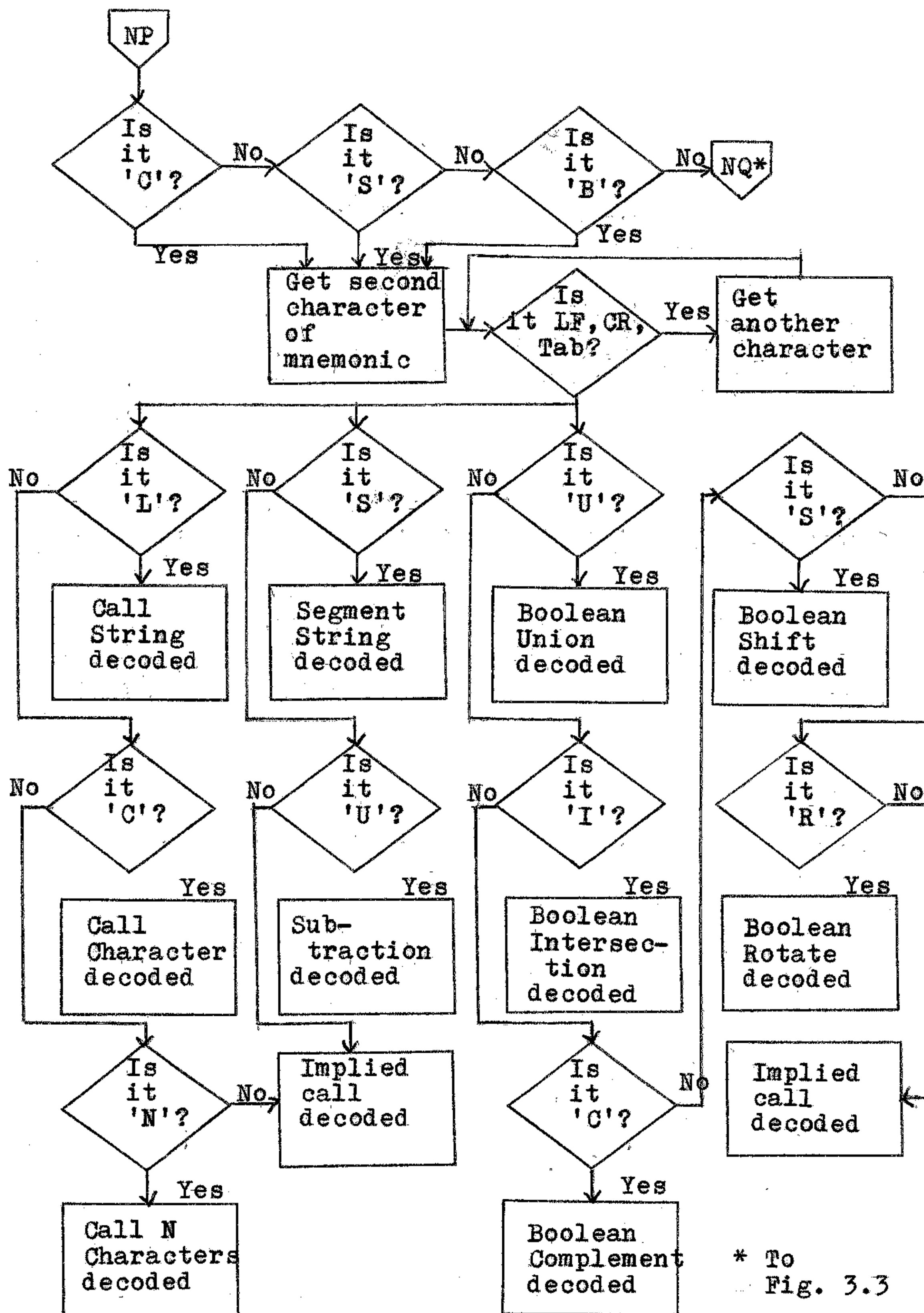

Fig. 3.2--Flowchart of decoding routine

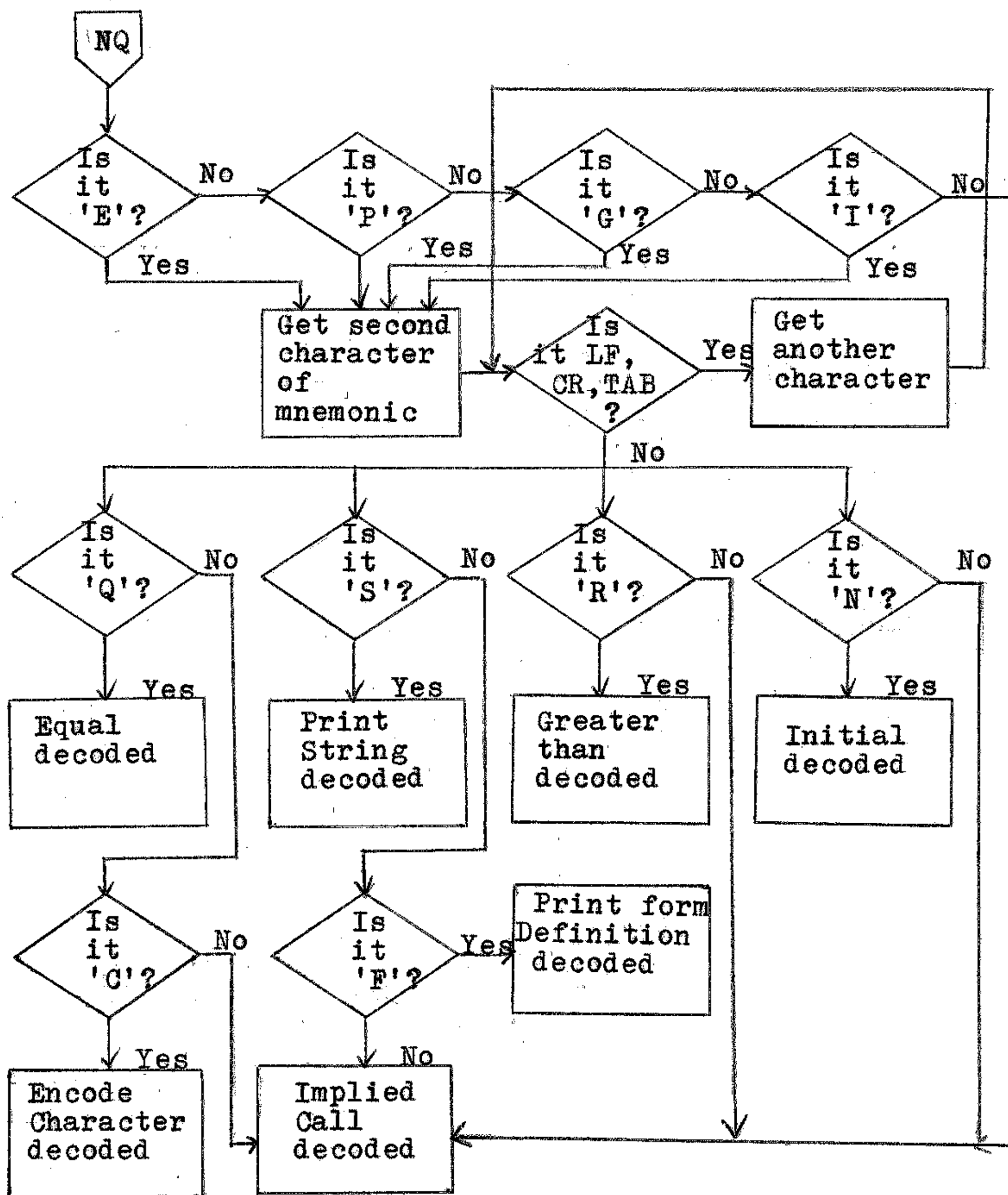

Fig. 3.3--Flowchart of decoding routine

Structure of Linked Pointers

The internal structure of the TRAC processor is based upon the linked lists of pointers. Each memory location contains a linking address to another cell which is not necessarily next to the current memory location.

The basic information unit of the Fairchild F24 computer is a twenty-four-bit word. The processor, in general, uses the rightmost twelve bits (bit 0-11) of the word to store the address of next element in the list. The ten bits (bit 12-21) to the left of the above twelve bits store the data item. The leftmost bit, that is, bit 23, is not used. The remaining bit 22 is, most of the time, not used but acts as a control during the scanning to indicate the mode of the primitive function. If a neutral function is recognized by the scanner, bit 22 of the current character in neutral string is set to 0. After the function evaluation, the returned string value goes to either active string or neutral string, depending upon the status of bit 22 of this specific character in neutral string. The general structure of the linked allocation is shown in Figure 4.

Shared Memory

Among the available 4096 ("4k") words of memory of the Fairchild F24 computer, the first 1000 locations are reserved for system loader and card loader, and approximately 2000 memory locations are occupied by the TRAC processor. The

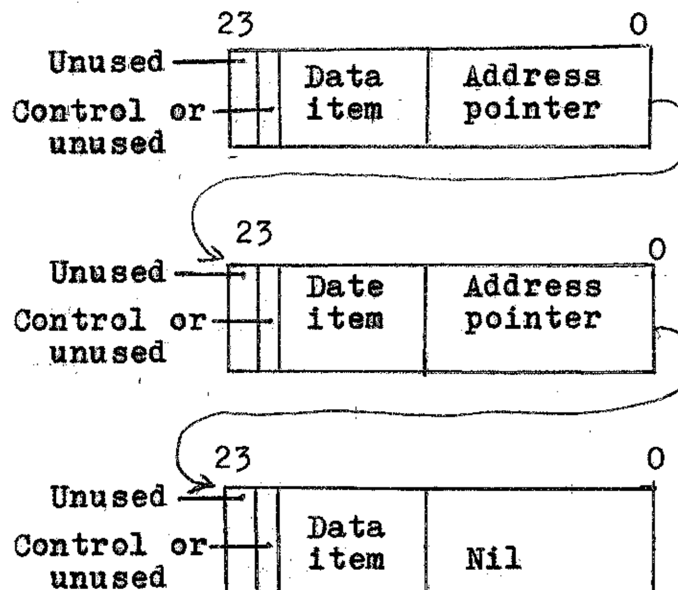

Fig. 4--Diagram of linked allocation

remaining 1000 cells are allocated for active string, neutral string, form store, intermediate result, and the pushdown stack used to search for matching parentheses in a TRAC text string. The organization of the memory allocation is shown in Figure 5.

Initially the TRAC processor reserves two hundred locations for the pushdown stack. This allows a single active string to contain up to two hundred left parentheses prior to the first right parenthesis.

Assigning a fixed amount of memory for the scanning stack at the beginning, frees the processor to deal with more complicated storage allocation policies for active string, neutral string, and form store without considering whether there are enough memory locations for storing the addresses of the left parentheses in active string. This

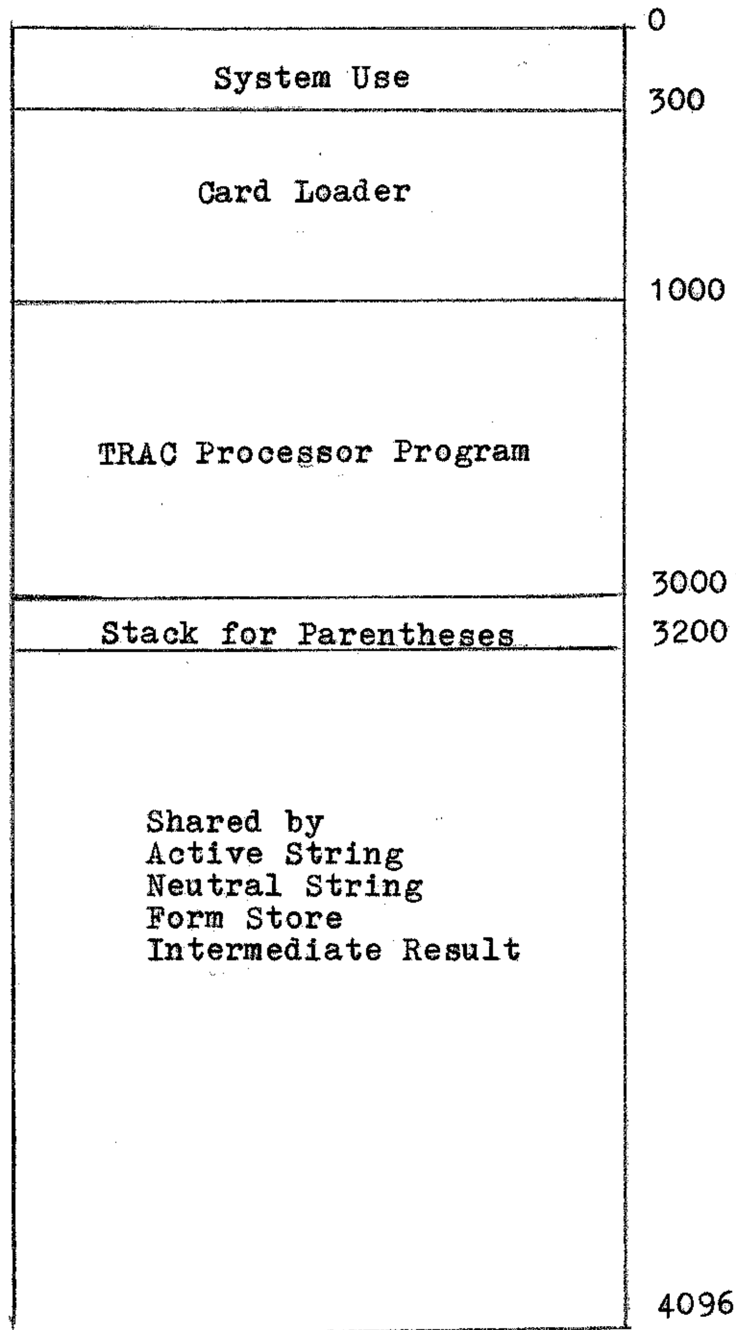

Fig. 5--Storage organization of TRAC processor

approach, by which portions of memory are permanently reserved for specific use, results in reduction of the time spent on the dynamic allocation and garbage collection. Garbage collection refers to the release of portions of memory which are no longer required. The description of releasing memory for the TRAC processor immediately follows this section.

The remaining storage, once the stack is allocated, is shared by active string, neutral string, form store, and intermediate results. The big available space is not divided into different sizes of blocks, as has usually been implemented (3). Rather, it is occupied by arbitrary length blocks in a first-come, first-served order, along with a very disciplined use of pointers.

This free space is allocated sequentially. A pointer, called AVAIL, points at the next available cell of free space. No matter where the current memory location is, the next free location always follows it, except that when the entire 4k memory is full, garbage collection is performed.

Garbage Collection

When the available 4k memory is fully occupied, garbage collection is performed to release all previously used memory which is not in current use and hence is not to be retained. The restructuring is accomplished by moving the

contents of all memory cells currently in use to one end of available storage. The released memory locations at the other end of storage then become available for further allocation.

The procedure restructuring memory begins with the memory cell next to the last location of the stack. This location is defined to be the head of the pointer list for defined form store. (If the head is zero, it means that the form store is empty and no named string has yet been defined.) The address of the next available free location starts at the cell following the head. The pointer AVAIL is set to point at this place. At the same time, the address pointer of the last memory location in 4k storage is adjusted to point at this new free location.

If the head of the pointer list of defined form store is not zero, it indicates that there are defined strings in form store. In this case, all named strings in form store are moved forward consecutively to become a group of named strings together instead of being in scattered form as prior to the performance of garbage collection. During the movement the address pointers of defined string in form store are properly modified to point at the newly released memory locations.

The active and neutral strings currently in use are not subject to garbage collection. However, if the last address pointer, when the memory becomes full, is in either

active or neutral string, it is to be updated to point at the just released memory next to the form pointer list.

Execution of Some Primitive Functions

The following two sections single out the techniques for handling the performance of two types of primitive functions. The first type is those primitive functions featuring macro attributes, such as define string, segment string, and call primitive functions. They are some of the most important and powerful primitive functions defined in the TRAC language. The second type is those primitive functions calculating arithmetic integers, such as addition, subtraction, multiplication, and division. The operands of the latter are in the form of numeric characters. During their execution conversion is required to transform them from the character string format into an internal numeric form for arithmetic.

Macro Primitive Functions

It has been mentioned that TRAC is a language coupled with macro ability. The primitive function DS (define string) defines the macro; SS (segment string) supplies parameters for the macro definition; CL (call string) invokes the calling of a macro.

The DS (define string) primitive function associates a string name with a string value retained in form store.

The TRAC processor keeps a list of pointers to point at the addresses of the defined strings in form store.

There is a head of the form-name list and tail of the form-name list. Initially, both the head and the tail have a zero value. When the first DS primitive function is encountered in an input program, the rightmost twelve bits of the next available memory location pointed to by the pointer AVAIL, are set to contain the address of next free location. For convenience, the address of the free location is called the form pointer. Beginning with this location, the name of the defined string and its value are allocated consecutively in free space and, meantime, they are delimited by a memory cell. Then the AVAIL pointer is advanced to point at the next free location which is neighboring the last character of the form-string value. Also the address of the current form is stored at the head of the list of form store.

When the next DS primitive function is executed from an active string, the above allocation procedure is repeated except that the address of the current form pointer is not stored at the head. It is, however, stored as the leftmost twelve bits of the previous form pointer. This linked list of form pointers provides a searching list for defined strings in form store. The rightmost twelve bits of each form pointer point to the current named string and the leftmost twelve bits point to the next

named string. If the leftmost twelve bit group of the form store is zero, it signifies the end of the list of names of defined strings in form store. The diagram of this linked list of form pointer is presented in Figure 6.

The SS (segment string) primitive function deletes and marks the substrings of a previously defined string in form store when they match the argument substring given in the SS statement. The first step to perform this primitive function is to find the named string in form store (the name being the first argument of the SS) by searching through the form pointer list. The next step is to mark the matched substrings. The marker is a character chosen in ascending ordinal value outside the valid character set in ASCII to avoid confusion with valid user characters.

If more than one substrings of form match the argument substring given in SS primitive function, each of the matched substring of form store is marked by the same ordinal value. For example, #(DS,K,ABCABZZAABLBAB)#(SS,K, AB)' causes four substrings AB of form K (underlined) to be marked by the identical ordinal value 1.

The SS primitive function accepts multiple substring arguments, such as #(DS,A,HAAZAHAALALA)#(SS,A,HAA,ZA,LA). When the substring argument HAA of SS is encountered in the neutral string during evaluation, two substrings HAA of form A are marked by ordinal value 1. When the next

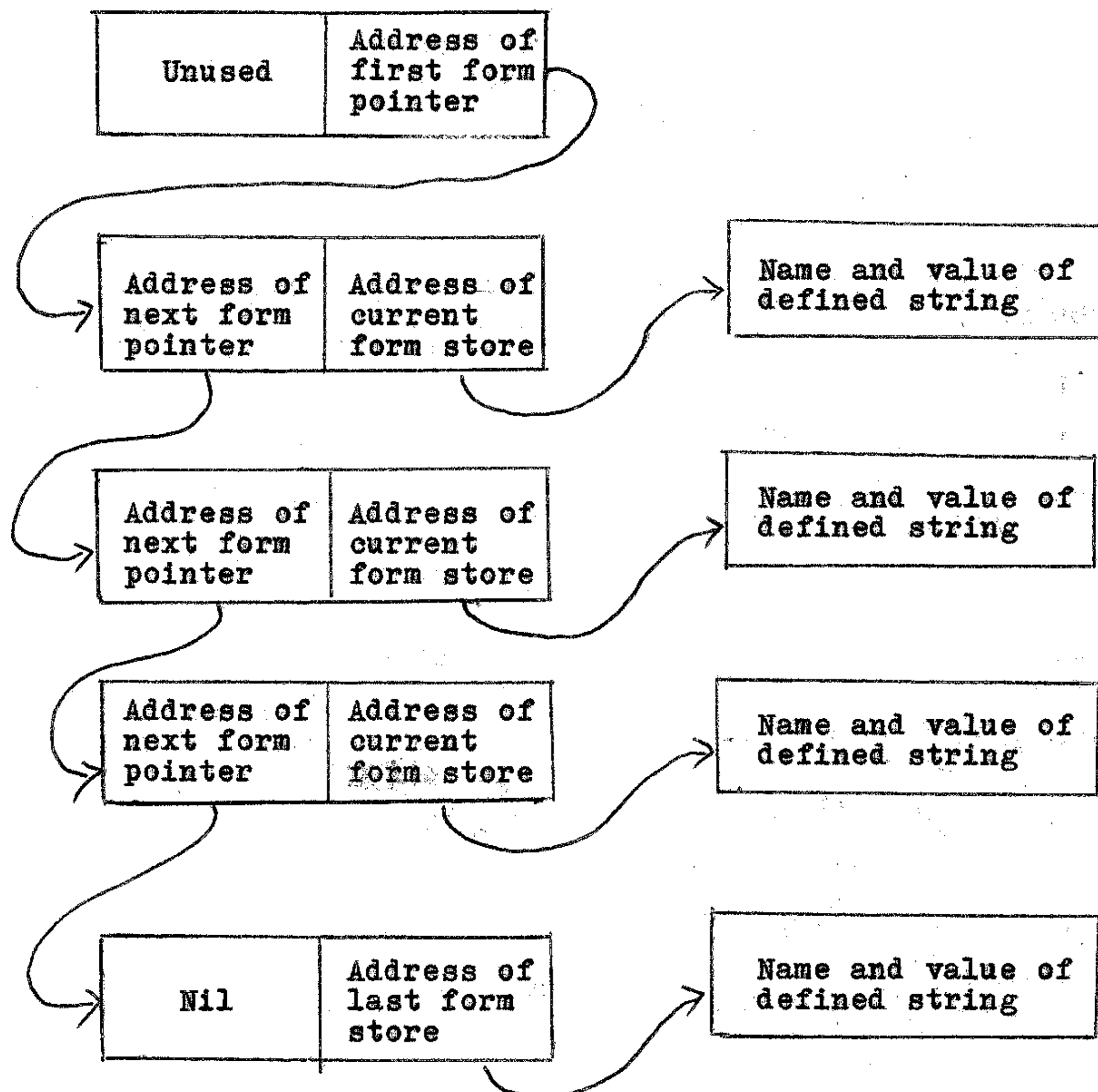

Fig. 6--Structure of form pointer and form store

substring argument ZA of SS is encountered, the only substring ZA of form A is marked by ordinal value 2. So is with the third substring argument LA except that two substrings LA of form A are marked by ordinal value 3.

The CL (call string) primitive function examines the occurrence of a named string in form store and returns the characters of the string as its value. As described in Chapter III, CL has two different appearances: without substring arguments such as #(CL,AA), and with substring arguments such as #(CL,BB,X,Y,Z).

For CL without substring arguments a search is first performed through the list of form store to find the defined string and then the characters of the string are returned as the value of CL.

For CL with substring arguments searching is not the first thing to be done. Rather, the substring arguments are first stored in the free area allocated to them and each of them is appended by a marker. The markers are in ascending order according to their relative position as substring arguments in CL primitive function. After substring arguments have been allocated and preceded by markers, the form store is searched to find the named string. When it is found in form store and consists of gap markers, the gap markers (which have certain ordinal values) are replaced by the substring arguments of corresponding ordinal values. The replacement, however,

is a pseudo-operation and no replacement is actually taking place. In fact, the characters of form, if not gap markers, are simply copied to the active or neutral string depending upon the mode (# OR ##) of the function. If the characters of form are gap markers, the argument substrings in memory with the corresponding ordinal values are also copied to the active or neutral string, depending upon the mode of the function.

Considerations for Arithmetic Operations

Though the TRAC language is designed for text processing, it does handle limited arithmetic operations on integers. The integers, however, are not represented in pure numeric format but are in character format with the ASCII code. The complete character set of this 7-bit code representation can be found in various references (1, 4, 7). This nonnumeric notation forces the processor to convert a string of ASCII characters into a numeric value so that internal calculations can be performed. Furthermore, the results have to be converted back to character strings for intermediate string values or for function values.

When the processor is tested under the simulator of Fairchild F24 computer, the "Teletype" input of TRAC-text strings in punched on cards, which simulate the actual input in ASCII code from a teletypewriter. The cards

are in the character set of EBCDIC (abbreviation of IBM 360 Extended Binary Coded Decimal Interchange Code), which is different from ASCII.

The simulator takes care of the conversion required for input from EBCDIC to 8-bit (with parity). The processor in turn, converts the seven information bits of the ASCII representation of the character into the internal coded decimal value on which the calculation is performed. For example, the character 5 is read into the IBM/360 memory as 11110101 in EBCDIC. The simulator makes it 0110101 in ASCII. Then the TRAC processor converts it into 101 in binary or 5 in base ten. The addition operation #(AD,5,5,OVER) will result the binary value 1010 or 10 in base ten in the F24 accumulator. For this value to be printed on Teletype, it is converted to a character string consisting of two ASCII characters 0110001 (character 1) and 0110000 (character 0). Therefore, the TRAC processor converts the binary value after the addition 1010 to the above ASCII characters for Teletype output (or for any other purpose, for that matter).

Besides considering conversion, the TRAC processor pays special attention to the problem of the sign of integers. The proper sign may be easily treated when the arithmetic operands are pure numeric integers. The TRAC language accepts integers in ASCII character string format. The sign, if present in text

string, is also in character form. Physically, the sign and integer characters are concatenated. The processor has to interpret the sign as the sign for the integer instead of considering it as a character alone.

Another factor adding to the complexity of signed integer is that the available model 1 of the Fairchild F24 computer can manipulate only positive numbers. The TRAC processor, however, is supposed to be able to handle both positive and negative integers defined by the TRAC language.

There are three different cases for the problem of sign: 1. both signs are positive, 2. either one of the signs is negative, 3. both signs are negative.

For first case the signs, not present in the integer operands, are assumed to be positive for all four integer arithmetic operations.

For the second case, in which one of the two signs is negative as indicated by the character '-' preceding the integer operand, considerations are given to addition, subtraction, and multiplication, division respectively. The reason for two respect treatments is that, when either one of the operand is negative, the sum and difference may be positive or negative while the product and quotient are always negative.

In addition and subtraction, such as #(AD,-3,6,PP)

and #(SU,-3,6,00), the integer operands preceded by the character '-' are transformed into their two's complement internal arithmetic formats before operation. If the sum or difference is positive, it is simply returned as the string value. If the sum or difference is negative, it is transformed into its two's complement form (this makes the value positive) and the character '-' is appended to its string value.

In multiplication and division, such as #(ML,-4,7,A) and #(DV,34,-7,Q,R), the integer operands preceded by '-' are not changed to their two's complement formats. Both operands are treated as positive during the operation. The returned string value, however, is attached by a '-' to indicate it is negative.

For the third case, in which both signs are negative as indicated by a '-' in front of both integer operands, respective considerations are given to addition, subtraction, and multiplication, division.

Since the sum of addition is always negative when both the signs of integer operands are negative, the two operands are treated as positive during operation. That means the two integers are not transformed into their two's complement forms. The returned numeric character string is appended by '-' to denote it is a negative value.

The difference of subtraction could be either positive or negative when both operands are negative. For instance,

the integer string value of #(SU,-7,-10,PPOO) is 3; the integer string value of #(SU,-14,-6,NN) is -8. Before subtraction, both operands are transformed into their two's complement formats. If the numeric value after evaluation is positive, its string value is simply returned. If the numeric value after evaluation is negative, it is first transformed into its two's complement format and then a '-' is attached to reflect the string value is negative.

When both operands are negative, the product and quotient are always positive. During the operation the operands are treated as positive. After evaluation the string value is simply returned as positive value.

CHAPTER BIBLIOGRAPHY

1. Gear, William C., Computer Organization and Programming, McGraw-Hill Book Company, New York, 1969.

2. Harrison, Malcolm C., Data Structures and Programming, Scott, Foresman and Company, Glenvies, Illinois, 1973.

3. Hassit, A., Lageschute, J. W., and Lyon, L. E., "Implementation of a High Level Language Machine," Association for Computing Machinery Communications, 16 (April, 1973), 209.

4. International Business Machines, IBM System 360 Principles of Operation, Form No. A22-6822.

5. Knuth, Donald E., Fundamental Algorithms, Vol. I of The Art of Computer Programming, Reading, Massachusetts, Addison-Wesley Publishing Company, 1973.

6. Mooers, C. N., "TRAC, A Procedure-Describing Language for the Reactive Typewriter," Association for Computing Machinery Communications, 9 (March, 1966), 215-219.

7. Niklaus, Wirth, System Programming, An Introduction, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

This chapter will present a summary of designing experience, offer some conclusions about the results, and make recommendations for possible further development of the current TRAC processor.

Linked Lists

The most important aspect of the designing phase is the choice of data structure for the processor. The organization of data objects in a programming environment is known as data structure. There are numerous data structures: *e.g.*, sequential allocation, stack, queues, trees, and linked lists. The performance of a program or a programming language is affected in an important manner by the data structure upon which it is based.

For the TRAC processor, a sequential list is used to keep track of the next available free memory location; the pushdown stack is reserved for finding matching parentheses within the nested TRAC text string; linked lists are used throughout the text evaluation procedure, which constitutes the body of the processor.

During the earliest stage of designing the processor, sequential allocation was used to manipulate relatively

simple TRAC primitive functions such as RS (read string), RC (read single character), and PS (print string). The allocation of storage was, at that time, taken care of by dividing the available memory into several different sizes of blocks.

However, the sequential allocation seemed awkward when it was applied to those primitive functions involving defined strings such as DS (define string), SS (segment string), CL (call string), CC (call character), DD (delete definition), and other related primitive functions. These functions have to do with, one way or the other, the addition, movement, deletion, and replacement of character strings within the given defined string. The linked lists are tried on these primitive functions and it is found that the above string operations are more easily and smoothly carried out than with the sequential allocation. Therefore, for unity, the linked lists are later chosen to be the basic organization to implement the processor except that the sequential allocation is used to keep track of the next free storage of the one big free area. The original adoption of sequential allocation to handle simple I/O operations is discarded.

The linked lists, furthermore, trigger the idea of "one big free area" for the problem of storage allocation. The original idea of several blocks of storage accompanying the sequential allocation during the earliest phase of

development was also dropped. If the storage were divided into different sizes of available area, there would be a tendency for them to perpetuate themselves during garbage collection (1). By careful management of list pointers within the only one available block, this difficulty has been avoided.

It has been said that, for linked allocation, twelve bits of the Fairchild twenty-four-bit word are used as address pointer and ten bits are used to store the character data. If the linked allocation were applied to a machine with an eight-bit word, it seems not feasible because the word can only accommodate the data item and no space can be used as an address pointer. There is an exception for IBM 360 and 370 systems. Their basic word length is a byte, or eight bits, but the instruction length can be two to six bytes. If the linked allocation were applied to a sixteen-bit word, the data item would be well contained in the word but the accommodation of address pointer would depend upon the total size of addressable memory of the machine.

From the above discussion of the possibility to extend linked allocation to eight- or sixteen-bit word machines, it can be seen that the linking approach takes up additional memory space for the links. This would be the dominating factor while making a choice between linked structure and other data organizations. Fortunately, the

data item used in the processor does not take up the whole twenty-four-bit word and there is already enough space for the address pointer.

The concept of list processing has usually been considered as inaccessible and complicated to ordinary programmers (1). It is something special and thus is for a special group of programming situations. The design of TRAC processor, however, utilizes extensively the linked lists, and has been benefited from this dynamic structure. While this successful example does not, and should not, imply that the structure of linked lists is applicable to every programming environment, it has been shown that the list processing is acceptable and can be understood as long as a judicious manipulation of linked pointers is maintained.

Assembler and Simulator

The TRAC processor has been extensively tested using the available Fairchild F24 assembler and simulator before it is loaded into the F24 computer memory. As a matter of fact, the entire designing process is heavily a programming effort. The correctness of the interpretation of each TRAC command is first examined from the printer output of a simulated execution. If the test result is correct, the actual execution of each TRAC command on the F24 computer system gives the identical result. Nevertheless,

the authentic reactive action offered by the Teletype is simulated by card input and printer output.

If there were no F24 assembler, there would be almost no way to start developing the TRAC processor on Fairchild computer. Pure machine language programming for an on-line application program containing about 2000 statements is absolutely not practical. If there were no F24 simulator, it would make the debugging of the program an unpleasant task. Testing a program of reactive application directly on the machine would usually suffer the difficulty from unexpected program looping and sudden machine lockup. At this time the only available source to find the cause of error is from the switches on the computer. These switches can tell nothing more than a snapshot of the contents of accumulator, the location counter when the program is terminated, and the status of some control switches. Though memory dumps can be requested on some machines, it is more difficult to read than the symbolic listing of the program obtained from the assembler output.

If a TRAC processor is to be developed and implemented on another machine, the assembler may not be necessary since the TRAC processor program could be written in a higher-level language which is to be translated into machine language by the compiler. In other words, the need for an assembler to develop a TRAC processor depends upon the specific machine configuration and the available system software.

As for the simulator, it is suggested that it should be made available to simulate the reactive typewriter or typewriter equipped with Teletype capabilities of the specific machine. The simulator should not only provide a memory dump to show the final contents of each cell used, but furnish the tracing facilities to find out the instruction or statement deviating the program execution.

Response Time

One of the major goals of the thesis is to see the TRAC processor work. The TRAC language is designed for one-line reactive application. The user enters TRAC text string on the Teletype and expects immediate response (output) from the Teletype. The time spent between the user entering the last character of TRAC text string and the Teletype outputting the first character of output string is called response time. In case of null-valued function, the output is indicated by the action of carriage return and line feed. The response time of the F24 TRAC processor on the Teletype keyboard generally takes one fourth to three seconds. The time of this order is required to promise efficient performance on a reactive system.

The response time varies according to the complexity of the TRAC text string entered on the Teletype. For example, the speed of feedback of resulting string value from the simple TRAC statement XY' is, of course, faster than

that of #(DS,A,XYXY)#(SS,A,X)#(CL,A,FEEDBACK)'. The nested structure #(DS,12,345)#(AD,##(CL,12),#(ML,#(CC,12),##(CN,12,2,NOMORE),0),Flow)' takes more time for the Teletype to respond than do simple structures.

Another important factor affecting the response time is the performance of garbage collection. When memory gets full, the frequency of restructuring the available storage increases. This somewhat slows down the response time. However, the Fairchild instructions each generally take 1.6 to 3.2 microseconds (1 microsecond = 10^{-6} second) for execution. This fast execution rate contributes to the not too drastic difference of response time between complicated and uncomplicated TRAC string expressions, between those which require garbage collection and those which do not.

Implementation of Secondary Storage Primitive Functions

All but three of the primitive functions defined in original TRAC language (2) are included in the F24 TRAC processor. These primitive functions are "store block", "fetch block", and "erase block", as described in Chapter III. They allow the keyboard user to move any named strings (forms) into a mass-storage device such as disk, tape, or drum, to retrieve the named strings from the mass storage, and to erase the named strings in the mass storage.

The future version of the F24 TRAC processor should be extended to handle these three primitive functions once the auxiliary storage device is made available for the F24 computer system. One of the advantages of adding these external storage management functions is that they protect the defined strings from accidental erasure. If the defined strings in memory were unexpectedly destroyed, a new copy could be loaded from the external device without having to redefine the strings. This is especially necessary when the defined strings are too long or too complicated to enter them again on the Teletype.

The three primitive functions are to store, fetch, and erase blocks of defined strings. By the time they are implemented, the blocks of defined strings may further be collected together as a higher level group. The group may be put together to form another collection. From there on, a more elaborate file processing ability of the TRAC processor could be initiated.

If these three primitive functions were ever added to the existing TRAC processor, the size of the program for the processor would be necessarily increased. This matter would, in turn, reduce the amount of free storage in the memory. Nevertheless, judging from the number of program instructions used for the existing primitive functions, it could be anticipated that the reducing of free memory

space would not affect the shrinking of the one big block too seriously.

The above anticipation is under the circumstances that the input/output instructions in the program dealing with external storage are relatively simple; that is supposing there is an external storage controller handling the interpretation of external I/O commands instead of letting the processor largely perform the job.

While the adding of external storage manipulation, primitive functions would reduce the available free space; there might be a gain of the available free storage in the other direction. This is under the assumption that all the defined strings are to moved into the secondary storage by the "store block" primitive function when they are created and to be brought into the memory by the "fetch block" primitive function when they are needed. When a block of defined strings is moved from memory to external storage, the address of that block in the external-storage device is stored in memory under the identical block name ("store block" does this). The address of a block sure takes less space than the actual contents of the block. Rather, the space saved would be traded off by the time spent on transferring blocks of defined strings back and forth between the main memory and the secondary devices.

Therefore, it is apparent that, if the three unimplemented primitive functions were added to the F24 TRAC processor, the size of the one big free area of storage in memory would be affected more or less by the following three factors: (1) the increase of program instructions, (2) the control of I/O functions concerning external storage devices, (3) the movement of blocks of defined strings to and from main storage and secondary storage.

Not only the implementation of external storage primitive functions influences the storage allocation, but it also will question the addressability of address pointer used in the linked allocation of the current processor. That is whether the twelve bits of address pointer will be able to point at the address external to the memory. For the future enlargement of the TRAC processor, the effect of the above factors on the storage allocation and the adaptability of linked lists to external devices deserve as much attention as from the file-processing point of view.

CHAPTER BIBLIOGRAPHY

1. Knuth, Donald E., Fundamental Algorithms, Vol. I of The Art of Computer Programming, Reading, Massachusetts, Addison-Wesley Publishing Company, 1973.

2. Mooers, C. N., "TRAC, A Procedure-Describing Language for the Reactive Typewriter," Association for Computing Machinery Communications, 9 (March, 1966), 215-219.

3. Sammet, J. E., Programming Language: History and Fundamentals, New Jersey, Prentice-Hall Inc., 1969.

APPENDIX A

SAMPLE TRAC PROGRAMS

Active, Neutral, and Quote Functions

```
#(DS,A,TRAC)'
#(DS,B(#(CL,A)))'
#(PS,(#(CL,B)))'#(CL,B)
#(PS,##(CL,B))'#(CL,A)
#(PS,#(CL,B))'TRAC
```

Miscellaneous Operations

```
#(DS,Y,THIS IS A TRAC PROGRAM)'
#(SS,Y, )'
#(CL,Y)'THISISATRACPROGRAM
#(CS,Y,PP)'THIS
#(CS,Y,QQ)'IS
#(CR,Y)'
#(CL,Y,&)'THIS&IS&A&TRAC&PROGRAM
```

Strings Concatenation

```
#(DS,A,THIS IS THE STRING)'
#(DS,B,TEXT PROCESSING)'
#(CL,A)#(CL,B)'THIS IS THE STRINGTEXT PROCESSING
#(CN,A,2,YY)#(CN,B,5,QQ)'THTEXT
#(DS,SHARP,(#))'
#(CL,SHARP)(DS,XX,XX)'
#(PS,##(CL,XX))'XX
```

Square of A Number

```
#(DS,SQUARE,(#(ML,*,*,OVER)))'
#(SS,SQUARE,*)'
#(SQUARE,12)'144
```

Factorial of A Number

```
#(DS,FACT,(#(EQ,1,N,1,(#(ML,N,#(CL,FACT,#(AD,N,-1,0)),0)))))'
#(SS,FACT,N)'
#(CL,FACT,6)'720
```

APPENDIX B

GLOSSARY OF TERMS

Active Function	those primitive functions (see Primitive Function) whose values returned from evaluation are to be rescanned and reevaluated.
Active String	the string is composed of the TRAC programs or substring currently scanned.
Form	a string which has been given a value and associated with a name.
Form Store	memory space occupied by the forms.
Form Position Pointer	the pointer used to point at the specific character of the form.
Idling String	#(PS,#(RS)); it is initially loaded in the active string, causes a string terminated by the end-of-string symbol "'" to be read from the Teletype into the active string, then the string is evaluated, printed and a fresh copy of the string is loaded.
Neutral Function	those primitive functions (see Primitive Function) whose values returned from evaluation are not to be scanned or evaluated.
Neutral String	a work area where string of characters is used for execution.
One Big Free Area	an area of TRAC memory consisting of 1000 locations shared by active string, neutral string, and intermediate value resulting from function evaluation.
Primitive Function	TRAC instructions specify the action to be taken upon the string.

BIBLIOGRAPHY

Books

Gear, William C., Computer Organization and Programming, New York, McGraw-Hill Book Company, 1969.

Harrison, Malcolm C., Data Structures and Programming, Glenvies, Illinois, Scott, Foresman and Company, 1973.

International Business Machines, IBM System 360 Principles of Operation, Form No. A22-6822.

Knuth, Donald E., Fundamental Algorithms, Vol. I of The Art of Computer Programming, Reading, Massachusetts, Addison-Wesley Publishing Company, 1973.

Niklaus, Wirth, System Programming, An Introduction, Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1973

Sammet, J. E., Programming Language: History and Fundamentals, Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1969.

Scott, Dan W., CASH Language Primer, University Computing Company, Dallas, 1969.

Wegner, P., Programming Languages, Information Structure and Machine Organization, New York, McGraw-Hill Book Company, 1968.

Articles

Hassit, A., Lageschute, J. W., and Lyon, L. E., "Implementation of a High Level Language Machine," Association for Computing Machinery Communications, 16 (April, 1973), 209.

McIlroy, M. D., "Macro Instruction Extensions of Compiler Language," Association for Computing Machinery Communications, 3 (April, 1960), 214-220.

Mooers, C. N., "TRAC, A Procedure-Describing Language for the Reactive Typewriter," Association for Computing Machinery Communications, 9 (March, 1966), 215-219.

Reports

Bosack, L., Eichenberger, P., Klein B., Levine, J., Theriault, D., and Young, J., "TRAC Language: Construction, Use and Philosophy," presented at the DECUS Symposium, Wakefield, Massachusetts, May 13, 1969.

Mooers, C. N., and Deutsch, L. P., "TRAC, A Text Handling Language," Proceedings, Association for Computing Machinery Twentieth National Conference, (August, 1965), 229-246.

Unpublished Materials

Eastwood, D. E., and McIlroy, M. D., "Macro Compiler Modification of SAP," unpublished memorandum, Bell Telephone Laboratories, Computation Laboratory, 1960.

Grimes, Glen T., "Design and Implementation of an Assembler for the Fairchild F24 Computer," unpublished master's problem in lieu of thesis, Department of Computer Sciences, North Texas State University, Denton, Texas, 1973.

Scott, Dan W., unpublished programs of F24 simulator on the IBM 360, F24 program loader, and F24 memory dump, Department of Computer Sciences, North Texas State University, Denton, Texas, 1973

Wickham, K., and Hamming C., "The TRAC Processor," unpublished paper for a course given by Dan W. Scott at Southern Methodist University, Dallas, December, 1969.