ED 082 489 EM 011 458

| | |
|---|---|
| AUTHOR | Mir, Carol Loeb |
| TITLE | A Comparison of String Handling in Four Programming Languages. Technical Progress Report. |
| INSTITUTION | North Carolina Univ., Chapel Hill. Dept. of Computer Science. |
| SPONS AGENCY | National Science Foundation, Washington, D.C. |
| REPORT NO | UNC-TPR-CAI-5 |
| PUB DATE | Sep 72 |
| NOTE | 108p.; Thesis submitted to the Department of Computer Science, University of North Carolina |
| EDRS PRICE | MF-$0.65 HC-$6.58 |
| DESCRIPTORS | *Comparative Analysis; Masters Theses; *Programing; *Programing Languages; Technical Reports |
| IDENTIFIERS | APL; *Character String Handling; PL I; SNOBOL 4; String Processing Languages; TRAC |

ABSTRACT

Character string handling in the programing languages SNOBOL 4, TRAC, APL, and PL/I are compared. The first two of these are representatives of string processing languages, while the latter two represent general purpose programing languages. A description of each language is given and examples of string handling problems coded in the four languages are provided. Finally, the languages are compared on the basis of their string handling abilities rather than on the basis of implementation-dependent characteristics. (Author)

# University of North Carolina at Chapel Hill

Department of Computer Science

# A COMPARISON OF STRING HANDLING IN FOUR PROGRAMMING LANGUAGES

Carol Loeb Mir

September 1972

Technical Progress Report CAI-5
to the
National Science Foundation

under Grant GJ-755

DEPARTMENT of COMPUTER SCIENCE

University of North Carolina at Chapel Hill

# A COMPARISON OF STRING HANDLING IN FOUR PROGRAMMING LANGUAGES

by

Carol Loeb Mir

A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1972

Approved by:

Peter Calingaert

Adviser

Stephen F. Weiss

Reader

Martin Dillon

Reader

CAROL LOEB MIR. A Comparison of String Handling in Four Programming Languages. (Under the direction of PETER CALINGAERT.)

The thesis compares character string handling in the programming languages SNOBOL4, TRAC, APL, and PL/I. The first two languages are representatives of string processing languages, while the latter two represent general purpose programming languages. A description of each language is given. Also included are examples of string handling problems coded in the four languages. The languages are compared on the basis of their string handling abilities and not on the basis of implementation-dependent characteristics.

## ACKNOWLEDGEMENTS

I would like to thank Dr. Peter Calingaert for suggesting the thesis and guiding me through its execution. He painstakingly corrected many rough drafts. I am grateful to my husband Vern for his suggestions and contributions to the thesis.

## TABLE OF CONTENTS

# 1 INTRODUCTION

The purpose of this thesis is to compare character string handling in different programming languages. Of particular concern are string operations in text handling. Sammet mentions [19, p. 385]

> The text material can be either natural language of some kind (e.g., this sentence), a string composed of a program in any language, or any arbitrary sequence of characters from some particular data area.

This thesis considers only natural language text material. Of course, this could be generalized to other special uses of string handling.

String processing and list processing languages are examples of symbol manipulation languages. The data which they manipulate are symbols, not numbers. Symbol manipulation languages are used in such areas as compiler writing, theorem proving, formula manipulation, and text processing.

Many accounts treat strings and lists together, but it is important to differentiate between them. A string is a sequence of characters; it is a data type in many programming languages. A list, on the other hand, is a structure of data, which may or may not be characters. Sammet [19,

p. 385] distinguishes between a string and a list by noting that the list is a way of storing information rather than a *type* of information to be represented.

String handling operations include concatenation of two strings, searching for a pattern, and replacing one pattern with another. Examples of list processing operations are putting information into a list, deleting information from a list, and combining two lists.

Since only string operations are of concern in this thesis, the following symbol manipulation languages are excluded from consideration: [see reference 17]

> list processors, such as LISP1.5 and IPL-V;
> linked block languages, such as $L^6$;
> pattern-directed structure processors, like CONVERT and FLIP.

The last group of languages perform string-like operations, but they operate on LISP list structures, not character strings.

Text editors like TEXT360 are useful for publishing documents. These editors include comrands for line and document updating, which are string handling tasks. For example, inserting a phrase in the middle of a sentence is essentially a pattern matching task. However, their commands do not give an insight into how string problems are dealt with, so text editors are not included in the thesis.

The thesis compares string handling in two kinds of languages. These are string processing languages and general purpose programming languages with built-in string handling capabilities. String processing languages can be classified as pattern-directed string processors and macro-expander string processors.

Included in pattern-directed string processors are all versions of the PANON, COMIT, and SNOBOL languages. These languages use the generalized Markov algorithm as a way of defining string processing operations. The Markov algorithm consists of a series of transformation rules. The languages perform substitutions on a string depending on the structure of the string according to the transformation rules. (For more information on the subject see [5].)

These languages, in particular PANON, may be used effectively to write the syntax analysis phase of compilers. In such cases a program is regarded as a long string to be analyzed. PANON is not considered in the thesis since it is more like a syntax-driven compiler than a string processor [3]. SNOBOL4, which includes many of COMIT's features, is discussed in detail. A main factor for using SNOBOL4 for comparison was the availability of an implementation. Also, COMIT lacks some desirable language features, such as the ability to name strings, and facilities for easy arithimetic operations.

Two languages which are in the category of macro-expander string processors are GPM and TRAC. To perform any operation in these languages (input/output, arithmetic, assignment, etc.), a macro must be called with the necessary parameters. Since the TRAC language is so different from other programming languages and does include several string handling functions, it has been included.

PL/I, unlike most other general purpose programming languages, provides good string handling capabilities and is included in the discussion. APL, also considered, is an example of a general purpose programming language that provides for character data but does not have good string handling functions.

The four languages included in the thesis, then, are SNOBOL4, TRAC, PL/I, and APL. A brief summary of each language is in Chapter 2.

In Chapter 3 two easy string problems are coded in each language. Also included in the chapter is a rather difficult string handling problem coded in SNOBOL4, PL/I, and APL.

Chapter 4 includes comparisons of the languages on the bases of what string operations are primitive in each language, and of ways string operations that are not primitive in a language might be coded in that language. The possible string handling problems for which the languages are suited or not suited are discussed.

All comparisons of the languages in the thesis are made on the basis of language features. Implementation-dependent considerations, such as compilation time, execution speed, and amount of storage used, have not been considered. A good comparison based on these latter criteria would have been extremely difficult for the following reasons. PL/I, TRAC, and SNOBOL4 programs were batch processed, but APL programs used an interactive time sharing system. TRAC, SNOBOL4, and APL were executed interpretively, but PL/I was compiled into an object deck for later execution. Thus, these differences would tend to hide results that might be evident from a comparison of more similar implementations.

The languages are examined on the basis of the string operations which are primitive in them, not string operations that can be added with a subroutine capability. A good programmer can code any string handling operation that he needs, but this should not figure in a language comparison, unless the language had no facilities for defining new string functions.

SNOBOL4 programs were run interpretively on an IBM 370/165 in batch mode. TRAC programs were run interpretively on an IBM 360/75 in batch mode. PL/I programs were run on an IBM 360/75 using the IBM PL/I F compiler. APL programs were run interpretively on an IBM 370/165 in a time sharing environment.

# 2 LANGUAGE DESCRIPTIONS

In this chapter a brief summary of each language is given. The language features discussed include data types, statement types, and functions.

## 2.1 SNOBOL4

SNOBOL is a string processing language which originated at Bell Laboratories in 1964; SNOBOL4 is the latest refinement. Its authors are D.J. Farber, R.E. Griswold, and I.P. Polonsky. Many of SNOBOL4's features, including its basic statement format, are influenced by COMIT [13], an earlier string handling language. References for the SNOBOL language are [8], [9], and [10].

### 2.1.1 Data Types

There are several different data types, the most important one being the string. Strings can be broken up into components, operated upon, and then put together again. Unlike what is done in COMIT, an earlier string manipulating language, strings may be assigned names. It is also possible to assign names to matched and partially matched substrings by the respective operations of conditional and immediate value assignment. An example of a string in its literal form is 'I AM A STRING'. One may write

```
     X = 'I AM A STRING'
```

where X is a variable that is assigned the string value 'I AM A STRING'. X is considered to be of type string.

A string must often be searched for a pattern. In SNOBOL4 a pattern is a structure that can be a string, a number of strings joined by the concatenation operator (a blank), a number of strings separated by the alternation operator (a | with at least one blank on each side of it), or possibly a combination of all three. The alternation operator allows matching of alternate patterns. Patterns may be combinations of both literal strings and variables whose values are strings or patterns. Examples are the pattern

```
     'EIT' 'HER' | 'OR'
```

(whose first alternate is equivalent to 'EITHER'), and the pattern

```
     'B' VAR1 | 'B' | VAR2
```

(whose first alternate is a literal concatenated with a variable). The statement

```
     IT = 'ONE' | 'TWO'
```

assigns to IT a pattern that matches either the string 'ONE' or the string 'TWO'. If Y = 'ONE', then the pattern Y | 'TWO' is an equivalent pattern to the previous value of variable IT .

There are also the arithmetic data types INTEGER and REAL, type ARRAY, and programmer-defined data types. Declarations of the data types of variables are not present in SNOBOL4. Instead, the type of a variable is dependent on the variable's last assigned value.

### 2.1.2 Statements

There are four different statement types: assignment, pattern matching (without replacement), replacement, and END. Actually all four statements follow a basic statement format consisting of five different fields, some of which may be absent in a particular statement. This format is:

<u>label</u> <u>subject</u> <u>pattern</u> = <u>object</u> <u>go-to</u>

Fields must be separated by at least one blank. If the label field is present, it must begin in Column 1. A statement not having a label must start in other than Column 1. There are no other specifications for the beginning of any of the other statement components. However, no characters may appear after Column 71. Continuation cards may be used, so fields may be as long as desired. No maximum length of any field is specified. Labels must begin with a letter or digit and extend to the first blank. The subject or object may be either a literal string or the name of a string. The pattern field may be any of the possibilities described previously for a pattern. The go-to field is used to indicate conditional and unconditional branching. In the statement

```
START  X = 'ABC'   :(NEXT)
```

the go-to field causes the statement whose label is NEXT to be branched to after X is assigned 'ABC'. Branching conditionally upon success or failure of a statement is done with a :S(label) or :F(label), respectively, in the go-to field. (Success or failure of a statement will be explained shortly.)

The assignment statement has already been illustrated in previous examples. Its format is

*label* *subject* = *object* *go-to*

*label* and *go-to* are optional. The value of the object is assigned to the subject.

The pattern matching and replacement statements are a little more involved. The pattern matching statement's format is:

*label* *subject* *pattern* *go-to*

*label* and *go-to* are optional. The entire subject is searched for an occurrence of the first alternate of the pattern; if it is not found, then the subject is searched for the second alternate, etc. The statement is said to succeed if the pattern is located in the subject; it fails otherwise. For example, consider

```
         STR = 'CABABET'
FIRST    STR 'AD' | 'AB'
```

Statement FIRST succeeds, matching pattern 'AB' with the first AB in the subject. A pattern matching statement with 'AD' in place of 'AD' | 'AB' in the pattern field would fail.

The result of a replacement statement is to substitute an object for the first occurrence of the matched pattern alternate in the subject. The basic format of a replacement statement is

*label* *subject* *pattern* = *object* *go-to*

*label* and *go-to* are optional. To replace the first B with an R in statement FIRST, one would write:

```
         STR = 'CABABET'
FIRST    STR  'B'  =  'R'
```

STR now has the value 'CARABET'. Suppose that it was desired to replace the second B rather than the first B with an R. Then it would be necessary to write:

```
         STR = 'CABABET'
FIRST    STR 'BE' = 'RE'
```

An END statement is simply END in the label field and signifies the end of a SNOBOL4 program.

The four kinds of statements and input/output are illustrated in the following short program whose purpose is to count the number of E's and I's in some input cards.

```
START     X = INPUT                       :F(END)
          SUM = 0
LOOP      X 'I' | 'E' =                   :F(OUT)
          SUM = SUM + 1                   :(LOOP)
OUT       OUTPUT = SUM                    :(START)
END
```

Input cards:

```
     HE RECEIVED A GIFT.
     A BEE STUNG THE BOY.
     OUR PROGRAMS HAD FAULTS.
```

Output lines:

```
     6
     3
     0
```

Execution of the statement labelled START causes one input card to be read and assigns X the value of the card. The go-to field :F(END) means that on failure (there are no more input cards) the program is finished. Otherwise the normal sequential order of the program is followed, i.e. go to the second statement. The second statement initializes SUM to 0. In the third statement X is searched for the first occurrence of the letter 'I'. If no I's are found, then 'E' is to be looked for. The lack of an object after the = sign means that the 'I' or 'E' is to be deleted from X. If an I or E is found, one is added to SUM, and X is searched again. When an I or E can no longer be found, the program branches to the statement labelled OUT, which causes the printing of a line with the value of SUM. The program then branches to START. The process continues until no more cards are in the input file, whereupon the program terminates. Notice that after all the I's and E's are found, X is the value of the input card with all I's and E's removed. For example, the final value of X is 'H RCVD A GFT' for the first input card.

The following program segment finds the first E or I in X; if either letter is found, it indicates which of the two it was. This is easily done using conditional value operation.

```
     X = 'RELIVE'
     X ('I' | 'E') . FIND =              :F(OUT)
     OUTPUT = FIND
```

The conditional value operator is a period (.), separated on both sides by at least one blank. In the above example, conditional value assignment associates a variable, FIND, with a pattern ('I' | 'E'), such that when pattern alternate 'I' matches the I in 'RELIVE', FIND is assigned 'I'.

### 2.1.3 Arithmetic

Arithmetic facilities are limited in SNOBOL4. Addition, subtraction, multiplication, division, and exponentiation of integer and real numbers may be done. Version 3 of SNOBOL4 permits mixed-mode expressions and real exponents.

### 2.1.4 Functions

There are several built-in or primitive functions in SNOBOL4. For example, SIZE(X) returns the number of characters in string X and TRIM(X) removes all trailing blanks in string X. REPLACE(X,Y1,Y2) replaces all occurrences of Y1 in string X by Y2. Several primitive functions are useful for pattern matching. LEN(X), where X is an integer, has a value of a pattern that matches any string of length X. The statement

```
     'RELIVE'  LEN(1) . A
```

results in A being assigned the value 'R'. SPAN(X) and BREAK(X), where X is a string of characters, will match runs of the characters of X in the subject. TAB(integer) and RTAB(integer) allow matching attempts to be started at a desired position in the subject. ARB (no argument) matches an arbitrary number of characters in the subject. For instance, in the pattern matching statement

```
     'THE PICTURE ON THE WALL'  'PICTURE'  ARB  'WALL'
```

ARB matches ' ON THE '. There is also a cursor position operator @ to assign the position in the subject where a match occurred. After execution of the following statement PTR will be assigned the value 4, the position just before 'PICTURE'.

```
     'THE PICTURE ON THE WALL'  @PTR  'PICTURE'  ARB  'WALL'
```

A second type of function in SNOBOL4 is the predicate. If the condition specified by the predicate is satisfied, the predicate is replaced by the null string. If the condition is not satisfied, the statement fails and no operation is performed. The statement

```
     I = LE(I,9) I + 1                    :F(END)
```

will succeed, adding 1 to I, so long as I is less than or equal to 9. Tha numeric predicates include LT, LE, GT, GE, EQ, and NE, whose meanings are what one would expect. INTEGER(X) determines whether X is an integer. Other predicates compare two strings instead of two numbers. For example, LGT(X,Y) succeeds if string X follows string Y in lexical ordering.

The third type of function is a function defined by the user. These functions may be redefined during program execution. No special notation is required for recursive function calls.

### 2.1.5 Other Features

Other features of the language include data type conversion, indirect referencing, delayed evaluation of expressions in patterns, and the possibility of changing the way the subject is scanned for a pattern.

SNOBOL4 programs are translated into Polish prefix object code, and then executed interpretively. This helps explain the good trace facilities in the language.

Some of the differences betwecn SNOBOL4 and the earlier SNOBOL and SNOBOL3 include improvements to I/O and arithmetic capabilities. Also, the array data type was not present in SNOBOL. There was no alternation operator in the earlier languages, so patterns had to be less intricate. A large number of the primitive functions which help in doing complicated pattern matching problems were not present in the earlier languages.

## 2.2 TRAC

TRAC is an entirely different kind of string handling language from SNOBOL4. It is a macrogenerator language designed to be interactive. Wegner [21] says that a macro definition may be viewed as a function definition f such that for every set of actual parameters a[1],...,a[n] in the allowed domain, a value string f(a[1],...,a[n]) is determined which consists of the string generated as a result of macro expansion. In macro assemblers the domain of actual parameters consists of any strings that result in well-formed lines of code, where the lines of code are the range of the function. However in TRAC the domain and range of arguments are to some extent arbitrary strings.

The two people responsible for the development of TRAC are Calvin Mooers and Peter Deutsch. The TRAC system was designed for interactive text processing. Sources of the TRAC concepts came from COMIT, LISP, and McIlroy's macro assembly system [5]. TRAC was developed independently from Strachey's GPM [21], although the languages are very similar. TRAC is discussed in [5], [14], [15], [16], [20], and

[21].

### 2.2.1 TRAC Instructions

The basic instruction format is:

:(FCN,p[1],p[2],...,p[k])'

:( indicates a call to FCN, where FCN is a two-letter TRAC primitive (or evaluates to a two-letter TRAC primitive). The arguments of FCN are p[1],p[2],...,p[k]; each p[i] is a string of characters. An activation symbol, usually the apostrophe, indicates the end of input and causes the processor to execute what was just entered. FCN is also referred to as a macro name.

Instructions are executed interpretively by consulting a table in memory for the name of the primitive and then transferring to a subroutine for executing the primitive. A new primitive is added to the language by adding it to the table. However no new primitive can be specified within a TRAC program. It must be entered before execution.

An instruction is executed by replacing the instruction with its value, which may be the null string. Instructions may cause side effects in the memory, I/O medium, or information which determines the mode of operation of the TRAC processor.

### 2.2.2 TRAC Primitives

TRAC primitives include, first of all, primitives which allow the language to be interactive.

:(RS)' indicates a string of characters is to be read from the typewriter until an end of string character is found, and that this instruction is to be replaced with what was just read.

:(PS,string)' prints the value of string. For example,

:(PS,IT IS RAINING)'

prints IT IS RAINING. After printing, the null string is left as the value of the instruction.

Macro definition is accomplished with the define string primitive. :(DS,name,string) says to evaluate name, evaluate string, and define the value of string to have as its name the value of name. For instance,

:(DS,A,:(RS))'

causes a string to be read, evaluated, and the result named A.

Macros are called with the call primitive. :(CL,name) says to call the name to which the name expression evaluates and replace the instruction with the name's value. Thus, the new string could be a new instruction.

Parameters may be introduced in a defined string with the segment string primitive. :(SS,name,p[1],p[2],...,p[k]) says to evaluate name, evaluate the parameters p[i], and call the named string and replace each instance in it of p[ i ] by a paramteter marker for i. The string is stored back in memory. For example, consider

:(SS,A,RAIN)'

If A has the value IT IS RAINING, then RAIN is replaced by a parameter marker. To see this new form for A, the print form primitive may be used. The value of :(PF,A)' would be IT IS <1>ING. Parameters may be replaced with actual parameters. :(CL,name,a[1],a[2],...,a[m]) replaces all occurrences of parameter markers with the corresponding actual parameters a[1],a[2],...,a[m]. If the number of actual parameters is less than the number of parameter markers, i.e. m<k, then null strings replace the remaining parameter markers. If m>k, then p[k+1],...,p[m] are ignored. The instruction

:(PS,:(CL,A,SNOW))'

prints the value of :(CL,A,SNOW) which is IT IS SNOWING.

### 2.2.3 Evaluation Modes

TRAC has three different evaluation modes: active, neutral, and quote.

The characters :( initiate the active mode. These symbols cause the interpreter to delay evaluation of the current function (if there is one) and evaluate all arguments following :( until the matching right parenthesis is found. For instance, in evaluating :(PS,:(CL,A,SNOW))', execution of the print string function is delayed until :(CL,A,SNOW) is executed. The string produced as a result of evaluating the active function is evaluated again, unless it is the null string.

The characters ::( initiate the neutral mode. The difference between this and the active mode is that after the characters between ::( and matching ) are evaluated once and a resulting string produced, the resulting string is not rescanned.

The quote mode, initiated by (, stops all evaluation of what is between the matching parentheses. Examples 1.,2., and 3. below show the differences among the three modes.

Assume these definitions are made for X and Y:

```
:(DS,X,BOOK)                         X has value BOOK
:(DS,Y,(:(CL,X)))                    Y has value :(CL,X)
```

Then

```
1. :(PS,:(CL,Y)) prints BOOK
2. :(PS,::(CL,Y)) prints :(CL,X)
3. :(PS,(:(CL,Y))) prints :(CL,Y)
```

Two stacks are necessary during evaluation, the active string stack and the neutral string stack. Every instruction is copied to the top of the active string stack and then scanned. Since parameters may also call TRAC functions, a stack is needed in which to put intermediate results of parameter evaluation. Thus, the necessity arises for the neutral string stack. A flowchart of TRAC evaluation (Figure 2.1) follows [15].

### 2.2.4 Arithmetic Primitives

TRAC has primitives to handle the usual arithmetic operations. For example, :(AD,d1,d2)' returns the sum of d1 and d2, which are strings representing numbers.

### 2.2.5 Decision Primitives

Two primitives EQ (equals) and GR (greater) provide decision facilities. The value of :(EQ,x1,x2,t,f) is t if character string x1 is equal to character string x2, otherwise the value is f. Similarly, :(GR,d1,d2,t,f) is t if d1 is greater than d2. GR's operands d1 and d2 must be strings representing numbers, not character strings.

### 2.2.6 Character Primitives

Each defined string (or form) has a form pointer associated with it. Initially the form pointer points to the first character of the string; it may be moved by four primitives: CC (call a character), CN (call a number of characters), CS (call a segment), and IN (index). The value of the instruction

:(CC,S,z)'

is the character in S pointed to by S's form pointer. As a side effect, the form pointer of S is moved ahead one

*ACTIVE STRING HAS A SCAN POINTER*
*CURRENT LOCATION IS IN NEUTRAL STRING*
*IDLING PROCEDURE IS :(DS,:(RS))*

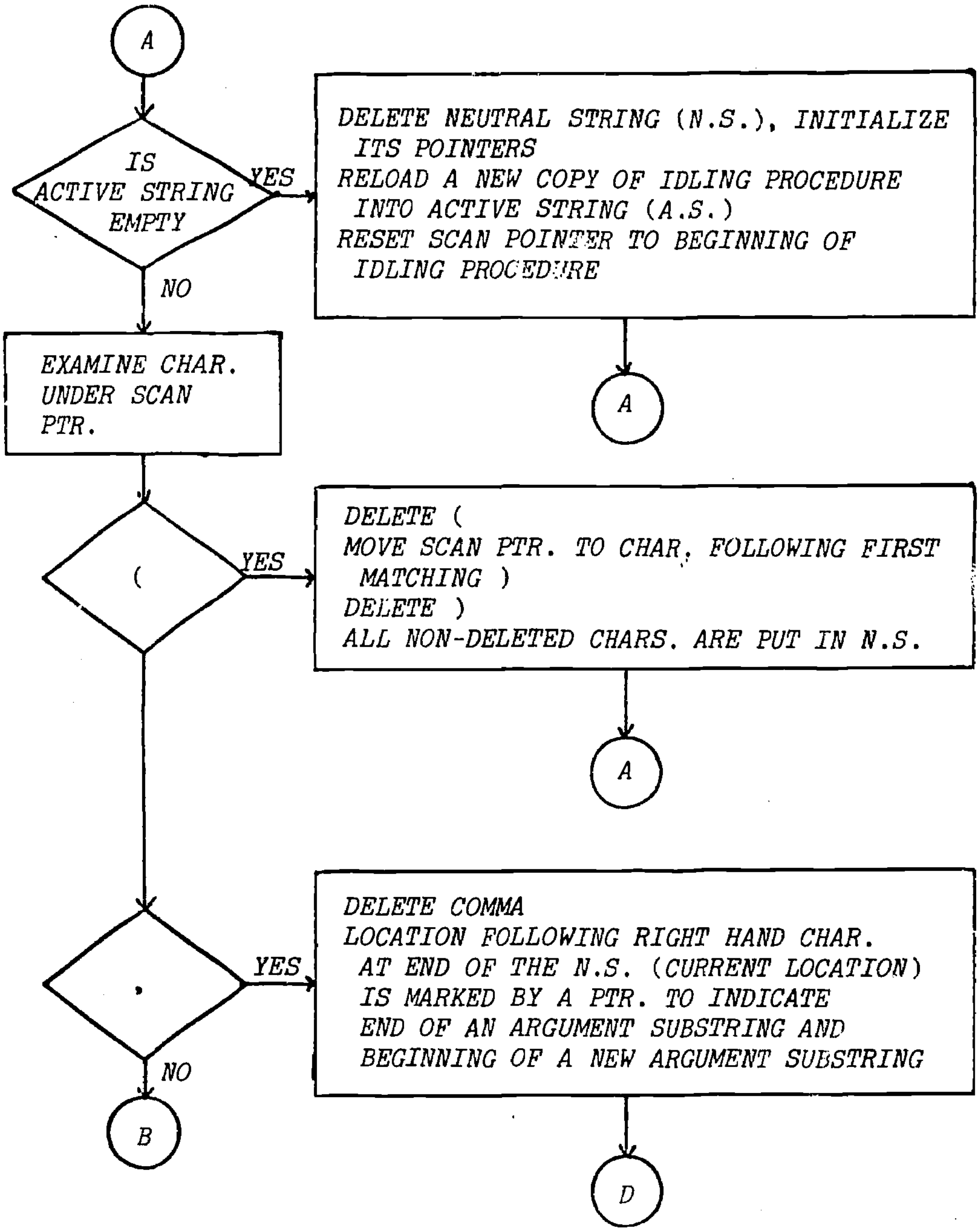

A
IS ACTIVE STRING EMPTY
YES
NO
DELETE NEUTRAL STRING (N.S.), INITIALIZE ITS POINTERS
RELOAD A NEW COPY OF IDLING PROCEDURE INTO ACTIVE STRING (A.S.)
RESET SCAN POINTER TO BEGINNING OF IDLING PROCEDURE
A
EXAMINE CHAR. UNDER SCAN PTR.
(
YES
DELETE (
MOVE SCAN PTR. TO CHAR. FOLLOWING FIRST MATCHING )
DELETE )
ALL NON-DELETED CHARS. ARE PUT IN N.S.
A
,
YES
NO
DELETE COMMA
LOCATION FOLLOWING RIGHT HAND CHAR. AT END OF THE N.S. (CURRENT LOCATION) IS MARKED BY A PTR. TO INDICATE END OF AN ARGUMENT SUBSTRING AND BEGINNING OF A NEW ARGUMENT SUBSTRING
B
D

Figure 2.1 TRAC Algorithm.

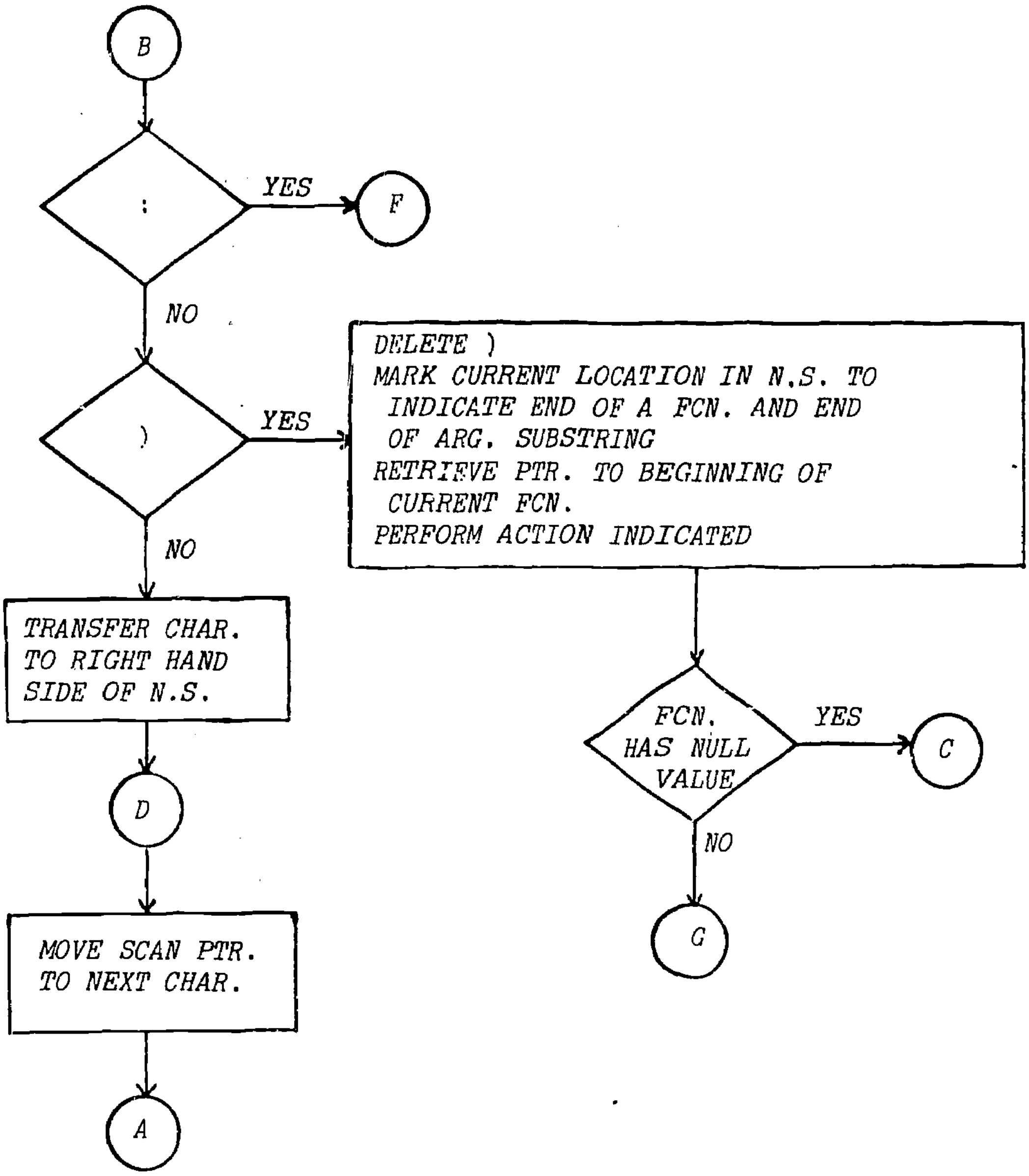

B
:
YES
F
NO
)
YES
DELETE )
MARK CURRENT LOCATION IN N.S. TO INDICATE END OF A FCN. AND END OF ARG. SUBSTRING
RETRIEVE PTR. TO BEGINNING OF CURRENT FCN.
PERFORM ACTION INDICATED
NO
TRANSFER CHAR. TO RIGHT HAND SIDE OF N.S.
D
MOVE SCAN PTR. TO NEXT CHAR.
A
FCN. HAS NULL VALUE
YES
C
NO
G

Figure 2.1 (cont.)

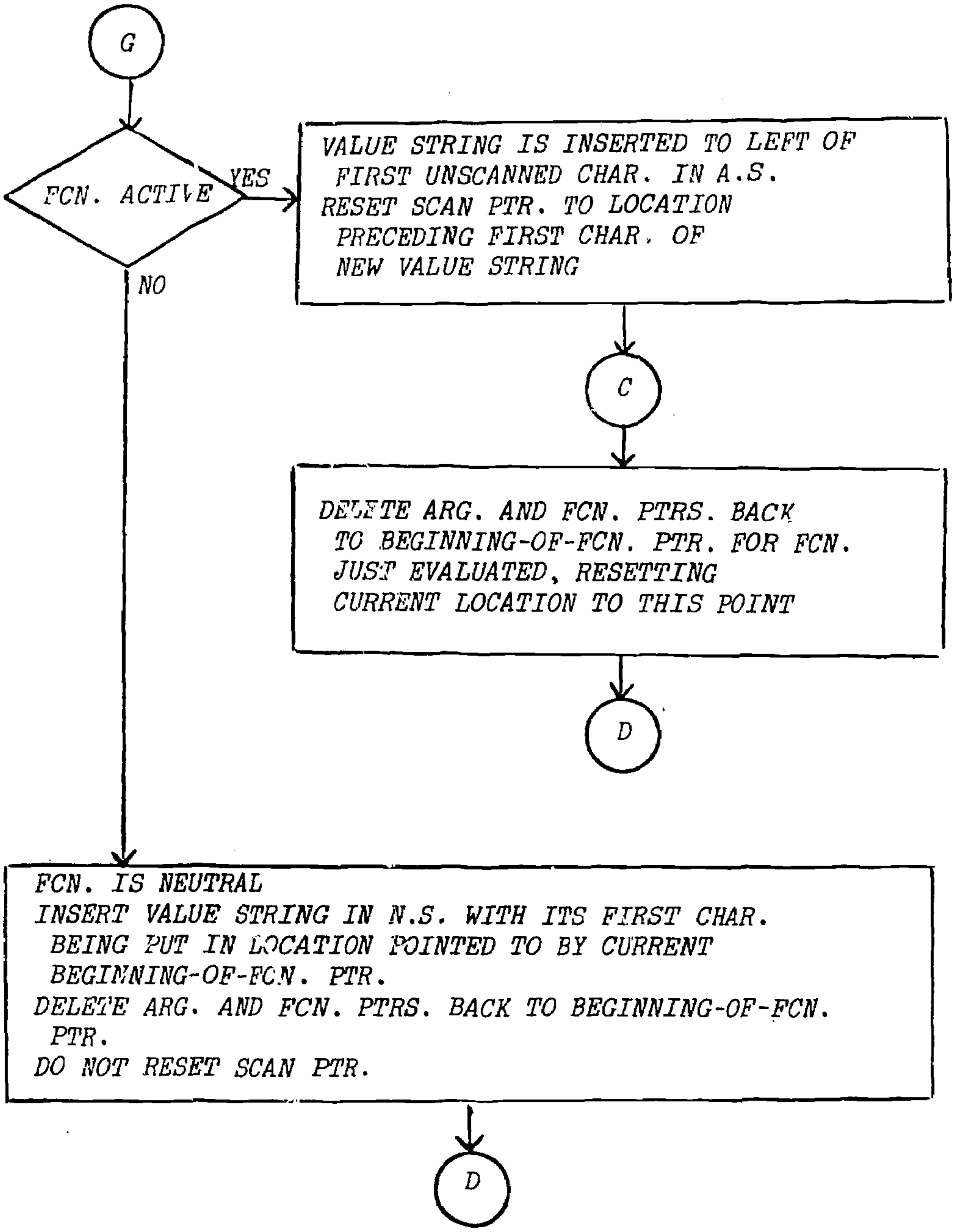

G
FCN. ACTIVE
YES
NO
VALUE STRING IS INSERTED TO LEFT OF FIRST UNSCANNED CHAR. IN A.S.
RESET SCAN PTR. TO LOCATION PRECEDING FIRST CHAR. OF NEW VALUE STRING
C
DELETE ARG. AND FCN. PTRS. BACK TO BEGINNING-OF-FCN. PTR. FOR FCN. JUST EVALUATED, RESETTING CURRENT LOCATION TO THIS POINT
D
FCN. IS NEUTRAL
INSERT VALUE STRING IN N.S. WITH ITS FIRST CHAR. BEING PUT IN LOCATION POINTED TO BY CURRENT BEGINNING-OF-FCN. PTR.
DELETE ARG. AND FCN. PTRS. BACK TO BEGINNING-OF-FCN. PTR.
DO NOT RESET SCAN PTR.
D

Figure 2.1 (cont.)

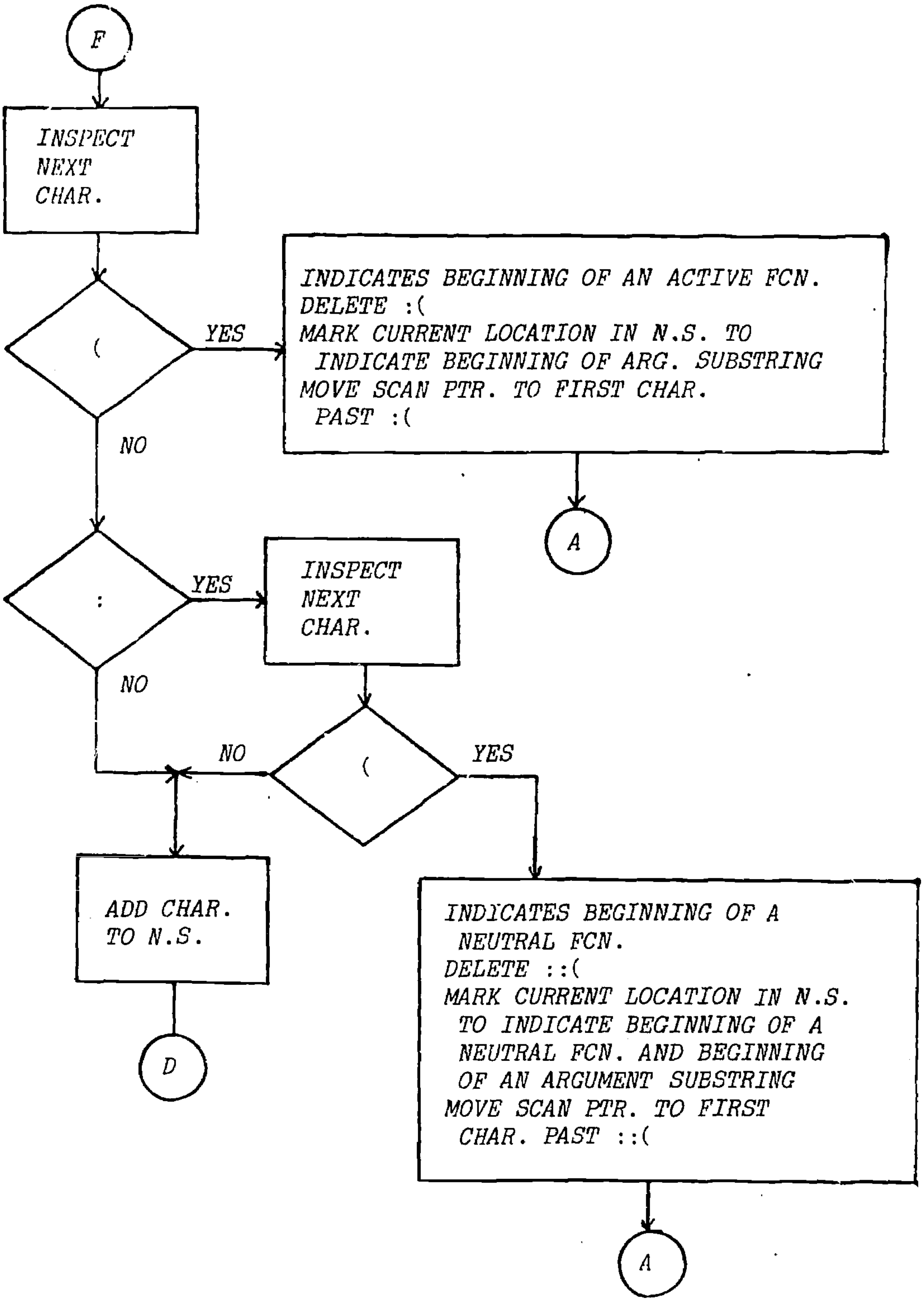

F
INSPECT NEXT CHAR.
(
YES
INDICATES BEGINNING OF AN ACTIVE FCN.
DELETE :(
MARK CURRENT LOCATION IN N.S. TO INDICATE BEGINNING OF ARG. SUBSTRING
MOVE SCAN PTR. TO FIRST CHAR. PAST :(
A
NO
:
YES
INSPECT NEXT CHAR.
NO
NO
(
YES
ADD CHAR. TO N.S.
D
INDICATES BEGINNING OF A NEUTRAL FCN.
DELETE ::(
MARK CURRENT LOCATION IN N.S. TO INDICATE BEGINNING OF A NEUTRAL FCN. AND BEGINNING OF AN ARGUMENT SUBSTRING
MOVE SCAN PTR. TO FIRST CHAR. PAST ::(
A

Figure 2.1 (cont.)

character. If, before :(CC,S,z)' is executed, the form pointer of S points beyond the last character of S, the value of the instruction is z. Similarly, if the form pointer is beyond the last character of S, the value of :(CN,S,k,z)' is z. Otherwise the value returned is the next k characters of S after the form pointer. The form pointer is moved ahead (or back if k is negative) k places. :(CS,S,z)' gives the segment of characters from the current position of the form pointer to the next parameter marker. :(IN,S,x,z)' searches S for substring x. If the substring is present, the value that is returned is the string between the beginning position of the form pointer and the matched string; the form pointer is moved to the character after the matched string. If there is no match, z is returned. The cursor-reset or call-restore function :(CR,S) resets the form pointer of S to the first character in S.

Some other functions useful in string processing are mentioned by van der Poel in [20]. One is the yes there function, :(YT,N,x,t,f). If string x is in N, then the value of the function is t, otherwise the value is f. :(LP,N) and :(RP,N) give, respectively, the number of characters to the left of the form pointer and to the right of the form pointer. Another function, IL (in left), is like IN but searches to the left in x. :(LG,x1,x2,t,f) determines whether string x1 is lexically greater than string x2. If so, the value returned is t; if not, the value returned is f.

A character primitive combined with EQ can move the form pointer ahead and return a null string as result. For example,

```
:(EQ,:(CS,SENT),)'
```

moves the pointer after a segment of SENT. There are no true-false exits, and a null string is returned.

## 2.3 APL

APL was originated by Kenneth Iverson. It was developed further in association with A.D. Falkoff. Discussions of APL may be found in [11] and [12].

APL is a general purpose programming language whose concise notation is good for interactive use. APL is particularly useful in dealing with vectors and multidimensional arrays. The APL discussed in the thesis is the implementation used in an APL/360 interactive system. The implementation provides a good repertoire of system action commands; these will not be discussed.

The double arrow (<-->) will be used in the following discussion to denote equivalence. This symbol is not part of APL but merely a notational convenience.

### 2.3.1 Data Types

As in SNOBOL4, there are no declarations of type of a variable. The only types are numbers and characters.

A scalar may be a number or a character. An array is built from scalars of the same type. Thus, an array cannot contain both numbers and characters.

A character string is a one-dimensional array of characters. Thus, any operation on the string is performed on each element individually. The importance of this feature is illustrated in Chapter 4. There is no conversion between characters and numbers.

### 2.3.2 Statements

The branch and the specification statements are the two basic statement types. Branch statements are used only in user-defined functions. Their explanation will be deferred until defined functions are discussed. Examples of specification statements are:

```
X←5÷2
Y←'I AM A STRING'
Z←1 2 3 4
Z1←3×5+2
Z2←(3×5)+2
```

Specification statements assign to the variable on the left hand side of the arrow the result of evaluating the expression on the right hand side of the arrow. In the examples given previously, X is assigned 2.5, Y is assigned the character vector inside the apostrophes (each character in the vector is an element of vector Y), and Z is assigned a vector with the first four integers as elements. Two elements of a vector of numbers, not a vector of characters, are separated for input and output by at least one blank. The value of Z1 is 21, not 17, because order of execution is right to left. However, Z2 does have value 17 since parentheses are used.

### 2.3.3 Indexing Arrays

[i] written after a vector, or [i;j] written after an array (i or j possibly omitted), are called indices or index functions. Like subscripts in other languages, the indices are used to reference elements of vectors and arrays. For example, suppose

```
B←'I AM A STRING'
C↔1 2
  3 4
  5 6
```

Then

```
B[1]↔'I'
B[1 5 9]↔'I T'
C[1;2]↔2
C[1;]↔1 2
C[;2]↔2 4 6
C[2 3;2]↔4 6
C[1 2;1 2]↔1 2
           3 4
'ABCDE'[3]↔'C'
```

As illustrated above, the indices (subscripts) inside the brackets may be scalars, vectors, or arrays.

### 2.3.4 Functions

There are two kinds of functions (or operators): primitive functions which are built into the system, and defined functions which are defined by the user. Primitive functions will be considered first.

Every primitive function is either monadic (one argument) or dyadic (two arguments). Whether an argument may be a scalar, a vector, or an array depends on the function used. The form of function result, i.e. scalar, vector, or array, depends on the type of arguments used. (A scalar is not considered to be a vector of length one.)

Primitive functions are considered to be either scalar or mixed. Scalar functions are those which return a scalar result for scalar arguments. However, their arguments may be arrays, which are operated on element by element by the function. The shape of the result is the same as that of one of the arguments. For example, suppose *S*1←1 2 3 4 and *S*2←5 6 7 8 . To evaluate S1+S2, the addition operator is applied to corresponding elements in the two vectors, yielding the result 6 8 10 12. If S1 or S2 is a scalar, then the scalar is paired with every element of the vector in evaluating the function. If *S*3←5 then *RESULT*←*S*3+*S*2 or *RESULT*←*S*2+*S*3 assigns to RESULT the vector value 10 11 12 13.

Many function symbols are used to represent two different functions in APL. The meaning of the symbol depends on the number of arguments it has. For example, ⌈ is the ceiling (next integer not less than) function when used monadically (with one argument) and the maximum function when used dyadically (with two arguments). For example, ⌈3.5↔4 and 3⌈3.5↔3.5 .

APL has relational operators which take scalar arguments and whose results are 1 if the relation holds for the arguments and 0 otherwise. For example,

3≥4↔0   ¯2<3↔1   'A'='B'↔0

Scalar relation functions equals and not equals may be used with character arguments, but the other relations cannot. The logical functions or, and, etc. take logical arguments (0's and 1's) and return 0 or 1 as value. For example, if

A←1010   B←1100

then

A∧B↔1000   A∨B↔1110   ~A↔0101

Any dyadic scalar function symbol may be followed by a reduction symbol /. This has the effect of applying the function symbol between successive components of the argument. For example, +/X says to add together every component of vector X. Reduction may also be used along any coordinate of an array.

Mixed functions may be defined on numbers or characters. The shape of the result is not necessarily the shape of one of the arguments. A mixed function must have a non-scalar either as an argument or as a result. An example of a non-scalar or mixed dyadic function is catenate, symbolized by a comma. This function says to concatenate its two arguments. For example, *'AB','CD'↔'ABCD'* . If X is assigned 'AB' and Y is assigned 'CD', then *X,Y↔'ABCD'* also.

Some of the more useful mixed functions will now be explained. These explanations may need to be referenced when reading later chapters.

### 2.3.4.1 Index generator

If N>0, ι*N* is a vector whose elements are the first N integers. For example,

```
ι1↔1    ι5↔1 2 3 4 5
```

ι0 is the null vector; it prints as a blank.

### 2.3.4.2 Index of

The dyadic use of iota, *A*ι*B* , is very important in string handling problems. *A*ι*B* gives the least index of the occurrence of each element of B in A, where A must be a vector. If an element of B does not occur in A, then the function returns 1 plus the highest index of A. Suppose

```
B←'A S'
A←'I AM A STRING'
```

Then

```
Aι'A'↔3
AιB↔3 2 8
```

### 2.3.4.3 Size

If X is a vector, then ρ*X* gives the number of elements in X. If X is an array, ρ*X* gives the dimensions of X in the form of a vector result. For example, if X is array

```
1 2
3 4
5 6
```

then ρ*X*↔3 2 , denoting three rows and two columns in X.

### 2.3.4.4 Reshape

The dyadic function ρ can create an array. In such usage the first argument specifies the dimensions the array is to have. The second argument specifies a vector of elements to be in the array. The statement

```
A←2 3ρ1 2 3 4 5 6
```

defines A to be array

```
1 2 3
4 5 6
```

### 2.3.4.5 Ravel

The comma (,) used monadically rewrites an array as a vector. Hence, *B*←,*A* assigns B the value 1 2 3 4 5 6 .

### 2.3.4.6 Membership

The membership function ϵ takes two arguments; it yields a logical array that has the dimensions of the first argument. The result has ones in the positions where elements of the first argument are members of the second argument, and zeroes in all other positions. For example,

(ι5)∊2↔0 1 0 0 0

and

'*I AM A STRING*'∊'*AEIOU*'↔1 0 1 0 0 1 0 0 0 0 1 0 0

Parentheses are necessary around ι5 in this example because of the right to left rule for function evaluation.

### 2.3.4.7 Compress and Expand

Compress and expand operators used with two arguments are represented by the forward slash and the backward slash, respectively. A logical vector may be used to compress or expand a vector or array. In compressing character arrays, characters in the second argument are deleted at the positions where there are zeroes in the first argument. No changes are made in the positions in the second argument where there are ones in the first argument. In expanding character arrays, the result is the same as the second argument but with blanks inserted in positions where zeroes appear in the first argument. For example, suppose *I*←1 0 0 1 and *A*←'*ROAM*' *B*←'*RM*' . Then *I*/*A*↔'*RM*' and *I**B*↔'*R  M*' .

### 2.3.5 Defined Functions

Defined functions are used to extend the language. The following is an example of a function definition.

```
   ∇DIM                ] FUNCTION HEADER
[1] SUM←(ρA)+ρB        ⊤
[2] AVER←SUM÷2         | FUNCTION BODY
   ∇                   ⊥
```

The del (∇) character before DIM indicates the beginning of the function definition mode. The last del ends function definition. DIM is the name of the function to be defined; [i] stands for statement number i. The statements constitute the function body. After function definition the body is associated with function name DIM. DIM could be called by:

```
A←'SIZE'
B←'SIZE1',A
DIM
AVER
```

DIM calculates the average size of A and B. Since A contains four characters and B contains nine characters, the value 6.5 is printed. DIM can be rewritten to have two arguments. The function header would be changed to

```
∇A DIM B
```

The function might then be called by

```
Z←'SIZE'
Z DIM 'SIZE1',Z
```

Again 6.5 is the result.

The basic format of a branch statement is →I. If I is a number or a label, the program branches to the corresponding statement in the function definition. If I is the null vector, the next instruction in statement number order is executed. If I=0 the execution is finished.

Branch statements are used in the following function definition.

```
     ∇ Z←DOUBLE STR
[1]    Z←''
[2]   LOOP:Z←Z,2ρ1↑STR
[3]    STR←1↓STR
[4]    →(0<ρSTR)/LOOP
     ∇
```

The above function DOUBLE doubles every letter of STR. Z is assigned the null string in statement 1. In statement 2 Z is concatenated with two copies of 1↑*STR* , the first character of STR. The first character of STR is dropped from STR in statement 3. Statement 4 causes a branch to the statement labelled LOOP if there is at least one more character in STR; otherwise the program stops.

Suppose DOUBLE is used in a statement, for example,

*STRING*←(*DOUBLE* '*ABC*'),(*DOUBLE* '*X*'),'*Y*'

Then STRING will have the value 'AABBCCXXY'.

The previous example illustrates that a defined function does not have to be referred to any differently from a primitive function. This means that a defined function may also appear in other function definitions.

Some defined functions are included in libraries available to the user. Recursive function definitions are allowed. Also, APL/360 allows functions to be traced as they are being executed and function definitions to be changed.

## 2.4 PL/I

PL/I is a general purpose programming language that can be used for a wide variety of problems. The original specifications for PL/I were written by the Advanced Language Development Committee of the SHARE FORTRAN Project, a group formed by SHARE and IBM.

PL/I contains many of the features of COBOL, FORTRAN, and ALGOL. Also, to some extent, PL/I was influenced by APL.

An important feature of PL/I is its modularity. The language is such that a user need only learn that subset of PL/I applicable to his problems.

PL/I is discussed in [2] and [18].

### 2.4.1 Data Types

Data fall into the categories of problem data and program control data. The latter category will not be discussed. Problem data may be divided into arithmetic data and string data. Attributes of a variable are declared in a DECLARE statement anywhere in the program. However, if any attribute is not declared explicitly, a default attribute is assigned.

Attributes of arithmetic variables are BASE (binary or decimal), SCALE (fixed or floating point), MODE (real or complex), and precision.

String data may be either character or bit strings. All string operations and functious may be performed on either kind. Strings may be declared to be of fixed or varying lengths. However, a maximum length must still be specified for a varying length string.

Both arithmetic data and string data may be organized into arrays and structures. A structure may contain both arithmetic and string variables, whereas all elements of an array must have identical attributes.

### 2.4.2 Block Structure

An important characteristic of PL/I is its block structure. Blocks are groups of statements that delimit the scope of variables. There are two kinds of blocks, procedures and BEGIN blocks.

Procedures are subroutines which are activated explicitly by being invoked. They may be passed parameters. BEGIN blocks are activated implicitly by being reached. No parameters are passed to BEGIN blocks.

### 2.4.3 Statement Types

PL/I has several different statement types. These include descriptive statements, such as DECLARE; I/O statements, such as GET and PUT; data movement and computational statements, such as assignment statements; program structure statements, such as PROCEDURE, BEGIN; and control statements, such as GO TO, IF, DO, CALL, RETURN. IF and GO TO statements provide, respectively, conditional and unconditional branching. IF statements can be quite complex. DO groups, delimited by DO and END statements, are used for control purposes; they can specify how many times and under what conditions a group of statements is to be executed. Some of the statements will be illustrated in the program following the PL/I discussion.

### 2.4.4 String Capabilities

Since PL/I has been influenced by FORTRAN, COBOL, and ALGOL, it is not usually considered a language in which to do string manipulation problems. However, there are several features of PL/I which permit fairly good string processing. In this respect PL/I differs from most general purpose programming languages.

Rosin has discussed these useful string features in a 1967 article [18]. Strings may be declared to be of fixed or varying length; fixed length is the default. String constants are delimited by apostrophes, e.g. 'I AM A STRING'.

Strings may be concatenated using the operator ||. The function LENGTH(string) returns the size of string. The relation operator equals (=) may be used to compare two strings. Also, all of the other relational operators can be used on string operands. The result depends on the collating sequence of the character codes. A replication factor may be placed before a character string constant, but not before the name of a character string. The factor, which is a constant, indicates how many times the character string constant is to be repeated.

The two extremely useful built-in string functions are SUBSTR and INDEX. SUBSTR(string,i,n) gives the n character long substring of string that begins in position i. If n is absent, then the rest of string from character i on is given. SUBSTR may also appear on the left hand side of assignment statements as a pseudo-variable, thus allowing values to be assigned to substrings. For example, the statement

```
SUBSTR(STR,3,9)='ABCDEFGHI';
```

replaces the third through eleventh positions of STR with the first nine letters of the alphabet. The INDEX function essentially does SNOBOL-like matching of a simple pattern. INDEX(string,substring) finds the left-most occurrence of substring in string. The position of the first character in the matched portion of string is returned, and 0 is returned if substring is not contained in string. This is a generalization of the iota operator of APL.

There are other string functions as well.

REPEAT(string,N) does essentially the same thing as a replication factor. However, string is not restricted to being a character string constant; it may be the name of a string. TRANSLATE(string,table1,table2) translates each character in string which appears in table1 to the corresponding character in table2. In the following example table1 is 'IG' and table2 is 'AD'.

```
     A = TRANSLATE('SING','IG','AD');
```

assigns 'SAND' to A. VERIFY(string1,string2) verifies that every character of string1 is present in string2. If so, 0 is returned. If not, the position (index) of the first character in string1 not present in string2 is returned.

A sample PL/I program follows that counts the number of I's and E's in an input card.

```
PR:
         PROCEDURE OPTIONS (MAIN);
         DECLARE X CHAR (25) VARYING,
                 SUM FIXED;
START:
         GET LIST (X);
         SUM = 0;
         DO I=1 TO LENGTH(X);
             IF SUBSTR(X,I,1)='I' |
                SUBSTR(X,I,1)='E'
             THEN SUM=SUM+1;
         END;
         PUT LIST (SUM);
         END PR;
```

# 3 SAMPLE STRING HANDLING PROBLEMS

This chapter contains two examples of easy string handling problems and one complex problem. These problems help show the different ways that basic string operations, which are discussed in detail in Chapter 4, are done in each language. Also, they use many of the language features discussed in Chapter 2.

Problem 1 is sorting N strings into alphabetical order. Problem 2 involves listing all words that begin with a vowel that occur in a line of text. Problem 3 is a rather complex text matching problem.

## 3.1 PROBLEM 1

The following strings are to be sorted:

CATCH
THROW
OUTFIELD
BASEBALL
BASE
CATCHER

A bubble sort program will be written in all four languages. In the first stage the bottom two strings, the N-1st and the Nth, are compared; the alphabetically earlier of the two strings is bubbled up and compared with the N-2nd string; the earlier of the two is bubbled up and compared with the N-3rd, etc., until the proper string is at the top of the sequence of strings. In the second stage the above process is repeated; the top item is not checked. After the second stage the top two strings are in order. The bubble sort continues until a stage when no two strings are interchanged.

The flowchart, which applies to all four bubble sort programs, is given in Figure 3.1. The algorithm is a common form for a bubble sort and is found in reference [6]. It was relatively easy on the basis of programming time to write the PL/I, SNOBOL4, and APL programs from the flowchart. The TRAC program took more time to code. A bubble sort is not the best method for sorting in APL, so an alternate method is also given in the chapter.

### 3.1.1 SNOBOL4

(Refer to Figure 3.2.)

The first input card contains N, the number of strings to be sorted. Succeeding cards contain the strings themselves. A one-dimensional array A of N items is created by the statement

```
A = ARRAY(N)
```

Each string is a member of array A. Notice that the indices of array elements are denoted by <>'s, not parentheses.

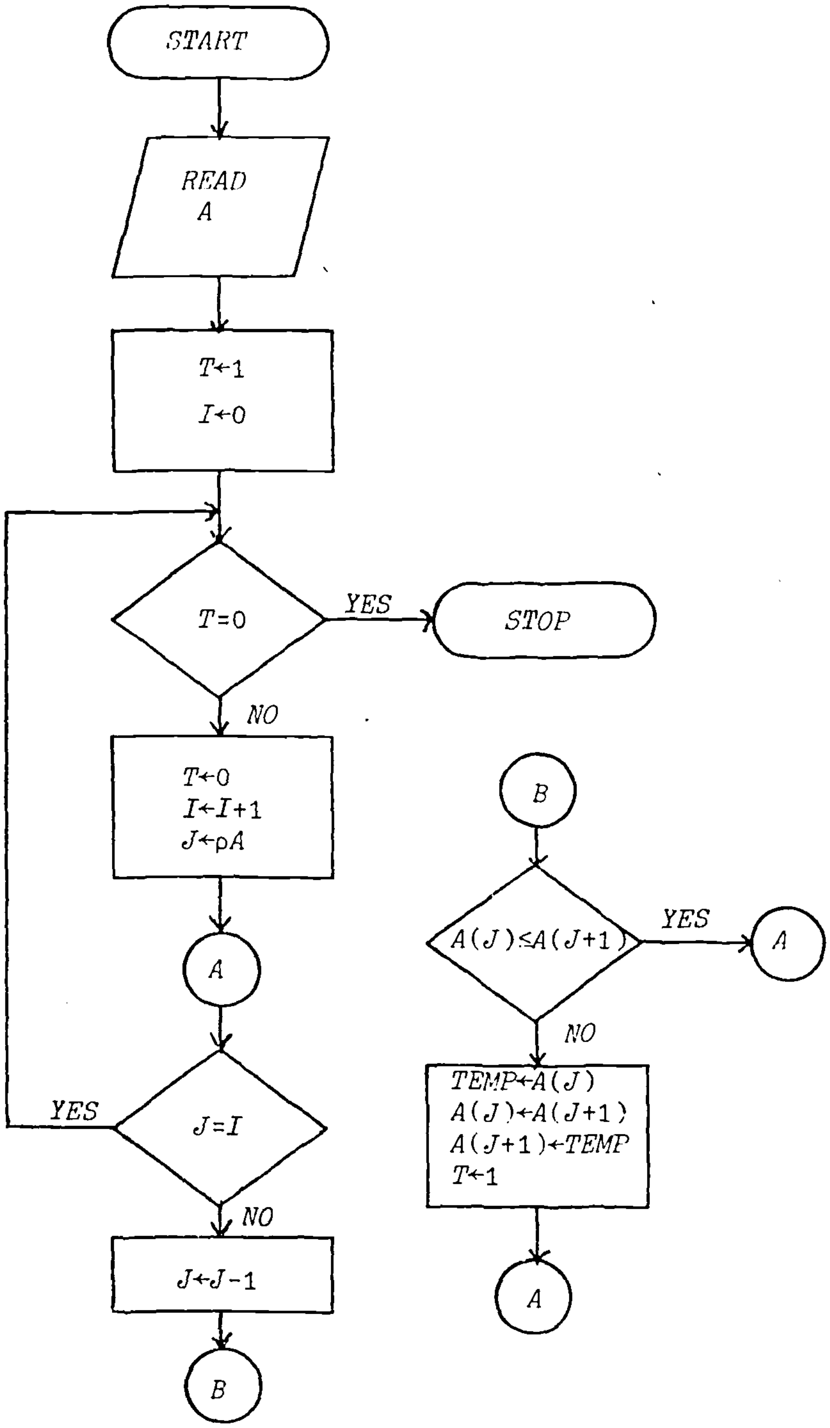


Figure 3.1  Flowchart for Problem 1.

```
* THIS PROGRAM IS SIMILAR TO THE ONE IN THE SNOBOL IV MANUAL
* INITIALIZE STAGE NO.
          I = 0
*
* GET NUMBER OF ITEMS TO BE SORTED
*
          N = TRIM(INPUT)                       :F(ERROR)
          A = ARRAY(N)
*
* READ IN THE ITEMS
*
READ      I = I + 1
          A<I> = TRIM(INPUT)                    :F(GO) S(READ)
*
* SORT THE LIST
*
GO        I = 0
          T = 1
SORT2     EQ(T,0)                               :S(PR)
          J = N
          T = 0
          I = I + 1
SORT1     EQ(I,J)                               :S(SORT2)
          J = GT(J,1) J - 1
          LGT(A<J>,A<J + 1>)                    :F(SORT1)
*
SWITCH    TEMP = A<J>
          A<J> = A<J + 1>
          A<J + 1> = TEMP
          T = 1                                 :(SORT1)
*
* PRINT SORTED LIST
*
PR        M = 1
PRINT     OUTPUT = A<M>                         :F(END)
          M = M + 1                             :(PRINT)
*
END

BASE
BASEBALL
CATCH
CATCHER
OUTFIELD
THROW
```

Figure 3.2 SNOBOL4 Program for Problem 1.

### 3.1.2 PL/I

(Refer to Figure 3.3.)

The PL/I and SNOBOL4 programs are very similar. However, in PL/I a maximum length for an element of A must be given (8 in this example). In SNOBOL4 it is not necessary to specify maximum lengths of array elements.

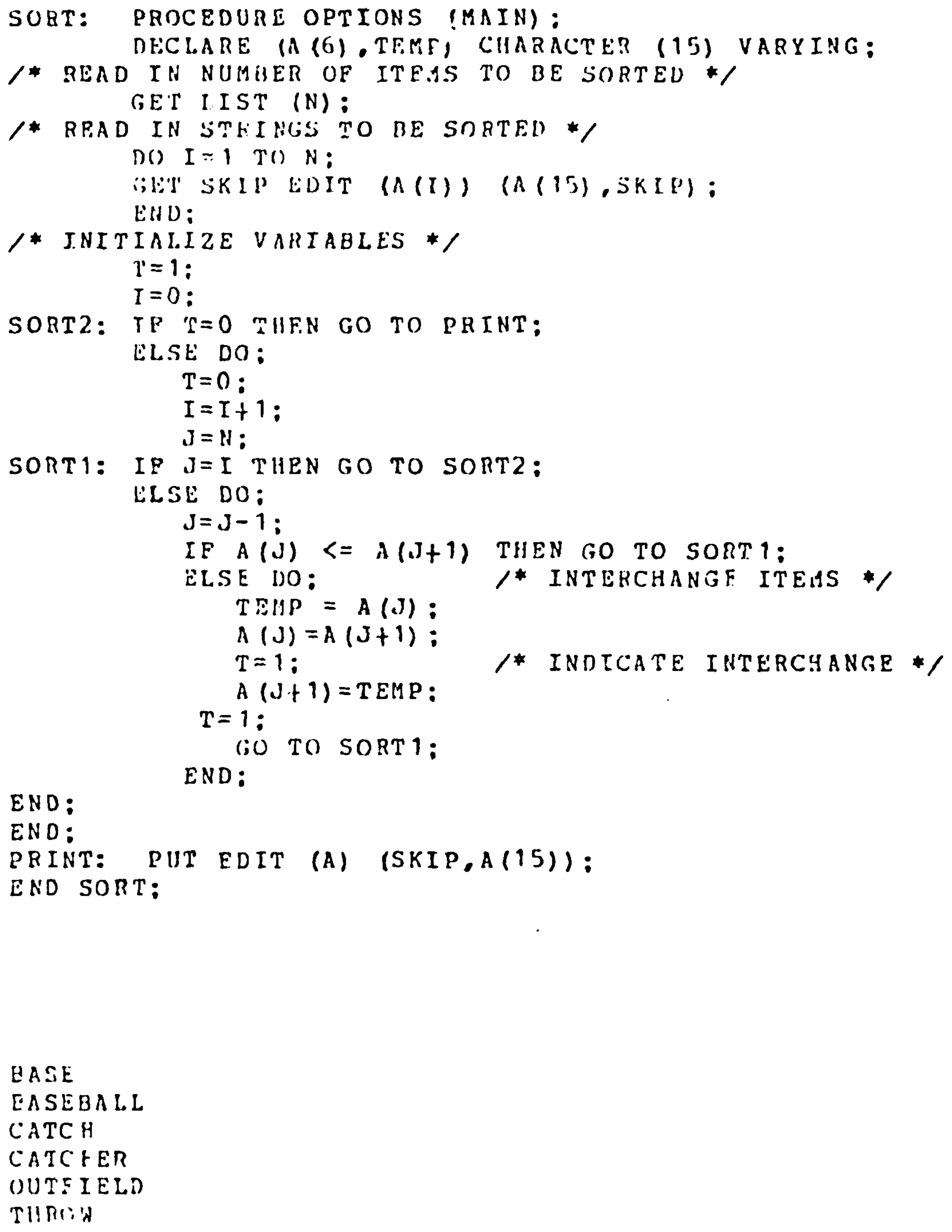
### 3.1.3 APL

(Refer to Figure 3.4.)

Since there is no collating sequence in APL, it is necessary to use a string S containing the letters of the alphabet in order preceded by a blank for reference in getting the proper lexical order.

The sequence of six strings to be sorted is stored as a two-dimensional array A. J is the index of the array element in A being considered, L indexes the position or column of the array member, and I is the stage number of the bubble sort process. In the previous examples in SNOBOL4 and PL/I, A was a vector (one-dimensional array of character strings), whereas in APL it is a two-dimensional array of characters.

PL/I and SNOBOL4, when comparing two strings of unequal lengths, left justify the shorter of the two strings and pad to the right with blanks. However, in APL a string is a vector of characters. Since the dimensionality of two



```
SORT:  PROCEDURE OPTIONS (MAIN);
       DECLARE (A(6),TEMP) CHARACTER (15) VARYING;
/* READ IN NUMBER OF ITEMS TO BE SORTED */
       GET LIST (N);
/* READ IN STRINGS TO BE SORTED */
       DO I=1 TO N;
       GET SKIP EDIT (A(I)) (A(15),SKIP);
       END;
/* INITIALIZE VARIABLES */
       T=1;
       I=0;
SORT2: IF T=0 THEN GO TO PRINT;
       ELSE DO;
          T=0;
          I=I+1;
          J=N;
SORT1: IF J=I THEN GO TO SORT2;
       ELSE DO;
          J=J-1;
          IF A(J) <= A(J+1) THEN GO TO SORT1;
          ELSE DO;          /* INTERCHANGE ITEMS */
             TEMP = A(J);
             A(J)=A(J+1);
             T=1;           /* INDICATE INTERCHANGE */
             A(J+1)=TEMP;
           T=1;
             GO TO SORT1;
          END;
END;
END;
PRINT:  PUT EDIT (A) (SKIP,A(15));
END SORT;
```

```
BASE
BASEBALL
CATCH
CATCHER
OUTFIELD
THROW
```

Figure 3.3 PL/I Program for Problem 1.

```
    ∇ SORT
[1]    I←0
[2]    S←' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[3]    A←⎕ρ⍞
[4]    →HERE
[5]    T←1
[6]   LOOP1:→(T=0)/OUT
[7]   HERE:T←0
[8]    I←I+1
[9]    J←(ρA)[1]
[10]  TEST:→(J=I)/LOOP1
[11]   J←J-1
[12]   L←0
[13]  LOOP2:L←L+1
[14]   →(L=(ρA)[2]+1)/TEST
[15]   →((S⍳A[J;L])≥S⍳A[J+1;L])/YES
[16]  NO:→TEST
[17]  YES:→((S⍳A[J;L])=S⍳A[J+1;L])/LOOP2
[18]   TEMP←A[J;]
[19]   A[J;]←A[J+1;]
[20]   A[J+1;]←TEMP
[21]   T←1
[22]   →TEST
[23]  OUT:→0
    ∇
```

```
      SORT
CATCH   THROW   OUTFIELDBASEBALLBASE    CATCHER
⎕:
      6 8
```

```
      A
BASE
BASEBALL
CATCH
CATCHER
OUTFIELD
THROW
```

Figure 3.4 APL Program for Problem 1.

vectors must match to be compared, and since APL strings are character vectors, two APL strings must be of the same length to be compared. Therefore the programmer must provide for padding.

The APL bubble sort program is similar to the SNOBOL4 and PL/I versions. However, it is not the best way of writing a sort in APL. Since APL has such a wide variety of primitives, there are more concise ways to code the sort problem. One of these ways is found in Katzan [12]. His way uses the decode function ⊥ and transpose function ⍉ as well as the size and index functions.

The expression *R⊥X*, where R is a radix and X is a vector of digits, denotes the value of X evaluated in a number system with radix R. For example, the value of 10⊥1 2 3 is 123. Thus, if *S*←' *ABCDEFGHIJKLMNOPQRSTUVWXYZ*' the value of 27⊥*S*⍳'*BC*' would be $(27^1 x3)+(27^0 x4)$ or 85.

The expression ⍉*X*, where X is an array, returns the transpose of X. For example, if *SMSTR*←2 4⍴'*THEYCAME*' then ⍉*S*⍳*SMSTR* is the array

```
21  4
 9  2
 6 14
26  6
```

Now consider

```
STRING←6 8⍴'CATCH   THROW   OUTFIELDBASEBALLBASE    CATCHER '
```

Then the function

```
    ∇R←SORT STRING;ALPH
[1] ALPH←' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[2] R←STRING[⍋(⍴ALPH)⊥⍉ALPH⍳STRING;]
    ∇
```

will order the elements of STRING.

STRING is the array:

```
CATCH
THROW
OUTFIELD
BASEBALL
BASE
CATCHER
```

Tracing through the operation step by step:

<u>Step 1</u>

The value of

```
ALPH⍳STRING
```

is

```
 4   2  21   4   9   1   1   1
21   9  19  16  24   1   1   1
16  22  21   7  10   6  13   5
 3   2  20   6   3   2  13  13
 3   2  20   6   1   1   1   1
 4   2  21   4   9   6  19   1
```

Each row contains the indices of a row of STRING in ALPH.

<u>Step 2</u>

The value of

```
⍉ALPH⍳STRING
```

is

```
   4  21  16   3   3   4
   2   9  22   2   2   2
  21  19  21  20  20  21
   4  16   7   6   6   4
   9  24  10   3   1   9
   1   1   6   2   1   6
   1   1  13  13   1  19
   1   1   5  13   1   1
```

The function ⍉ transposes the matrix obtained in Step 1.

Step 3

The value of (⍴*ALPH*)⊥⍉*ALPH*⍳*STRING* is

```
4.291988451E10
2.234358071E11
1.761941507E11
3.244612824E10
3.244608781E10
4.291988864E10
```

The first number is equal to

$$(27^{7}x4)+(27^{6}x2)+(27^{5}x21)+(27^{4}x4)+$$
$$(27^{3}x9)+(27^{2}x1)+(27^{1}x1)+(27^{0}x1)$$

The other numbers are calculated in the same way.

Step 4

The function ⍋ assigns ranks to the elements of (⍴*ALPH*)⊥⍉*ALPH*⍳*STRING* . The value of ⍋(⍴*ALPH*)⊥⍉*ALPH* ⍳*STRING* is 5 4 1 6 3 2 .

Step 5

Finally indexing the rows, the value of

*STRING*[⍋(⍴*ALPH*)⊥⍉*ALPH*⍳*STRING*;]

is the sorted list

```
     BASE
     BASEBALL
     CATCH
     CATCHER
     OUTFIELD
     THROW
```

### 3.1.4 TRAC

(Refer to Figure 3.5.)

The TRAC version of the bubble sort provides many contrasts with the previous programs. For instance, there are no arrays in TRAC. However, there is a way to get around this deficiency. The variables that need to be array-like could be named A1, A2, A3, etc. Then :(A:(J)) can be used to reference a[j].

In TRAC, as in APL, character strings that need to be compared must have the same length. Otherwise, when comparing two strings of unequal lengths, the shorter of the two will be right-justified and padded to the left with zeros. This contrasts with the left-justification of character strings done in SNOBOL4 and PL/I. For alphabetization, therefore, the program must provide left justification.

There is no equals operator that may be used to compare two numbers. In SORT1 of the TRAC program GR must be used twice to test for equality.

The TRAC bubble sort is organized as a series of calls to NEXT, SORT2, NEW, SORT1, LOOP1, LOOP2, and PRINT.

```
:(DS,N,6)'
:(DS,T,0)'
:(DS,J,1)'
:(DS,NEXT,(:(DS,T,1):(DS,I,0):(SORT2)))'
:(DS,SORT2,(:(GR,:(T),0,(:(NEW)),(:(DS,K,1):(PRINT)))))'
:(DS,LOOP2,(:(DS,TEMP,:(A:(J))):(DS,A:(J),:(A:(AD,:(J),1))):
(DS,A:(AD,:(J),1),:(TEMP)):(DS,T,1):(SORT1))))))'
:(DS,SORT1,(:(GR,:(J),:(I),(:(LOOP1)),(:(GR,:(I),:(J),(:(LOO
P1)),(:(SORT2)))))))'
:(DS,LOOP1,(:(DS,J,:(SU,:(J),1)):(LG,:(A:(J)),:(A:(AD,:(J),1
)),(:(LOOP2)),(:(SORT1)))))'
:(DS,PRINT,(:(PS,:(A:(K))):(DS,K,:(AD,:(K),1)):(GR,K,N,,(:(P
RINT)))))'
:(DS,NEW,(:(DS,T,0):(DS,I,:(AD,:(I),1)):(DS,J,:(N)):(SORT1))
)'
:(DS,ASSIGN,(:(DS,I,0):(ALOOP)))'
:(DS,ALOOP,(:(DS,I,:(AD,:(I),1)):(GR,:(I),:(N),(:(NEXT)),(:(
DS,A:(I),:(RS)):(ALOOP)))))))'
:(ASSIGN)'CATCH   'THROW   'OUTFIELD'BASEBALL'BASE    'CATCH
ER '
BASE    BASEBALLCATCH   CATCHER OUTFIELDTHROW
```

Figure 3.5 TRAC Program for Problem 1.

SORT2 tests for T greater than 0, which indicates that more interchanges are necessary. If T is not greater than 0, K is initialized to 1 and the program branches to PRINT. When T is greater than 0, NEW is called.

NEW resets T to 0, increments I by 1, sets J equal to N, and calls SORT1.

In SORT1 the GR primitive is used to compare J with I. If J is greater than I, LOOP1 is called. Otherwise J and I must be compared again, using GR. If I is not greater than J, then I and J must be equal and the program branches to SORT2.

LOOP1 decrements J by 1. Next, a[j] and a[j+1] are compared using the lexical ordering primitive LG. If a[j] is lexically greater than a[j+1], LOOP2 is called to interchange the two. Otherwise SORT1 is called.

LOOP2 switches a[j] and a[j+1] and sets T to 1 to indicate that an interchange has taken place. SORT1 is called.

PRINT is defined recursively. Each time PRINT is called it prints a string, increments K, and calls itself. When K exceeds N, the program stops.

## 3.2 PROBLEM 2

(See Figures 3.6, 3.7, 3.8, and 3.9.)

```
         TEXT = TRIM(INPUT)
         TEXT1 = TEXT
LOOP     TEXT BREAK(' ') . WORD LEN(1) =   :F(END)
CHECK    IT = SIZE(WORD) - 1
         WORD  ANY('AEIOU')  LEN(IT)                 :F(LOOP)
         OUTPUT = WORD                               :(LOOP)
END
```

```
I
ALL
A
```

Figure 3.6 SNOBOL4 Program for Problem 2.

```
VOWEL: PROCEDURE OPTIONS (MAIN);
       DECLARE WORD CHARACTER (15) VARYING,
               TEXT CHARACTER (80) VARYING,
               TEXT1 CHARACTER (80) VARYING,
               L    CHARACTER (1);
       GET EDIT (TEXT) (A(80));
       TEXT1=TEXT;
LOOP:  PT=INDEX(TEXT,' ');
       IF PT = 0 THEN GO TO PRINT1;
       WORD = SUBSTR(TEXT,1,PT-1);
       TEXT = SUBSTR(TEXT,PT+1) ;
       L = SUBSTR(WORD,1,1);
       IF L='A' | L='E' | L='I' |
          L='O' | L='U'
       THEN PUT EDIT (WORD) (A(15));
       GO TO LOOP;
PRINT1: END VOWEL;
```

```
I              ALL            A
```

Figure 3.7 PL/I Program for Problem 2.

```
    ∇ VOW2
[1]    TEXT←⍞
[2]    TEXT1←TEXT
[3]    TEXT←' ',TEXT,' '
[4]    LIST←''
[5]    VEC←(TEXT∊' ')/⍳⍴TEXT
[6]    I←0
[7]   INC:→((I←I+1)=⍴VEC)/0
[8]    WORD←TEXT[VEC[I]+⍳(VEC[I+1]-(VEC[I]+1))]
[9]   TEST:→(WORD[1]∊'AEIOU')/PR
[10]   →INC
[11] PR:LIST←LIST,' ',WORD
[12]   →INC
    ∇


      VOW2
I WANT TO LIST ALL WORDS THAT BEGIN WITH A VOWEL


      LIST
 I ALL A
```

Figure 3.8 APL Program for Problem 2.

```
:(DS,TEXT,I WANT TO LIST ALL WORDS BEGINNING WITH A VOWEL)'
:(DS,TEXT1,::(TEXT))'
:(SS,TEXT, )'
:(DS,VOWEL,AEIOU)'
:(DS,WORD,(::(CS,TEXT)))'
:(DS,CHAR,(::(CC,W)))'
:(DS,NEWWORD,(:(GR,:(RP,TEXT),0,(:(DS,W,:(WORD)):(DS,LET,:(C
HAR)):(COMPAR))))))'
:(DS,COMPAR,(:(EQ,:(LET),::(CC,VOWEL),(:(PRINT)),(:(GR,:(RP,
VOWEL),0,(:(COMPAR)),(:(TEST))))))))'
:(DS,PRINT,(:(CR,W):(PS,:(W) ):(TEST)))'
:(DS,TEST,(:(CR,VOWEL):(NEWWORD))))'
:(NEWWORD)'
I ALL A
```

Figure 3.9 TRAC Program for Problem 2.

The purpose of this problem is to list all the words in a line of text that begin with a vowel. For simplicity there is no punctuation.

Words have to be isolated. In SNOBOL4 the BREAK function, in conjunction with a conditional variable WORD, does this. The ANY function of SNOBOL4 is convenient for matching any of the vowels with the first character of WORD. In PL/I each vowel must be compared individually. Again SNOBOL4's pattern matching superiority is apparent. The SNOBOL4 and PL/I programs dispose of a word in TEXT after it is assigned to variable WORD.

A different approach is taken in APL since TEXT is an array. The index of each blank character is placed in vector VEC. Each word is isolated and checked.

The TRAC program is organized as a series of calls to NEWWORD, COMPAR, PRINT, and TEST. The cursor of VOWEL must be reset before comparisons with each word. W is the current word under consideration. LET is the first letter in the current word.

## 3.3 PROBLEM 3

An interesting problem that illustrates many of the operations needed in string handling is the following.

Consider a student sitting at a terminal who is answering questions in a foreign language drill. The interactive system types a question that the student is to answer. If the student types the correct answer, the system responds with an R and types the next question. If the student missed the answer, he must try another reply. It would be helpful for the student to receive feedback that some of his answer was correct. For example, consider this hypothetical drill in English. The student's answers are preceded by a question mark.

```
What is the capital of France?
?Marseilles
_ar__
?Paras
Par_s
?Paris
R


What are the three R's?
?reeding, riting, awrithmetick
re_ding, _riting, arithmetic
?reading, writing, arithmetic
R
```

The procedure for comparing the student's answer with the correct answer is as follows. If the two answers are of equal length, they are compared, and R is returned if they are the same. If the two answers are not of equal length or are of equal length and not the same, the student answer is searched from left to right for n-character length sequences of the correct answer.

Assume that the the value of n is first 7, then 2. In the second drill question 'reading' would be the first sequence the student answer is searched for, 'eading,' is the second seven character length sequence, 'ading, ' the third, etc. No match occurs until 'riting,'.

When a match occurs, the letters following the matched sequence in the correct answer (M) and student answer (S) are searched one by one until the letter in M and the corresponding letter in S are not the same. For example, after 'riting,' is found in S the characters ' ' and 'a' will also be matched. In programming the problem, filler characters, the asterisk and the slash, are substituted for the matched characters in M and S, respectively. For instance, in the previous example, after 'riting, a' is matched, M and S would be:

M

reading, w*********rithmetic

S

reeding, /////////writhmetick

In future match attempts substrings with /'s and *'s are ignored. The sequence 'rithmet' would match a substring in M successfully, and the subsequent 'ic' would also match. Thus, after all 7-length sequences are tried, M and S would be:

M

reading, w******************

S

reeding, ////////w/////////k

Next, M is searched for all possible 2-character length sequences in S that match M substrings. 're' matches, but no additional characters do, so

M

```
**ading, w*******************
```

S

```
//eding, ////////w//////////k
```

The process continues until all possible substrings have been tried.

The M string is converted to an answer for the student. Every asterisk now in M will print as the character it stands for. For example, the letter 'r' will be substituted for the first letter in M in the answer, and 'e' for the second. Any character, other than a blank, will be replaced in the answer by the underline character (_). Blanks are given in the returned answer. In addition to an answer with blanks and underlines, the student receives a percentage of the letters in his answer that appear in the correct answer.

The flowchart for the program (Figure 3.10) follows.

SNOBOL4 has many string manipulating functions that were useful in writing the program. The SUBSTR and INDEX functions of PL/I were sufficient to do the necessary string processing in that language. However, the program was not as easy to do in APL/360. Even though APL provides indexinq

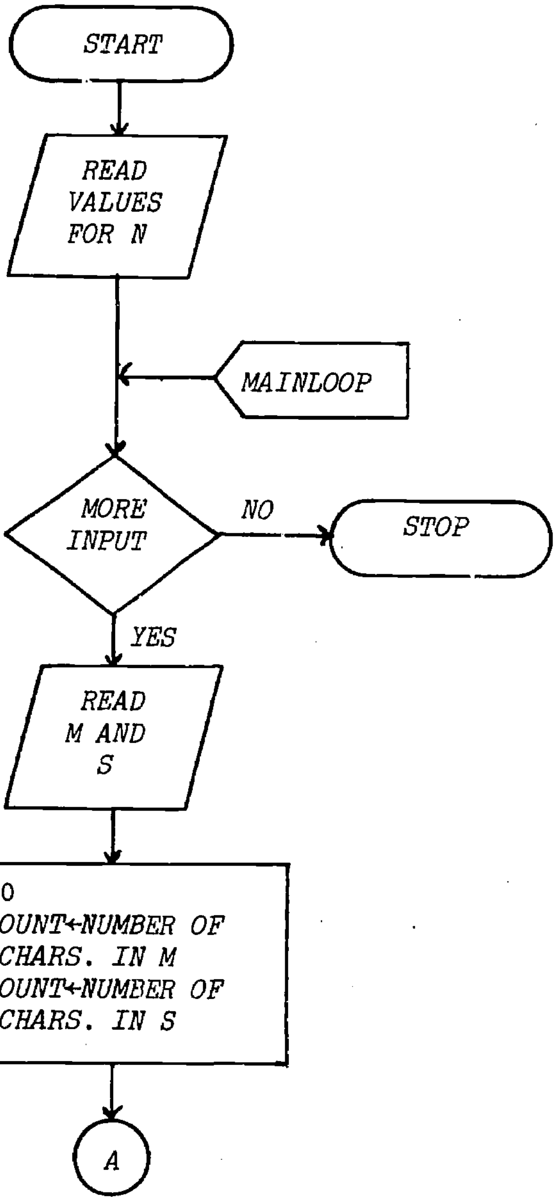


Figure 3.10 Flowchart for Problem 3.

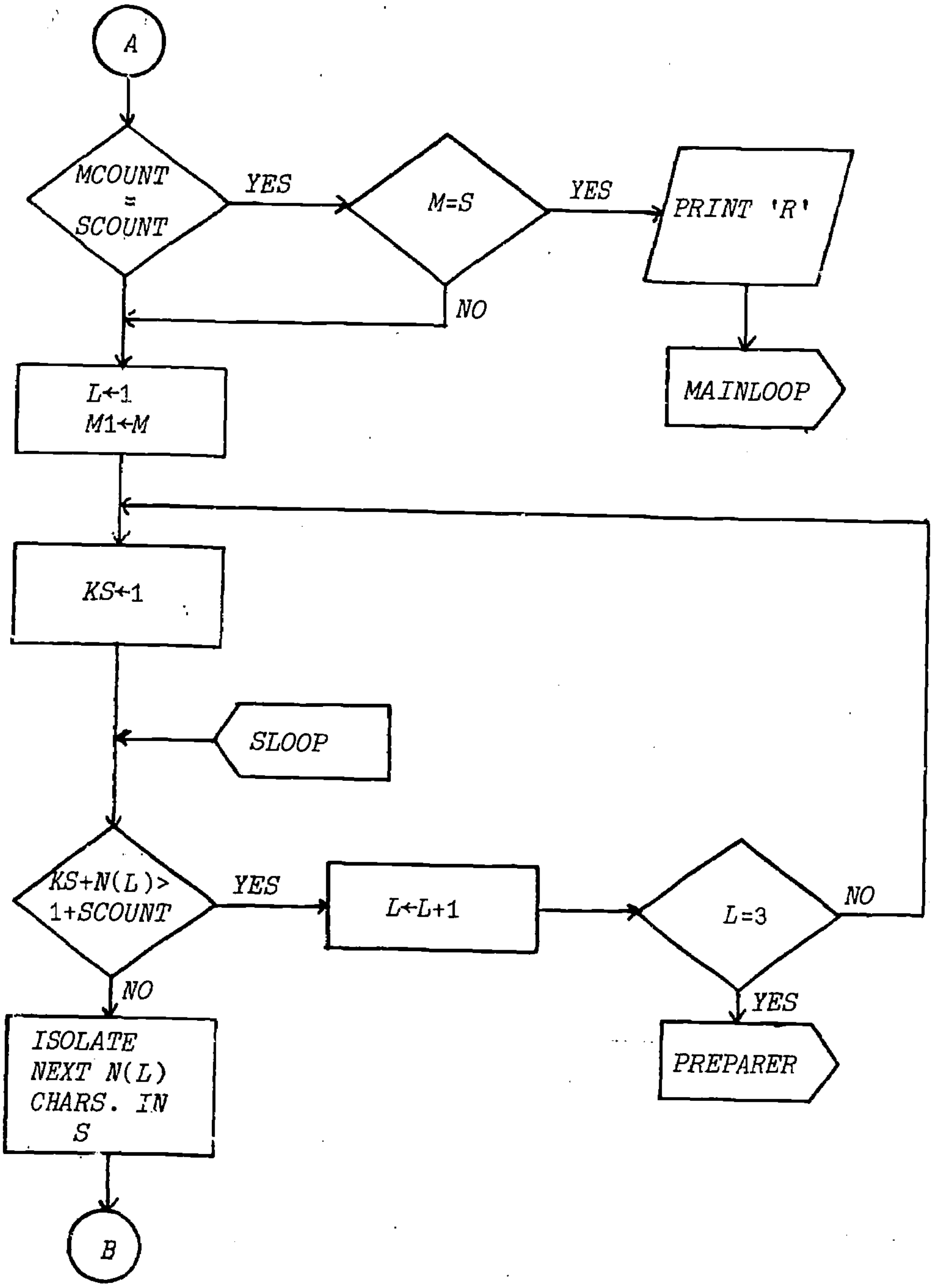


Figure 3.10 (cont.)

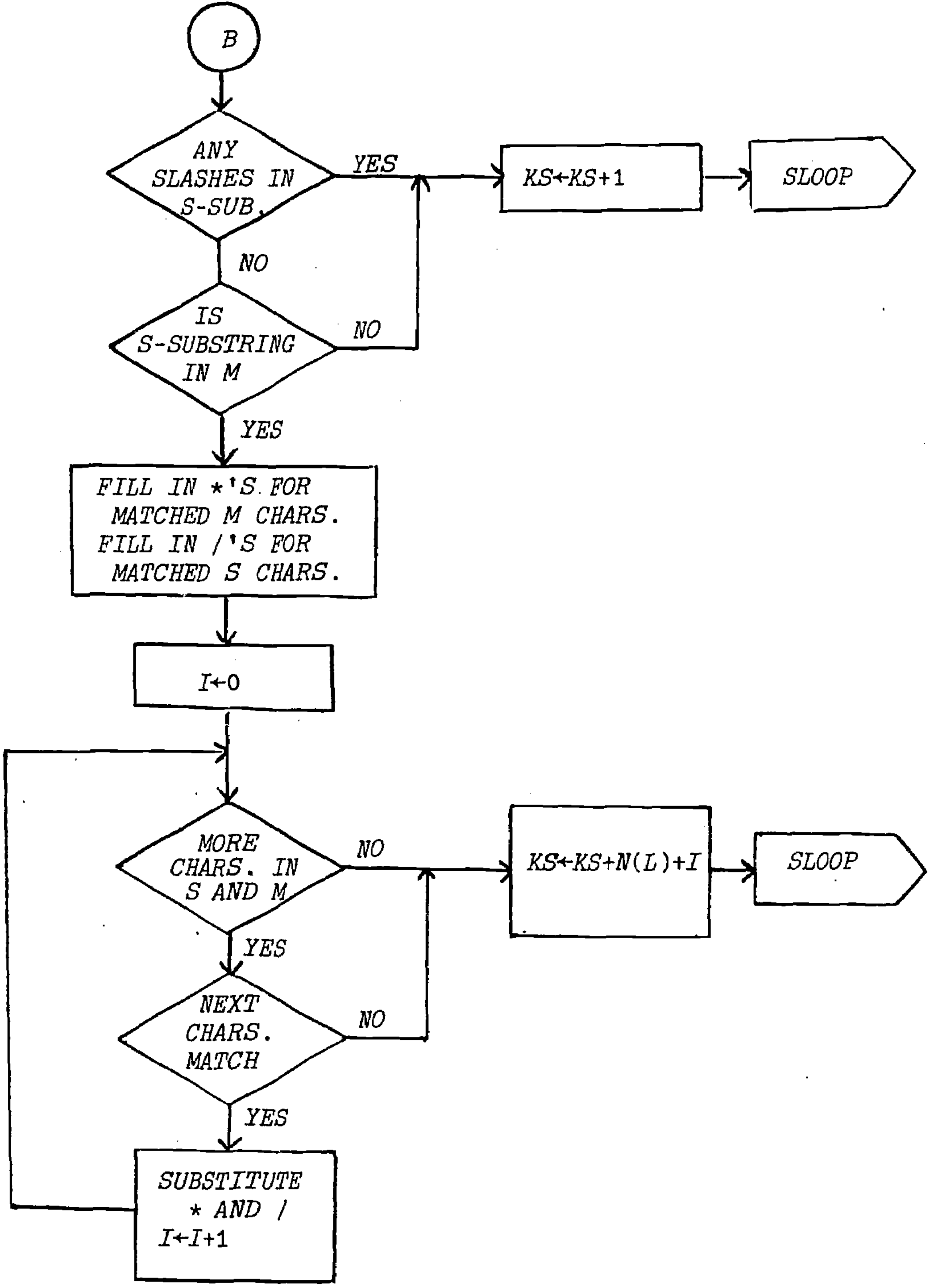


Figure 3.10 (cont.)

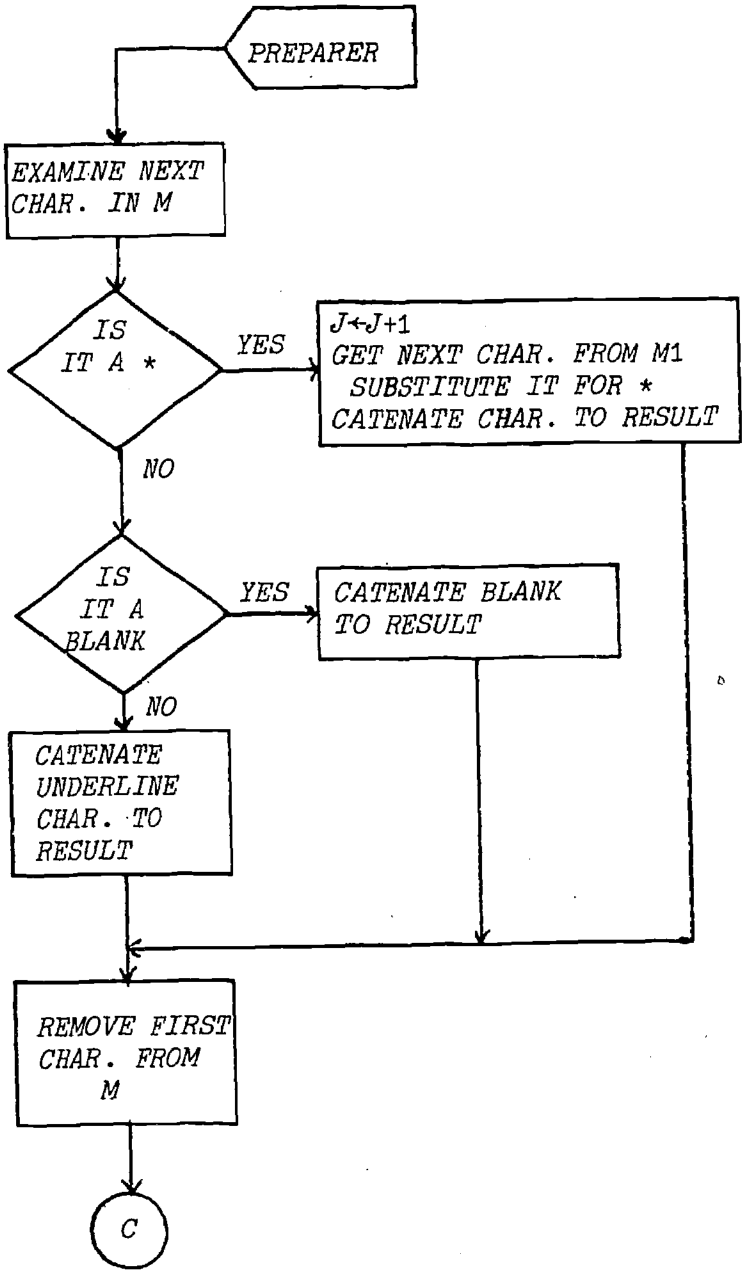

PREPARER
EXAMINE NEXT CHAR. IN M
IS IT A *
YES
J←J+1 GET NEXT CHAR. FROM M1 SUBSTITUTE IT FOR * CATENATE CHAR. TO RESULT
NO
IS IT A BLANK
YES
CATENATE BLANK TO RESULT
NO
CATENATE UNDERLINE CHAR. TO RESULT
REMOVE FIRST CHAR. FROM M
C

Figure 3.10 (cont.)

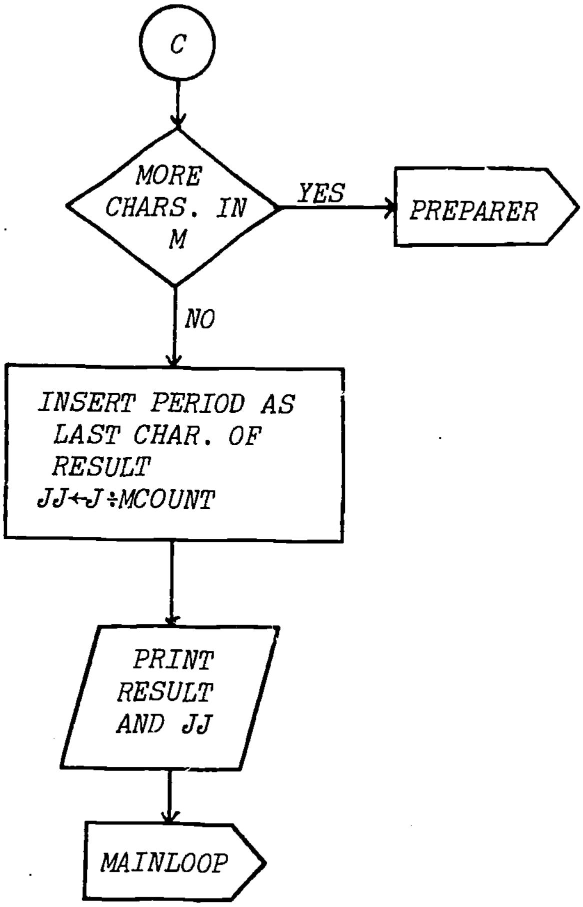


Figure 3.10 (cont.)

(the iota operator), it lacks an equivalent of the PL/I INDEX function.

Some of the variable names used are the same in all the programs. M is the correct answer; M1 is a copy of M. S is the student's answer. M and S change as matches are found. J counts the number of characters that match; JJ is the fraction of characters (J/(size of M)) that matched. N indicates how many characters are to be matched at once. N must be less than or equal to the minimum of the sizes of M and S. To be useful, however, the values of N should be small. L indicates which value of N is currently being used. RESULT is the string that is returned to the student.

As matches occur, asterisks replace the matched characters in M, and slashes replace the matched characters in S.

In the PL/I and APL programs, KS is equal to the position of the first character in the S-substring that is about to be checked. However, in the SNOBOL4 program, KS is equal to the current value of the cursor, the index of the character in S before the one about to be checked.

### 3.3.1 SNOBOL4

(Refer to Figure 3.11.)

The patterns MPADPAT and SPADPAT match N<L> characters in the patterns STARS and SLASHES, respectively. MPAD and

```
*
* N= NO. OF CHARACTERS TO BE MATCHED
* J COUNTS NUMBER OF CHARACTERS THAT MATCHED
         N = ARRAY(2)
MOREN    N<1> = TRIM(INPUT)
         N<2> = TRIM(INPUT)
*
* PATTERNS TO BE USED IN PROGRAM
         MPADPAT = LEN(*N<L>) . MPAD
         SPADPAT = LEN(*N<L>) . SPAD
         STARS = '********'
         SLASHES = '////////'
         S2 = LEN(*KS) *TAB(N<L> + KS) . S3
         S4 = *LEN(KK + I) *TAB(KK + I + 1) . S5
*
*
MAINLOOP M = TRIM(INPUT)                              :F(THRU)
         OUTPUT =
         OUTPUT =
         OUTPUT = M
         S = TRIM(INPUT)                              :F(THRU)
         OUTPUT = S
         J = 0
* COUNT NO. OF CHARACTERS IN M AND S
         MCOUNT = SIZE(M)
         SCOUNT = SIZE(S)
         EQ(MCOUNT,SCOUNT)                            :F(SET)
*
* IS M-SUBSTRING EQUAL TO S-SUBSTRING?
         IDENT(M,S)                                   :F(SET)
         OUTPUT = 'R'                                 :(MAINLOOP)
*
* INITIALIZE VARIABLES
SET      L = 1
         RESULT = ''
* NEED A COPY OF M
         M1 = M
* KS POINTS TO CHARACTER BEFORE ONE TO BE MATCHED
SRESET   KS = 0
* SET MPADPAT TO A PATTERN OF N<L> STARS AND
* SPADPAT TO A PATTERN OF N<L> SLASHES
         STARS MPADPAT
         SLASHES SPADPAT
```

Figure 3.11 SNOBOL4 Program for Problem 3.

```
*
SLCCP    GT(KS + N<L>,SCOUNT)                            :S(NEWN)
* ISOLATE NEXT N<L> CHARACTERS IN S
         S  S2
* ANY SLASHES IN S-SUBSTRING?
         S3  ANY('/')                                    :S(KSINC)
* CHECK FOR A MATCH; IF SUCCESSFUL, FILL IN *'S FOR MATCHED
*   CHARACTERS IN M AND /'S FOR MATCHED CHARACTERS IN S
* K POINTS TO THE LAST MATCHED CHARACTER IN M
* KK POINTS TO THE LAST MATCHED CHARACTER IN S
         M S3 @K = MPAD                                  :F(KSINC)
         S  S3  @KK  = SPAD
* CHECK FOR ADDITIONAL CHARACTERS THAT MATCH;
*   FILL IN *'S AND /'S
         I = 0
AGAIN    S  S4
         M TAB(K + I) . HEAD  S5 = HEAD '*'              :F(CALC)
         S TAB(KK + I) . TAIL  S5 = TAIL '/'
         I = I + 1
* AT LEAST ONE MORE CHAR. IN M AND S?
         GT(K + I,MCOUNT)                                :S(CALC)
         GT(KK + I,SCOUNT)                               :F(AGAIN)
* YES, AT LEAST ONE MORE CHAR.
CALC     KS = KK + I                                     :(SLOOP)
*
NEWN     L = L + 1
         EQ(L,3)                                         :F(SRESET)
*
*
*
PREPARER M LEN(1) . TEMP =                               :F(PREOUT)
         IDENT(TEMP,'*')                                 :S(ZC)
ZA       IDENT(TEMP,' ')                                 :F(ZB)
         RESULT = RESULT ' '                             :(ZD)
ZB       RESULT = RESULT '_'                             :(ZD)
ZC       J = J + 1
         M1 LEN(1) . ANSWER
         RESULT = RESULT ANSWER
ZD       M1 LEN(1) =                                     :(PREPARER)
*
*
* REPLACE LAST CHARACTER WITH A PERIOD
PREOUT   RESULT RTAB(1) . TEMP1 LEN(1) = TEMP1 '.'
```

Figure 3.11 (cont.)

```
OUT          OUTPUT =
             OUTPUT = RESULT
* CONVERT TO REAL NUMBERS
             AJ = CONVERT(J,'REAL')
             AMCOUNT = CONVERT(MCOUNT,'REAL')
             AJJ = AJ / AMCOUNT
             OUTPUT = AJJ                                       :(MAINLOOP)
*
KSINC        KS = KS + 1                                        :(SLOOP)
*
THRU
END
```

```
DAS HAUS IST NICHT GROSS.           ←CORRECT ANSWER
DAS VATERHAUS IS VERNICHTET.        ←STUDENT'S ANSWER

DAS HAUS IS_ NICHT _____.           ←COMPUTER RESPONSE
0.6399999                           ←PERCENTAGE OF CORRECT LETTERS


MA SOEUR EST MARIEE.
MA SIR ET MARREE.

MA S___R E_T MAR_EE.
0.7500000


CETTE LECON EST DIFFICILE.
CET LECON EST DIFISEAL.

CET__ LECON EST DIF______.
0.6538461


LA JEUNE FILLE EST JOLIE.
LA JEUNE FILLE EST JOLIE.
R
```

Figure 3.11 (cont.)

SPAD are strings equal to the N<L> characters in the patterns MPADPAT and SPADPAT, respectively. The pattern STARS is used to replace matched characters in M. Similarly, SLASHES is used to replace matched characters in S.

S2 matches N<L> characters in S, beginning with the (KS+1)st character; S3 is a string equal to those N<L> characters.

After a match of N<L> characters in M has occurred, KK is set to one less than the position of the next character in S. Similarly, K is set to one less than the position of the next character in M. I indicates the number of the character past KK that is being checked for a match.

S4 is a pattern which matches the (KK+I+1)st character with a character in S. S5 is the string containing that character. If S5 is the (K+I+1)st character, a star and slash are substituted in M and S, respectively.

### 3.3.2 PL/I

(Refer to Figure 3.12.)

AA is the N(L)-length substring of S that starts in position KS. M is searched for an occurrence of AA. If there is a match, then A is set to the index of the match.

### 3.3.3 APL

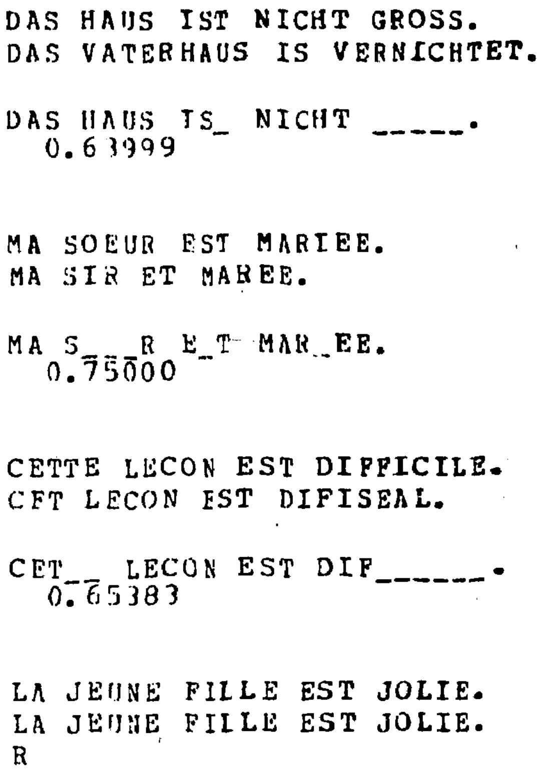
(Refer to Figure 3.13.)

```
PROBLEM:   PROCEDURE OPTIONS (MAIN);
           DCL RESULT CHARACTER (80) VARYING,
               (M,M1,S) CHARACTER (80),
               (MCOUNT,SCOUNT,N(2),KS,L,B,A,AAA) FIXED,
               AA CHAR (80) VARYING,
               JJ FIXED DECIMAL (6,5);
           ON ENDFILE(SYSIN) GO TO THRU;
/* N IS NUMBER OF CHARACTERS TO BE MATCHED */
           GET LIST ((N (L) DO L=1 TO 2));
/* J IS THE FRACTION OF MATCHED CHARACTERS PER STRING */
/* READ CHARACTERS INTO CHAR. STRING VAR.'S M AND S .*/
MAINLOOP: GET EDIT (M) (SKIP, A(80))  ;
           GET EDIT (S) (A(80));
/* PRINT THE STRINGS */
           PUT SKIP(3) EDIT (M) (A(80));
           PUT EDIT (S) (SKIP,A(80));
/* INITIALIZE NO. OF MATCHED CHARACTERS */
           J=0;
/* COUNT NO. OF CHARACTERS IN EACH ARRAY */
           MCOUNT=INDEX(M,'.');
           SCOUNT=INDEX(S,'.');
           IF MCOUNT=SCOUNT
           THEN IF M=S THEN DO;
             PUT SKIP LIST ('R');
               GO TO MAINLOOP;
           END;
/* COPY OF M  */
           M1=M;
/* INITIALIZE RESULT */
           RESULT = '';
     NEWN: DO L=1 TO 2;
/* KS IS EQUAL TO THE POSITION OF THE FIRST CHARACTER IN */
/* THE SUBSTRING OF S THAT IS BEING CHECKED              */
           KS = 1;
   SLOOP: IF KS+N(L)>SCOUNT+1 THEN GO TO NEWNEND;
/* ANY SLASHES IN S-SUBSTRING? */
/* AA IS THE N(L)-LENGTH SUBSTRING OF S, BEGINNING */
/* WITH THE CHARACTER IN POSITION KS               */
           AA=SUBSTR(S,KS,N(L));
           AAA=INDEX(AA,'/');
/* IF A SLASH, GO TO NEWKS */
           IF AAA¬=0 THEN GO TO NEWKS;
/* IS S-SUBSTRING IN M? */
/* IF SO, A IS THE INDEX OF THE FIRST OCCURRENCE OF AA */
           A=INDEX(M,AA);
```

Figure 3.12 PL/I Program for Problem 3.

```
          IF A¬=0
          THEN DO;
/* YES, S-SUBSTRING IS IN M */
              DO B=0 TO N(L)-1;
                  SUBSTR(M,A+B,1)='*';
                  SUBSTR(S,KS+B,1)='/';
              END;
/* DO ANY ADDITIONAL CHARACTERS MATCH? */
              DO I=0 BY 1
              WHILE (A+N(L)+I<=MCOUNT & KS+N(L)+I<=SCOUNT);
                  IF SUBSTR(M,A+N(L)+I,1) =
                     SUBSTR(S,KS+N(L)+I,1)
                  THEN DO;
                      SUBSTR(M,A+N(L)+I,1)='*';
                      SUBSTR(S,KS+N(L)+I,1)='/';
                  END;
                  ELSE DO;
                      KS=KS+N(L)+I;
                      GO TO SLOOP;
                  END;
              END;
          END;
/* NO, S-SUBSTRING IS NOT IN M */
          ELSE NEWKS: KS = KS+1;
          GO TO SLOOP;
NEWNEND:  END NEWN;
/* PRINT PARTIALLY MATCHED STRING */
PREPARER: DO I=1 TO MCOUNT;
          IF SUBSTR(M,I,1)='*'
          THEN DO;
              RESULT = RESULT || SUBSTR(M1,I,1);
              J = J+1;
          END;
          ELSE IF SUBSTR(M1,I,1) ¬= ' '
               THEN RESULT = RESULT || '_';
               ELSE RESULT = RESULT || ' ';
          END PREPARER;
/* MAKE SURE LAST CHARACTER IS A PERIOD */
          SUBSTR(RESULT,MCOUNT,1)='.';
          PUT EDIT (RESULT) (SKIP(2), A(80));
          JJ=J/MCOUNT;
          PUT SKIP LIST (JJ);
      OUT: GO TO MAINLOOP;
     THRU: END PROBLEM;
```

Figure 3.12  (cont.)



```
DAS HAUS IST NICHT GROSS.
DAS VATERHAUS IS VERNICHTET.

DAS HAUS IS_ NICHT _____.
  0.63999


MA SOEUR EST MARIEE.
MA SIR ET MAREE.

MA S___R E_T MAR_EE.
  0.75000


CETTE LECON EST DIFFICILE.
CET LECON EST DIFISEAL.

CET__ LECON EST DIF______.
  0.65383


LA JEUNE FILLE EST JOLIE.
LA JEUNE FILLE EST JOLIE.
R
```

Figure 3.12 (cont.)

```
    ∇ STRINGS
[1]   N←⎕
[2]  MAINLOOP:M←⍞
[3]   S←⍞
[4]   M1←M
[5]   L←1
[6]   J←0
[7]   →((ρM)≠ρS)/NEWN
[8]   I←1
[9]  ITER:→(M[I]≠S[I])/NEWN
[10]  →((I←I+1)≤ρM)/ITER
[11]  ⎕←'R'
[12]  →MAINLOOP
[13] NEWN:KS←1
[14]  RESULT←''
[15] SLOOP:→((KS+N[L])>1+ρS)/NEWNEND
[16]  D←S SUBSTR KS,N[L]
[17]  →((D⍳'/')<1+ρD)/NEWKS
[18]  HIGH←0
[19] TEST1:→((HIGH←HIGH+HIGH1←⌈/TEMP←M[HIGH↓⍳ρM]⍳D)=1+ρM)/NEWKS
[20]  SUB←(TEMP⍳HIGH1)-1
[21]  →(((HIGH-SUB)+N[L]-1)>ρM)/NEWKS
[22] TEST:→(M[(HIGH-SUB)+((⍳N[L])-1)]≠D)/TEST1
[23]  A←HIGH-SUB
[24]  B←0
[25] ALOOP:M[A+B]←'*'
[26]  S[KS+B]←'/'
[27]  →((B←B+1)<N[L])/ALOOP
[28]  I←0
[29] BLOOP:→(~((A+N[L]+I)≤ρM)∧((KS+N[L]+I)≤ρS))/OTHER
[30]  E←A+N[L]+I
[31]  F←KS+N[L]+I
[32]  →((M SUBSTR E,1)≠(S SUBSTR F,1))/OTHER
[33]  M[A+N[L]+I]←'*'
[34]  S[KS+N[L]+I]←'/'
[35]  I←I+1
[36]  →BLOOP
[37] OTHER:KS←KS+N[L]+I
[38]  →SLOOP
[39] NEWKS:KS←KS+1
[40]  →SLOOP
[41] NEWNEND:→((L←L+1)≠3)/NEWN
[42] PREPARER:I←1
```

Figure 3.13 APL Program for Problem 3.

```
[43] CLOOP:→(M[I]≠'*')/ZA
[44]  RESULT←RESULT,M1[I]
[45]  J←J+1
[46]  →INC
[47] ZA:→(M1[I]=' ')/ZB
[48]  RESULT←RESULT,'_'
[49]  →INC
[50] ZB:RESULT←RESULT,' '
[51] INC:→((I←I+1)≤ρM)/CLOOP
[52] FINAL:RESULT[ρM]←'.'
[53]  ⎕←RESULT
[54]  ⎕←JJ←J÷ρM
[55] OUT:→MAINLOOP
    ∇


      STRINGS
⎕:
      7 2


DAS HAUS IST NICHT GROSS.
DAS VATERHAUS IS VERNICHTET.
DAS HAUS IS_ NICHT _____.
0.64


MA SOEUR EST MARIEE.
MA SIR ET MARREE.
MA S___R E_T MAR_EE.
0.75


CETTE LECON EST DIFFICILE.
CET LECON EST DIFISEAL.
CET__ LECON EST DIF______.
0.6538461538


LA JEUNE FILLE EST JOLIE.
LA JEUNE FILLE EST JOLIE.
R
```

Figure 3.13 (cont.)

M is searched for an occurrence of the S-substring D. HIGH1 is set to the maximum of the indices of the occurrences in M of the letters contained in D. If HIGH, the sum of HIGH1 and the previous value of HIGH, is equal to 1 + the size of M, then one or more of the letters in D does not occur in M and the program branches to NEWKS. SUB is assigned one less than the position of HIGH1 in TEMP. The substring in M of length N(L) beginning with the character in the (HIGH-SUB)th position is compared with D. The program branches to TEST1 if the substrings do not match.

# 4 COMPARISONS AND DISCUSSION

The first part of this chapter briefly mentions some of the different features in each language: data formats, statement formats, storage allocation, input/output, and subroutine capability. Next follows a discussion of string operations. Some string operations that are primitive in one language, but not in others, are coded in the other languages.

## 4.1 DATA FORMATS

### 4.1.1 SNOBOL4

The data of SNOBOL4 include both character strings and numbers, although operations on numbers are not an important part of the language. Conversion is done automatically between numbers and strings. For example, 'ABC' 3 is equivalent to 'ABC3' and '12345' + 1 is equivalent to 12346. Patterns are built from strings by using alternation and concatenation. None of the other three languages has a pattern data type.

### 4.1.2 TRAC

In the TRAC language both instructions and data are strings. If arithmetic primitives are called, the parameters will be treated as numbers. Each instruction string is evaluated and replaced with a value string, which may itself be evaluated in turn.

### 4.1.3 APL

The data of APL are characters and numbers. A character vector, however, is a vector of single characters and not a string of characters, as is the case in SNOBOL4, PL/I, and TRAC. Arrays may be formed using characters or numbers. Conversion between numbers and characters is not done, and it is not permissible to mix the two data types.

### 4.1.4 PL/I

Data in PL/I consist mainly of fixed and floating point numbers and character and bit strings. Arrays and structures can be made from the data. Each identifier or variable is considered to have attributes which usually are specified in DECLARE statements. Strings and their maximum lengths are not declared in the other three languages, but this must be done explicitly in PL/I. Conversion is done automatically between numbers and strings.

## 4.2 STATEMENT FORMATS

### 4.2.1 SNOBOL4

All statements in SNOBOL4 are of the form

_label_ _subject_ _pattern_ = _object_ _go-to_

In various uses some of the five parts are omitted. This statement format permits pattern matching to be specified easily.

### 4.2.2 TRAC

All statements in TRAC are written as

:(FCN,p[1],p[2],...,p[k])

where FCN is a two-letter TRAC primitive and p[1],p[2],..., p[k] are arguments.

### 4.2.3 APL

There are two types of statements in APL, branch and specification. Specification statements are similar to assignment statements. Branch statements are used chiefly in function definitions.

### 4.2.4 PL/I

Unlike the other languages, there are many different statement types in PL/I. These include the DECLARE statement, assignment statement, DO statement, IF statement, input/output statements, and others.

## 4.3 STORAGE ALLOCATION

### 4.3.1 SNOBOL4

Storage allocation is done dynamically. When storage space is filled, the storage is regenerated. That is, all needed data are collected, and all data inaccessible to the SNOBOL4 program are deleted. The user is unaware of this process. Such programming techniques as building patterns in a loop use a lot of storage and should be avoided to prevent frequent storage regenerations. The user does not reserve space explicitly for variables, except for arrays.

### 4.3.2 TRAC

Storage is divided into several areas by the TRAC interpreter. User operations specified by the define string primitive are kept in a form store. The active string stack and the neutral string stack contain only the parts of the current instruction that is being evaluated.

### 4.3.3 APL

Storage reservation is done implicitly by the APL system. That is, the user does not have to declare any variables explicitly. Storage in TRAC and SNOBOL4 is also implicit. When using the APL system, the user has his own working storage, called a workspace. An active workspace has room for internal system needs, storage, and transient information. When inactive, a workspace is put in a library on secondary storage.

### 4.3.4 PL/I

Storage space for variables is allocated within begin and procedure blocks. Usually the DECLARE statement is used to reserve the space. Unlike the other languages, the maximum size of a character string must be specified in the DECLARE statement.

## 4.4 INPUT/OUTPUT

### 4.4.1 SNOBOL4

Input/output is done by "association". The variable INPUT is usually associated with the card reader, and the variable OUTPUT is usually associated with the printer. For example,

```
     TEXT = INPUT
```

assigns to TEXT the data on the next input card.

```
     OUTPUT = LINE
```

assigns to OUTPUT the information in variable LINE; numbers are automatically converted to string characters for printing.

### 4.4.2 TRAC

Input/output operations are not given special treatment; TRAC primitives handle these operations. RS reads a string from the input device and PS prints a given string.

### 4.4.3 APL

Whenever an expression or variable is typed by itself, the APL system responds by printing the value of that expression or variable.

Within function definitions, if a quad character ⎕ is written to the right of the specification arrow, the system types

⎕:

and waits for the user to type an expression. Also within function definitions, if a quad character with a quote mark inside it ⍞ is written to the right of the specification arrow, the system stops and waits for character input to be typed.

### 4.4.4 PL/I

Input/output in PL/I may be stream-oriented. Data are regarded as one continuous stream of information, not constrained to physical record sizes. GET and PUT are associated with stream input/output.

However, the user does have the choice of using record input-output. Data are organized into logical records which are treated as a whole.

## 4.5 SUBROUTINE CAPABILITY

### 4.5.1 SNOBOL4

The user may define functions by using the DEFINE function. After a function has been defined, it may be invoked the same way as built-in SNOBOL4 functions.

### 4.5.2 TRAC

New operations can be defined using DS primitives. Since there is no iteration in TRAC, recursion must be used frequently in these operation definitions.

### 4.5.3 APL

Defined functions give subroutine capability. If they have arguments and return a value, they may be defined as either a binary or a unary operator.

### 4.5.4 PL/I

PL/I permits both internal and external subroutines (procedures). Some procedures may be called as functions and return a value.

## 4.6 BASIC STRING OPERATIONS

From the many accounts that I have read ([3] and [19]), I regard the following as the most basic of all string operations:

concatenation of two strings
insertion of a substring
deletion of a substring
pattern matching or PL/I INDEX

Another operation, pattern matching with replacement, is often regarded as primitive (for example, in SNOBOL4). However, it is a combination of all the above. Pattern matching with replacement involves finding the occurrence of a substring in a string (pattern matching), replacing it with either a nonnull substring (insertion), or the null string (deletion), and then putting the string together again (concatenation).

### 4.6.1 Concatenation

Concatenation of two strings is the most basic of all string operations. Concatenation is done in SNOBOL4 by implication. For instance, consider the SNOBOL statement

subj pat1 pat2 = obj1 obj2

In this example the pattern used is the concatenation of pat1 and pat2. Similarly the object is the concatenation of obj1 and obj2.

In APL and PL/I, on the other hand, an explicit operator for concatenation is used. In APL this operator is the comma, and a restriction is imposed that characters and numbers cannot be concatenated. The symbol || joins two strings to be concatenated in PL/I, and unlike APL, automatic conversion to characters is done if a number is found.

TRAC, like SNOBOL4, does concatenation implicitly. The results of evaluating two macro calls written next to each other are concatenated. Frequently one or both of these calls returns a null value, even though side effects occur.

### 4.6.2 Insertion of a substring

Suppose it is desired to insert the word 'THE' after the tenth character of string STR. This could be done in SNOBOL4 as follows:

```
STR LEN(10) . VAR1 = VAR1 'THE'
```

The above statement replaces the first ten characters of STR, assigned to conditional variable VAR1, with VAR1 concatenated with the word 'THE'.

The same operation could be done in PL/I with the statement

```
STR = SUBSTR(STR,1,10) || 'THE' || SUBSTR(STR,11);
```

The following APL statement will do the insertion:

```
STR←(10↑STR),'THE',10↓STR
```

(See section 2.3.5 for an example of ↑ and ↓.)

The following TRAC definition for STR will do the same operation as the above three:

```
:(DS,STR,:(CN,STR,10)THE:(CS,STR))
```

### 4.6.3 Deletion of a substring

Consider the operation of deleting the eleventh through thirteenth characters of STR. The following SNOBOL4 statement will do this:

```
STR TAB(10) LEN(3) =
```

The following PL/I statement will do the deletion:

```
STR = SUBSTR(STR,1,10) || SUBSTR(STR,14);
```

In APL the operation could be done as follows:

```
STR←STR[⍳10],13↓STR
```

This operation is rather complicated when written in TRAC. Consider

```
:(DS,STR,:(CN,STR,10):(EQ,:(CN,STR,3),):(CS,STR))
```

The hard part is to move STR's form pointer ahead to the fourteenth character from the tenth without getting the characters in between. The above use of EQ does this.

### 4.6.4 Pattern matching

SNOBOL4 is really the only language of the four in which it is easy to do complicated pattern matching tasks. Consider the following task: replace the first occurrence of PAT1 in the string STR with 'THE', or if PAT1 is not present, replace the first occurrence of PAT2, or if PAT2 is not present, replace the first occurrence of PAT3.

In SNOBOL4 only one statement is needed to do this:

```
STR  PAT1 | PAT2 | PAT3  =  'THE'
```

This operation requires more statements when done in PL/I:

```
A=INDEX(STR,PAT1);
B=INDEX(STR,PAT2);
C=INDEX(STR,PAT3);
IF A=0 THEN
    IF B=0 & C¬=0 THEN
        STR=SUBSTR(STR,1,C-1) || 'THE' ||
            SUBSTR(STR,C+LENGTH(PAT3));
    ELSE IF B¬=0 THEN
        STR=SUBSTR(STR,1,B-1) || 'THE' ||
            SUBSTR(STR,C+LENGTH(PAT2));
    ELSE;
ELSE STR=SUBSTR(STR,1,A-1) || 'THE' ||
         SUBSTR(STR,A+LENGTH(PAT1));
```

APL and TRAC code for this same problem would be extremely long. The same problems in coding are shown in the following examples of pattern matching with replacement.

### 4.6.5 Pattern matching with replacement

Consider a typical pattern matching problem such as finding whether the word 'THE' is present in a sentence and if so, deleting or replacing its first occurrence in the sentence. The SNOBOL4 language is dedicated to doing just this kind of problem.

The PL/I index function, INDEX(string,substring), finds whether an occurrence of substring is present in string. INDEX returns the index of the first character of the matched portion of the string. If there is no match, a value of 0 is returned. There is no way in PL/I to indicate without an index value the success or failure of a pattern match.

With its present string primitives, PL/I cannot answer the question "Is a 'THE' present?" without also finding the position in the sentence of the first 'THE', because the INDEX function is the only way to determine whether a substring is present in a string. For example, INDEX(SENT, 'THE'). The pattern matching with replacement operation in PL/I must know the index and would be done as follows:

```
SENT = SUBSTR(SENT,1,INDEX(SENT,'THE')-1) ||
       replacement ||
       SUBSTR(SENT,INDEX(SENT,'THE')+3);
```

SNOBOL4 uses the cursor function @ to give the position of the match. For example,

```
SENT  @POSN  'THE'
```

POSN returns the index of the first 'THE' in SENT.

TRAC takes a different approach to the problem. Like SNOBOL4 and unlike PL/I, it may find whether a substring is present in a string without finding the index of this occurrence. This is done with the Yes There (YT) primitive. The same problem may also be solved using the IN primitive. In that case everything in the string up to the substring to be matched is returned as value. In either approach it is unnecessary to know the index of the match.

Pattern matching with replacement is usually done with the following sequence in TRAC: a define string primitive (DS) defines the string, the segment string (SS) lists the substring(s) to be replaced, and the call (CL) primitive calls the string with the indicated replacements.

The sequence, involving macro (string) definition and parameter calls, is inherent in the design of TRAC. If no replacement for a parameter is given in the CL operation, the null string is substituted for that parameter, thus deleting it. However, this sequence is different from the original problem because all occurrences, not just the first, are changed.

To replace just the first occurrence, other primitives must be used. For instance, consider the following. The initial (IN) function finds the first occurrence of THE. The resulting value is the portion of SENT preceding 'THE'. The form pointer now points to the first character after 'THE'. A :(CN,SENT,-3) instruction resets the form pointer to the T of 'THE'. A left pointer primitive (LP) finds the number of characters to the left of the pointer and assigns these characters to a variable LEFT with a DS primitive. An instruction with :(EQ,::(CN,SENT,3,)) would move the form pointer to the first letter after 'THE' and would give a null result. Then the value of a call segment (CS) of SENT would give the remainder of SENT.

Unfortunately APL does not have any readily available functions to solve pattern matching problems. This deficiency is the reason string problems are so difficult to code in APL. The deficiency is present because APL, which regards strings as arrays, operates uniformly on these strings. Thus string operations are done character by character; every character is treated the same. For example, using the index function,

*'HE WAS THE RIGHT ONE'⍳'THE'*

examines the string 'HE WAS THE RIGHT ONE' to find first, the character T, then the character H, and finally character E. The result is the vector 7 1 2. To allow scanning for the string 'THE', a fairly involved defined function must be used, as in Chapter 3.

The replacement problem in APL is not difficult once the substring is found. Suppose variable IND is assigned the index of the first 'THE' in string SENT and variable WORD is to be inserted in place of 'THE'. Then an instruction

```
     SENT←((IND-1)↑SENT),WORD,(IND-2)↓SENT
```

would do the necessary replacement, assuming SENT has at least one character.

4.7 OTHER STRING OPERATIONS

It is essential in doing string problems to be able to find the size of a string easily. For instance, consider scanning a string for the occurrence of several copies of a substring. It would be desirable to know the length of the substring so that when an occurrence is found, the length could be used in maintaining a cursor for the start of the next scan. SNOBOL4 has the SIZE function, PL/I the LENGTH function, and APL the size function to do this operation. Finding the length of a string is slightly harder in TRAC. The form pointer must be set to the beginning of the string by :(CR,string), and then :(RP,string) will return the number of characters to the right of the form pointer, i.e. the length of the string.

Another operation that should be readily available is comparing two strings. Usually two functions are available for this purpose - either to compare the strings for their sameness or to compare them for their difference, namely IDENT and DIFFER in SNOBOL4, = and ¬= in PL/I, and equals and not equals in APL. TRAC is different. :(EQ,X1,X2,t,f) tests X1 and X2 for equality and branches to t or f accordingly. IDENT and DIFFER are said to return values of success or failure and then a separate instruction in the go-to field indicates the branch.

Two strings must be of the same length to be compared in APL. If strings X and Y are compared using the equals or not equals operator, a vector the size of X (or Y) will be returned. This vector indicates whether the characters in the respective positions of X and Y matched. In SNOBOL4 and PL/I, if the strings are not of equal length, the shorter of the two is padded with blanks.

A lexical ordering operator is also quite useful. SNOBOL4 has LGT(X1,X2) to test whether X1 precedes X2 in the collating sequence of the machine being used. All the relational operators of PL/I may be used to compare two strings for lexical order with respect to the machine's collating sequence. :(LG,X1,X2,t,f) in TRAC tests lexical ordering depending on collating sequence and, as above, branches accordingly. APL does not consider a machine's collating sequence and thus can have no lexical ordering operator. However, a user may define his own collating sequence. For example,

```
ALPH←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
L1←'C'
L2←'D'
```

Now any relational operator may be used in place of the > in the following:

```
(ALPH⍳L1)>(ALPH⍳L2)
```

The <u>index</u> of the occurrence of L1 and L2 in ALPH serves the purpose of a lexical operator, but again only for single characters, not strings. (The decode operator can be used - see Chapter 3.)

The SUBSTR operator of PL/I turns out to be useful in the other three languages as well. To review,

```
X1 = SUBSTR(X,I1,I2)
```

assigns to X1 the I2 characters of X beginning with the I1 character. In SNOBOL4 one might use

```
X TAB(I1 - 1)  LEN(I2) . X1
```

Similarly in APL

```
X1←X[¯1+I1+⍳I2]
```

More thought is necessary to do the operation in TRAC. The following would do the SUBSTR operation in TRAC.

```
:(DS,SUBSTR,(
             :(CR,<1>)
             :(EQ,:(CN,<1>,:(SU,<2>,1)),)
             :(CN,<1>,<3>)))
```

Since it is permissible to eliminate the CL primitive, SUBSTR could be invoked by :

```
:(SUBSTR,X,:(I1),:(I2))
```

instead of

```
:(CL,SUBSTR,X,:(I1),:(I2))
```

(However, this SUBSTR function does not allow for the case where argument <3> is omitted.)

The following two tasks frequently occur in lower level assembler coding. One string handling task is take a string, define two lists of characters, and then replace the occurrences of the characters in the first list in the string by the corresponding members of the second list.

The REPLACE function in SNOBOL4 does this. Consider the following:

```
STRING1 = 'THE BEAR IS GONE'
TABLE1 = 'B A'
TABLE2 = 'D;E'
STRING2 = REPLACE(STRING1,TABLE1,TABLE2)
```

STRING2 has value 'THE;DEER;IS;GONE'.

PL/I has the built-in function TRANSLATE to accomplish this replacement. Assuming the previous definitions for STRING1, TABLE1, TABLE2, the statement

```
STRING2 = TRANSLATE(STRING1,TABLE1,TABLE2)
```

assigns to STRING2 the value 'THE;DEER:IS;GONE'.

Using segment gaps, the problem may be coded in TRAC. Procedure COMMA calls every character in its argument one at a time. After execution of COMMA, every character except the last in the argument is delimited on both sides by a comma.

```
:(DS,COMMA,(
            :(GR,:(RP,<1>),0,
            ((,)::(CC,<1>)::(CL,COMMA,<1>))),))
```

Then

```
    :(DS,TABLE1,(B A))'
    :(DS,TABLE2,(D;E))'
    :(DS,STRING1,THE BEAR IS GONE)'
    :(SS,STRING1,:(COMMA,TABLE1))'
    :(DS,STRING2,(:(STRING1:(COMMA,TABLE2))))'
    :(PS,:(STRING2))'
```

The result 'THE;DEER;IS;GONE' is printed.

The task may be done in APL with the following code.

```
     STRING1←'THE BEAR IS GONE'
     TABLE1←'B A'
     TABLE2←'D;E'
     I←1
     STRING2←STRING1
    LOOP:A←STRING2⍳TABLE1[I]
     →(A=1+⍴STRING2)/INC
     STRING2[A]←TABLE2[I]
     →LOOP
    INC:→((I←I+1)>⍴TABLE1)/OUT
     →LOOP
    OUT:→0
```

Consider the following problem in each language: find the index of the first nonblank character in a string.

The SPAN function of SNOBOL4 in the statement

```
STRING  SPAN(' ')
```

matches all blank characters in STRING up to the first non-blank. SPAN must match at least one character, or failure is indicated. A function in PL/I very similar to this in its effect is VERIFY, which in

```
VERIFY(STRING,' ')
```

returns the index of the first non-blank in STRING. It returns zero if STRING contains only blanks.

The difference between SPAN and VERIFY reflects a basic difference in SNOBOL4's and PL/I's approach to string problems. SPAN is used in a pattern matching statement; the statement is said to succeed or to fail. On the other hand VERIFY returns zero if all characters of the first string are present in the second string. Otherwise the index of the first character in the first string which is not present in the second string is returned. In SNOBOL4 the cursor operator @ may be used to find the index of success. For example, IND is assigned the index of the first nonblank in the following statement:

```
     STRING SPAN(' ') @IND
```

Notice that two operators are necessary in SNOBOL4 to perform the same function that one operator, VERIFY, does in PL/I. Thus in a sense SPAN is a more primitive operation than VERIFY. This shows a difference in the languages, namely in SNOBOL4 the index of a pattern match is separate from the pattern itself.

APL does not have a single primitive for this problem. However, the operation may be done using the following:

```
     (STRINGϵ' ')ι0
```

The operation might be done in TRAC as follows:

```
:(DS,LOOP,(
         :(EQ,::(CC,<1>), ,
              (:(DS,I,:(AD,:(I),1))
                :(GR,:(I),:(LEN),
                     (:(PS,ALL BLANKS)),
                     (:(LOOP,<1>))))))'
:(DS,I,1)'
:(DS,LEN,:(RP,STRING))'
:(LOOP,STRING)'
```

4.8 DISCUSSION

The languages exhibit strengths in different areas of string handling. Clearly, SNOBOL4 is superior for pattern matching problems. This is particularly evident in the SNOBOL4 and PL/I pattern matching problem in section 4.6.4.

The SNOBOL4 pattern data type gives great flexibility in creating and referencing patterns. In SNOBOL4 it is possible to find whether a pattern match is successful or unsuccessful without determining an index value. If a match is successful, replacement takes place; otherwise, no replacement occurs. In PL/I, however, an index value must be tested.

If a general purpose programming language is needed for a string problem, then PL/I is usually a good language to use. Its INDEX and SUBSTR primitives are very powerful. However, there are restrictions. After all, PL/I is a general purpose programming language and is not dedicated to string handling tasks. SNOBOL4, being dedicated to pattern matching problems, has its main statement form designed with this in mind. PL/I does not, so it would require a language extension to make this sort of problem easy in PL/I.

Rosin [18] has proposed modifications to PL/I to improve string handling. First, he suggests that the default for the character string type be VARYING, not FIXED. Specification of a string's maximum length would be optional. Second, he feels that the SUBSTR notation, when SUBSTR is used as a pseudo-vari able, is confusing. Instead he suggests something like X(A,I:J+I-1) in place of SUBSTR(X(A),I,J); X(I:I) for SUBSTR(X,I,1); etc. If B= 'WXYZ', I=2, J=3, then B(I:J)='XY'.

Other modifications, modeled somewhat after SNOBOL4, would make pattern matching and replacement easier. Rosin defined five new operators to be used: UPTO, BEFORE, AFTER, FROM, and IN. If X='ABCDEFG' and Y='DE', then X UPTO Y is 'ABCDE', X BEFORE Y is 'ABC', X AFTER Y is 'FG', X FROM Y is 'DEFG', Y IN X is 'DE'. Two or more of these operators may be used in the same expression. For example, X FROM Y BEFORE 'G' is 'DEF'.

Like SNOBOL4, if Y is not present in X for any of the operations, the scan fails. Any expression involving any of the five operations may be written on the left hand side of a statement; Rosin refers to this as a pseudo-expression. For example,

```
Z = 'CAT'
'A' IN Z = 'O'
```

changes Z to 'COT'.

There are disadvantages to these suggestions which Rosin himself brings up. The words UPTO, BEFORE, AFTER, FROM, and IN might have to be reserved words in PL/I, contradicting the PL/I design of no reserved words. Further, pseudo-expressions make the equals sign ambiguous. For example, consider:

```
DCL C BIT(2), D BIT(5);
D UPTO C = '1'B = '1'B;
```

In the above statements either of the two = signs could be a comparison and the other an assignment operator.

APL does have some primitives useful in string handling, but it is in need of some sort of PL/I-like INDEX function before it could be used extensively in pattern-type problems. In any sort of string operation in APL, one must not lose sight of the fact that character strings are arrays of characters. This feature in the APL design prevents good string handling, as there is no string, just an array of characters. This leads to problems when it is desired to treat a group of characters non-uniformly. For example, it would be nice to have the ability to find the index of the first occurrence of a certain word in an APL character vector. Unfortunately, with the iota operator, the character vector will be searched for the first occurrence of each letter of the word individually. A vector result will be returned, and further manipulation is necessary to get the correct answer. (See third example in Chapter 3.)

Thus, for good string handling in APL, it is necessary to be able to treat a sequence of character array elements as a string. Possibly an operator could be introduced to produce a string from a character array. Then the result could be used in string operations like those of PL/I and SNOBOL4. It would also be desirable to be able to operate on sequences of differing lengths. This would facilitate comparison of strings of differing lengths.

TRAC may indeed be useful in text editing applications when used interactively, but any real usefulness was not evident from this investigation. Any operation that needs to be done more than once must be coded to be recursive since there is no iteration operator. Errors caused by mismatching parentheses and choosing the wrong mode are hard to find. Also, TRAC makes it difficult to structure a program.

I feel that TRAC is much too difficult to learn and even when learned, is still difficult to use. Unlike PL/I, where a programmer has to know only a small subset of the language to write programs, a novice TRAC programmer must be aware of all the TRAC nuances before he can code in the language.

Faced with the problem of choosing one of the four languages for a text editing system, system-implementation questions aside, I would choose SNOBOL4. SNOBOL4 gives the ability not only to perform pattern matching easily, a necessity in text editing, but also to perform many other kinds of string operations easily. PL/I, APL, and TRAC do not have good pattern matching facilities. These three languages would of course be more useful for string handling if additional string operators were added to the language.

# BIBLIOGRAPHY

1. Abrahams, P. W. Symbol Manipulation Languages. In *Advances in Computers*, *Vol. 9*, Academic Press, New York, 1968, pp. .51-111.

2. Bates, F. and Douglas, M. L. *Programming Language/One*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

3. Bobrow, D. G. (Ed.) *Symbol Manipulation Languages and Techniques, Proceedings of the IFIP Working Conference on Symbol Manipulation Languages*. North-Holland Publishing Company, Amsterdam, 1968.

4. Bobrow, D. G. and Raphael, B. A Comparison of List-Processing Computer Languages. In *Programming Systems and Languages*, Saul Rosen (Ed.), McGraw-Hill, New York, 1967, pp. 490-509.

5. Breme, H. J. An Analysis of the TRAC Language. Report Number CC-2200, Engineering Research Center, Western Electric, Princeton, N.J., 1967.

6. Brooks, F. P. and Iverson, K. E. *Automatic Data Processing*. John Wiley and Sons, Inc., New York, 1969, pp. 343-44.

7. Caracciolo Di Forino, A. String Processing Languages and Generalized Markov Algorithms. In *Symbol Manipulation Languages and Techniques, Proceedings of the IFIP Working Conference on Symbol Manipulation Languages*, D. Bobrow (Ed.), North-Holland Publishing Company, Amsterdam, 1968, pp. 191-202.

8. Farber, D. J., Griswold, R. E., and Polonsky, I. P. SNOBOL, A String Manipulating Language. *Journal of the ACM* *11*, 1 (1964), 21-30.

9. Forte, Allan. *SNOBOL3 Primer: An Introduction to the Computer Programming Language*. MIT Press, Cambridge, Mass., 1967.

10. Griswold, R. E., Poage, J. F., and Polonsky, I. P. *The SNOBOL4 Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1968.

11. IBM Corporation. APL/360 User's Manual. Form GH 20-0683-1, 1970.

12. Katzan, H. Representation and Manipulation of Data Structures in APL. In Proceedings of a Symposium on Data Structures in Programming Languages, J. Tou and P. Wegner (Eds.), SIGPLAN Notices 6, 2 (Feb. 1971), 366-97.

13. MIT Research Laboratory of Electronics and the Computation Center. An Introduction to COMIT Programming. MIT Press, Cambridge, Mass., 1961.

14. Mooers, C. N. How Some Fundamental Problems are Handled in the Design of the TRAC Language. In Symbol Manipulation Languages and Techniques, Proceedings of the IFIP Working Conference on Symbol Manipulation Languages, D. Bobrow (Ed.), North-Holland Publishing Company, Amsterdam, 1968, pp. 178-88.

15. ________. TRAC, A Procedure-Describing Language for the Reactive Typewriter. Communications of the ACM 9, 3 (Mar. 1966), 215-19.

16. ________ and Deutsch, L. P. TRAC, A Text Handling Language. Proceedings ACM 20th National Conference, 1965, pp. 229-46.

17. Raphael, B. et al. A Brief Survey of Computer Languages and Algebraic Manipulation. In Symbol Manipulation Languages and Techniques, Proceedings of the IFIP Working Conference on Symbol Manipulation Languages, D. Bobrow (Ed.), North-Holland Publishing Company, Amsterdam, 1968, pp. 1-54.

18. Rosin, R. F. Strings in PL/I. PL/I Bulletin No. 4. Sponsored by Working Group 4 (WG4) of the Special Interest Group on Programming Languages (SIGPLAN) of the Los Angeles Chapter of the ACM, Sept. 1967, pp. 1-12.

19. Sammet, J. Programming Languages: History and Fundamentals. Prentice-Hall, Englewood Cliffs, N.J., pp. 382-470.

20. van der Poel, W. L. The Programming Language TRAC and Its Implementation. Presented at IBM Germany Computer Science Seminar, Stuttgart, Germany, Sept. 20-21, 1971.

21. Wegner, P. Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, New York, 1968, pp. 151-74.