Volume 9

KANJAP 1984

WORKING

PAPERS

in

LINGUISTICS

edited by

Letta Stranetzali

© Partial funding for this journal is provided by the Graduate Student Council from the Student Activity Fee

STUDIES IN NATIVE AMERICAN LANGUAGES III

Linguistics Graduate Student Association University of Kansas, 1984
<table>
<thead>
<tr>
<th>Articles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul Voorhis</td>
</tr>
<tr>
<td>Catawba Morphology in the Texts of Frunk Speck and of Matthews Red Thunder Cloud</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Nils R. Thompson</td>
</tr>
<tr>
<td>Lexical Representation of Salish Verb Roots: A Preliminary Examination</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>Douglas W. Isaacson</td>
</tr>
<tr>
<td>A Revised Approach to Southern Paiute Phonology</td>
</tr>
<tr>
<td>47</td>
</tr>
<tr>
<td>John E. McLaughlin</td>
</tr>
<tr>
<td>JENNY: An Interactive Program in Basic for Analyzing Comanche (and Other) Texts (With Sample Text)</td>
</tr>
<tr>
<td>81</td>
</tr>
<tr>
<td>John E. McLaughlin</td>
</tr>
<tr>
<td>Description of a Pukean Field Matrix Permutation Program</td>
</tr>
<tr>
<td>95</td>
</tr>
<tr>
<td>Danny Barrager</td>
</tr>
<tr>
<td>Eseleem Linguistic Materials</td>
</tr>
<tr>
<td>127</td>
</tr>
<tr>
<td>David L. Shaul</td>
</tr>
<tr>
<td>Katherine Turner</td>
</tr>
<tr>
<td>James Collins</td>
</tr>
<tr>
<td>Joseph P. Koss</td>
</tr>
<tr>
<td>Anita Copeland</td>
</tr>
<tr>
<td>The Structure and Function of Nootkan Baby Talk</td>
</tr>
<tr>
<td>141</td>
</tr>
<tr>
<td>Contents of Previous Volumes</td>
</tr>
<tr>
<td>169</td>
</tr>
</tbody>
</table>
JENNY: AN INTERACTIVE PROGRAM IN BASIC FOR ANALYZING COMANCHE (AND OTHER) TEXTS
[WITH SAMPLE TEXT]

John E. McLaughlin
University of Kansas

Jenny was designed to analyze and translate Comanche texts in a consistent and relatively efficient and rapid manner. While the program is nominally intended to analyze Comanche data, the modifications required to adapt it for other languages are usually very slight. Jenny was written on a Heath/Zenith H/Z-100 computer with 192K and two disk drives, although it will probably run on a system with considerably less memory available to BASIC. In a system which dynamically allocates memory (that is, all free memory is used before any house-cleaning is done), the more free memory available, the longer the program will run before the system halts for reorganization and house-cleaning. Jenny does not end at that point, but, depending on how often during the execution of the program reorganization is forced, may take one to three minutes to reorganize memory before resuming operation. I have not tested it on any computers besides my own.

Jenny is written in ZBASIC, a version of Microsoft BASIC which is designed specifically for use with the H/Z-100's 16-bit microprocessor. Certain commands may need to be modified depending on the particular dialect of BASIC which the user has available. This version of the program also requires an 80-column video monitor, so some adaptation will be required to run it on a 40-column monitor.

Purpose

One problem that every linguist with fieldnotes has encountered is the extensive time required to prepare texts for publication. Unfortunately, the usual solution is to publish the texts with only an English translation. While this practice does get the material into the public’s hands, the linguistic use of the texts is limited to those who already possess a knowledge of the language.

Jenny provides a means by which texts can more rapidly be analyzed and edited for publication. It also provides a means by which glosses and underlying forms can be kept consistent throughout a given body of material without constant cross-checking. In addition, it formats the texts in a layout which, with only minimal work on a word-processor, can be used for publication, providing more information for the reader than simply an English translation of the raw text would provide.

Kansas Working Papers in Linguistics, 1984, Volume 9, Pages 81-93
Using the Program

Using Jenny requires some preparation before using. The preparation consists of typing the text(s) to be analyzed onto a disk file. Any word-processor can be used to accomplish this, but it is important to remember several things about the BASIC language when doing this. First, Jenny reads a line at a time from the text for analysis. BASIC can read up to 255 characters as a line, so it is important to break up the text into sentences with a carriage return at the end of each one. Some word-processors (e.g., PeachText) display the lines as broken on the screen, but in fact, the only carriage return is at the end of the paragraph. These word-processors usually display some sort of special mark where the carriage returns actually occur, so it is important to physically end each natural sentence in the text with a real carriage return. Second, BASIC recognizes three characters as end-of-line indicators, at which it stops reading. These three characters are the carriage return, the comma, and the quotation mark. If the text to be analyzed contains commas or quotation marks, they must be replaced before running the text through Jenny or else BASIC will treat them exactly as a carriage return. In the Comanche texts which I have been using Jenny with, I have eliminated all the commas and replaced the quotation marks with a double hyphen "--". Using a semi-colon for a comma is also a useful way to bypass BASIC's recognition of a comma as an end-of-line character.

Once the text has been typed onto a disk file in the correct format, the file should be named "RAWTEXT.DOC". On a separate disk, two files should be named, although it is not necessary to put anything in them before using Jenny. These two files are "SURFACE.DOC" and "MORPHS.DOC". Jenny will also open a file named "NEWTEXT.DOC" on the same disk as "RAWTEXT.DOC" during its operation, so there should be sufficient room on the disk for both files ("NEWTEXT.DOC" will be approximately five times as large as "RAWTEXT.DOC" if all of the text is analyzed at one session).

After loading Jenny into memory and typing "RUN", the screen clears and the user is instructed to insert the dictionary disk (the one on which the files "MORPHS.DOC" and "SURFACE.DOC" occur) and the text disk (the one on which the file "RAWTEXT.DOC" occurs) into drives A and B respectively. Jenny then takes a few moments to load the two arrays containing surface forms and underlying forms and to open the file "RAWTEXT.DOC" for input and the file "NEWTEXT.DOC" for output (at this point, anything that had previously existed in "NEWTEXT.DOC" will be erased so it is important to rename "NEWTEXT.DOC" after each session). Jenny will then read the first sentence of the document and display it at the top of the screen along with the first word of the sentence. It then searches the surface array for the longest string that matches the first characters in the first word. Once it finds a string
that matches, it displays the string that it matched on the right side of the screen. Beneath the word being analyzed, Jenny then displays the underlying form which corresponds to the surface string with its gloss and asks the user if it is the correct underlying form or not. The user has three choices at this point—hitting any key except "A" or "N" accepts the form, hitting "N" indicates that the underlying form shown is not the correct one, and hitting "A" indicates that the underlying form is correct, but needs to be amended. Answering "N" will cause Jenny to search for the next longest matching string at which point it will again query the user as to whether to accept, reject, or amend the form found.

If no form matches the string, Jenny will ask whether the user wishes to add a form to the dictionary. If the response is anything other than "N", Jenny will ask for the surface string and the underlying form. Jenny will again search the MORPHSS array to determine if this underlying form is already in the dictionary. If it is, Jenny will ask if it is the correct underlying form. If it is, the surface form will be cross-indexed to the underlying form in the MORPHSS array. If it is not, Jenny will continue to search the MORPHSS array until another matching form is found at which point it again queries the user whether to accept it or not. If no correct underlying form is found in the MORPHSS array, Jenny will ask the user for the underlying form (again) and a gloss. These are then filed in the Morphs array if correct.

When adding new forms to the SURFACES and MORPHSS arrays, Jenny tells the user how much of the array has been used. If the number approaches the size of the variables SIZE and MORSIZE in lines 230 and 240, the size of these variables should be increased before running the program the next time. This will increase the size of the SURFACES and MORPHSS arrays.

As each word is analyzed, the morpheme-by-morpheme gloss will be printed below the word. When a word has been completely analyzed, the next word will be displayed and the process will begin again. After the sentence has been finished, the screen will clear and the sentence will be printed at the top of the screen. Below, in formatted structure, the morpheme-by-morpheme glossed sentence will be displayed. Finally, the user will be asked to provide a free translation of the sentence. The same rules apply for this translation as applied for editing "RAWTEXT.DOC", that is, no commas, quotation marks, or carriage returns (at least not in the middle of the sentence). Jenny will then ask if the user wishes to continue. If the answer is anything but "N", the next sentence will be analyzed. If the answer is "N", Jenny will then close the text files "RAWTEXT.DOC" and "NEWTEXT.DOC" and sort the SURFACES array in descending order from longest string to shortest (this takes a little while).
The file "NEWTEXT.DOC" should now be renamed. The text is virtually ready for print although it will be necessary to add titles, and commas and quotation marks if necessary.

It should be noted that while the file "SURFACE.DOC" may be entered and tampered with, the file "MORPHS.DOC" should not be edited except through the procedures provided in the program. The reason for this is that the SURFACE$ array consists of a surface string and a number. The number refers to a particular position in the MORPHS$ array. If forms are rearranged, deleted, or added in the middle of the file "MORPHS.DOC", the numbers for all following entries will be wrong and the SURFACE$ array will be referring to incorrect forms.

Program Listing

200 CLS
210 OPTION BASE 1
220 DEFINT A-Z
230 SIZE=1000
240 MORPHS$=1000
250 LITSIZE=15
260 LINESIZE=72
270 SPACES=STRING$(60,"
")
280 DIM
SURFACES$(MORPHS$,2),UNDER$(SIZE,2),MORPHS$(LITSIZE),GLOBS$(LIT
SIZE),GENTS$(15),GENTS$(15)
290 LOCATE 10,1:PRINT "INSERT DICTIONARY DISK IN DRIVE A (Press any key to continue)"
300 GOSUB 2500
310 LOCATE 15,1:PRINT "INSERT TEXT DISK IN DRIVE B (Press any key to continue)"
320 GOSUB 2500
330 OPEN "+",#1,"a:surface.doc"
340 OPEN "+",#2,"a:morphs.doc"
350 I=1
360 WHILE I<MORPHS$ AND NOT EOF(1)
370 FOR J=1 TO 2
380 INPUT #1,SURFACES$(I,J)
390 NEXT J
400 I=I+1
410 WEND
420 I=1
430 WHILE I<SIZE AND NOT EOF(2)
440 INPUT #2,A
450 FOR J=1 TO 2
460 INPUT #2,UNDER$(I,J)
470 NEXT J
480 I=I+1
490 WEND
500 GOSUB
510 OPEN "I",#1,″b:rawtext.doc″
520 OPEN "O",#2,″newtext.doc″
530 CLS
540 INPUT #1,SENTS
550 FOR J=1 TO 15
560 MENTS(J)=""
570 NEXT J
580 NEXT I
590 Q=1
600 R=0
610 X=1
620 Y=LEN(SENTS)
630 R=R+(\"\\LINELINESIZE\")
640 IF Y MOD LINELINESIZE>0 THEN R=R+1
650 IF (MID$(SENTS,X,1)<="A" OR MID$(SENTS,X,1)<="Z") AND
      (MID$(SENTS,X,1)<="a" OR MID$(SENTS,X,1)<="z") AND X<Y AND THEN
       X=X+1:GCP 650
660 IF X<Y THEN 20
670 FIRST=2
680 WHILE MID$(SENTS,X,1)<="A" AND MID$(SENTS,X,1)<="z" OR
      MID$(SENTS,X,1)<="A" AND MID$(SENTS,X,1)<="z" OR
      MID$(SENTS,X,1)<="\"
690 X=X+1
700 WEND
710 WORDS=MID$(SENTS,FIRST,X-FIRST)
720 LOCATE 1,1:PRINT SENTS
730 LOCATE 5,1:PRINT SPACE$ 
740 LOCATE 5,1:PRINT WORDS 
750 LOCATE 7,1:PRINT SPACE$ 
760 PRINT SPACE$ 
770 GOSUB 2520 
780 XWORDS=Words$ 
790 A=1 
800 M=1 
810 G=1 
820 I=0 
830 J=1
840 WHILE SURFACES$(5,1)<LEFT$(XWORDS$,LEN(SURFACES$(5,1))) AND 
      LEN$(SURFACES$(5,1))>0
850 S=S+1
860 WEND
870 IF LEN(SURFACES$(5,1))=0 THEN 1350 
880 W=VAL(SURFACES$(5,2)) 
890 IF LEN(UNIONS$(W,1))=0 THEN 1650 
900 LOCATE 5,41:PRINT STRINGS$(40," ) 
910 LOCATE 5,41:PRINT SURFACES$(5,1)
LOCATE 11,1:PRINT "Surface form: ";SURFACES$(Z,1)
1120 LOCATE 11,1:PRINT "Underlying form: ";UNDERS$(W,1)
1130 LOCATE 11,1:INPUT "Surface form: ";SURFACES$(Z,1)
1140 LOCATE 11,1:PRINT SPACES
1150 LOCATE 11,1:PRINT SPACES
1160 GOSUB 2550
1170 IF AS="n" OR AS="N" THEN 1180 ELSE 1190
1180 LOCATE 11,1:PRINT SPACES
1190 LOCATE 11,1:INPUT "Underlying form: ";UNDERS$(W,1)
1200 LOCATE 13,1:PRINT SPACES
1210 LOCATE 13,1:PRINT "Gloss: ";UNDERS$(W,2)
1220 GOSUB 2550
1230 IF AS="n" OR AS="N" THEN 1240 ELSE 1250
1240 LOCATE 13,1:PRINT SPACES
1250 LOCATE 13,1:INPUT "Gloss: ";UNDERS$(W,2)
1260 MORPHS$$(A)=UNDERS$(W,1)
1270 GLOSS$(A)=UNDERS$(W,2)
1280 LOCATE 7,1:PRINT MORPHS$$(A)+"-
1290 LOCATE 8,1:PRINT GLOSS$$(A)+"-
1300 n=LEN(MORPHS$$(A))+1
1310 G=LEN(GLOSS$$(A))+1
1320 A=A+1
1330 XWORD$=XWORDS1+LEN(SURFACES$(Z,1))
1340 IF LEFT$(XWORD$,1)="" THEN 1740 ELSE 920
1350 LOCATE 25,1:PRINT SPACES
1360 LOCATE 25,1:PRINT "Do you wish to add a form? (y/n) ";Y
1370 GOSUB 2550
1380 IF A<="a" AND A<>"n" THEN 1390 ELSE 1740
1390 I=1

1400 WHILE LEN(SURFACES$(I, 1))>0
1410 I=I+1
1420 MEND
1430 LOCATE 23, 1: PRINT SPACE$ 
1440 LOCATE 23, 1: PRINT "Length of surface array is": I
1450 GOSUB 2520
1460 LOCATE 11, 1: INPUT "What is the surface form": SURFACES$(I, 1)
1470 LOCATE 13, 1: INPUT "What is the underlying form": SURFACES$(I, 2)
1480 GOSUB 2550
1490 IF A$="N" OR A$="n" THEN 1450
1500 C=C+1
1510 WHILE LEN(UNDER$(C, 1))>0 AND UNDER$(C, 1)>SURFACES$(I, 2)
1520 C=C+1
1530 MEND
1540 IF LEN(UNDER$(C, 1))>0 THEN 1550 ELSE 1610
1550 LOCATE 11, 1: PRINT SPACE$
1560 LOCATE 11, 1: PRINT UNDER$(C, 1)
1570 LOCATE 13, 1: PRINT SPACE$
1580 LOCATE 13, 1: PRINT UNDER$(C, 2)
1590 GOSUB 2550
1600 IF A$="n" OR A$="N" THEN C=C+1: GOTO 1510
1610 SURFACES$(I, 2)=STR$(C)
1620 GOSUB 2520
1630 I=I+1
1640 GOTO 880
1650 GOSUB 2520
1660 LOCATE 11, 1: INPUT "What is the underlying form": UNDER$(W, 1)
1670 LOCATE 13, 1: INPUT "What is the gloss": UNDER$(W, 2)
1680 GOSUB 2550
1690 IF A$="N" OR A$="n" THEN 1650 ELSE 1700
1700 GOSUB 2520
1710 LOCATE 23, 1: PRINT SPACE$
1720 LOCATE 23, 1: PRINT "Length of underlying array is ": VAL(SURFACES$(I, 2))
1730 GOTO 1260
1740 MSENTS$=""
1750 GSENTS$=""
1760 I=1
1770 WHILE LEN(MORPHS$(I))>0
1780 MSENTS$=MSENTS$+MORPHS$(I)=""
1790 GSENTS$=GSENTS$+Glosses$(I)=""
1800 I=I+1
1810 MEND
1820 MSENTS$=LEFT$(MSENTS$, LEN(MSENTS$)-1)
1830 GSENTS$=LEFT$(GSENTS$, LEN(GSENTS$)-1)
1840 IF LEN(MSENTS$)>LEN(GSENTS$) THEN L=LEN(MSENTS$) ELSE 1880
1850 A$L=L: GSENTS$=GSENTS$+" A,"
1870 GOTO 1930
1880 IF LEN(GSENT$) = LEN(MSENT$) THEN L = LEN(GSENT$): ELSE 1920
1890 AL = L - LEN(MSENT$)
1900 MSENT$ = MSENT$ + STP$NINGS(AL, " ")
1910 GOTO 1930
1920 I = 1 + LEN(MSENT$)
1930 IF LEN(MSENT$) + Q + LINESIZE THEN Q = Q + 1
1940 MSENT$ = MSENT$ + GSENT$ + MSENT$
1950 GSENT$ = GSENT$ + GSENT$
1960 AL = AL + L
1970 FOR I = 1 TO LINESIZE
1980 MORGSS%$(I) = ""  
1990 CLOSE$(I) = ""
2000 NEXT I
2010 IF X = Y THEN 650
2020 CLS
2030 LOCATE 1, 1: PRINT SENTS$
2040 LOCATE 5, 1
2050 FOR I = 1 TO 15
2060 IF MENTS$(I) = "" THEN 2070 ELSE 2100
2070 PRINT MENTS$(I)
2080 PRINT GSENT$(I)
2090 PRINT
2100 NEXT I
2110 LOCATE 21, 1: PRINT "Translation": TRANS$ R = R + (LEN(TRANS$) - 
2120 IF LEN(TRANS$) MOD LINESIZE > 0 THEN R = R + 1
2130 R = R + Q
2140 PRINT "$", ""
2150 PRINT "$", ""
2160 IF $ = "$", " " THEN 2170 ELSE 2170
2170 PRINT "$", SENTS$
2180 FOR I = 1 TO 15
2190 IF MENTS$(I) = "" THEN PRINT "$", MENTS$(I): PRINT "$", GENTS$(I)
2200 NEXT I
2210 PRINT "$", TRANS$
2220 IF NOT EOF($) THEN 2230 ELSE 2260
2230 LOCATE 25, 1: PRINT "Do you wish to continue? (y/n) <Y>"
2240 GOSUB 2500
2250 IF A$ = "Y" OR A$ = "n" THEN 2260 ELSE 330
2260 CLOSE
2270 CLS
2280 DIM TEMP(MORSIZE)
2290 FOR I = 1 TO MORSIZE
2300 IF LEN(SURFACES$(I, 1)) > 0 THEN 2310 ELSE 2380
2310 C = I
2320 PRINT SURFACES$(I, 1) = " "
2330 PRINT "$", MORSIZE
2340 IF LEN(SURFACES$(J, 1)) > 0 THEN IP SURFACES$(I, 1) = SURFACES$(J, 1)
THEN C=C+1
2350 NEXT J
2360 IF TEMP(C)>0 THEN C=C+1: GOTO 2360
2370 TEMP(C)=1
2380 NEXT I
2390 OPEN "+o",#1,"asurface.doc"
2400 OPEN "+o",#2,"asmorphs.doc"
2410 FOR I=1 TO MORSIZE
2420 IF TEMP(I)<MORSIZE AND TEMP(I)>0 THEN PRINT #1,"SURFACES$(TEMP(I),1)" ","SURFACES$(TEMP(I),2)
2430 NEXT I
2440 FOR I=1 TO SIZE
2450 IF LEN(UNDER$(I,1))>0 THEN PRINT "+2,1"
"UNDER$(I,1)" ","UNDER$(I,2)
2460 NEXT I
2470 CLOSE
2480 CSL
2490 END
2500 AS=INKEYS
2510 IF AS="**" THEN 2500 ELSE RETURN
2520 LOCATE 11,1:PRINT SPACE
2530 LOCATE 13,1:PRINT SPACE
2540 RETURN
2550 LOCATE 25,1:PRINT SPACE
2560 LOCATE 25,1:PRINT "ACCEPT (y/n): <y>"
2570 GOSUB 2500
2580 RETURN

Variables

AS: Responses to queries
BI: Variable used to force reorganization of memory
FIRST$: Marks the beginning of a word in SENT$,
GLOSSES$: Array of glosses for each word
GSENT$: Array of the glosses for each sentence
LINE$: Temporary storage of glosses for a sentence
LITSIZE$: Length of line usu
MORPH$: Array of morphemes for each word
MORSIZE$: Length of SURFACES array
MSSENT$: Array of the morphemes for each sentence
MSENT$: Temporary storage of morphemes for a sentence
SENT$: The sentence being analyzed
SIZE$: Length of UNDER$ array
SPACE$: String of spaces used to partially clear screen
SURFACES$: Array of surface strings and numbers referring to
UNDER$ array
Below is the text as typed into the file "RAWTEXT.DOC". Immediately following the original text is the text after analysis by Jenny (LINESIZE was reduced to 60 for this example). (The text here is a previously unpublished text in Canonge's fieldnotes.)

1 su'anakYse' wasape' tojaboko'o'a bomaninA.
2 ukYhi u rykJynikukYse' oka'ahnakaty uwaka bitynU.
3 sitykYse' --jew. tanihcu' hina yny hatbyn17-- meky.
4 --ponamy ny'-- meky.
5 sity oha'ahnakatykYse' --hakYse' y pomapy7-- meky.
6 --jew. nyca' ny'ebctu ma bomanin'ary-- mekyo'se' sit y u nilwijiU. --ma cahku'etY nah ma ywi'eeju. Ry ryree'tyky
bityhc1 wihu naq mawyhto'leeu.--
7 sitykYse' soo ma jywihc1 su'anety biawoo'etY u wyhto'suwa'eeju.
8 --wasape'. ny buha'al ny sutaahc1-- meky.
9 sitykYse' --ke yny caa hina hanintY-- meky.

\ifstream c .np
1 su'anakYse' wasape' tojaboko'o'a bomaninH.
2 su'ag'na-ky-se' wasampe' toja pokoko'a
3 there-LOC-EVID-PRT bear mtn berry-CBJ
4 pomah-nil-na
5 pick-around-COMT
6 Somewhere there bear was picking mountain berries.

\ifstream 6 .np
2 ukYhi u rykJynikukYse' oka'ahnakaty uwaka bitynU.
3 u=kah1 u
tykka'r-hpyni-ku-ky-se' oha'ahnakentyn
4 dem posy 'in CBJ eat-FREQ-DS SUB-EVID-PRT coyote
5 uwaka pityG-nuh
6 DEM-POST arrive compl
7 As he was eating some of them, coyote came up to him.
This one said: 'Oh brother. What are you doing?'

This coyote said: 'Where are your berries?'

This one said to him: 'Oh. I go picking them inside me. Breaking them off; I just swallow them. Arriving among my children; I then just omit them up.'
There are several things in Jenny which can be changed to suit individual needs. Of course, the various file names can be changed depending upon the requirements of the particular word processor in use. I use Peachtex, therefore text files have the extension "DOC". Also, the Peachtex formatting command found in line 2160 can be changed to suit a different word processor. In Peachtex, the command "\iflinesc<np> means "if there are less than six lines left at the bottom of the page, begin a new page".

In Comanche, I have used y instead of i and j instead of y. I have also used t for glottal stop and a capital for voiceless vowels. Line 650 lists the various ASCII characters used for a particular language, in this case all small and capital letters and \t. This line can be changed to accommodate any character set in use for a particular language.
If necessary or desired, the restrictions on punctuation required for the original text can be avoided with the use of LINE INPUT rather than INPUT in line 540. The same restrictions on the English translation can be avoided by the use of LINE INPUT in line 2110.

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPL</td>
<td>completive</td>
</tr>
<tr>
<td>CONT</td>
<td>continuous</td>
</tr>
<tr>
<td>DEM</td>
<td>demonstrative</td>
</tr>
<tr>
<td>DIM</td>
<td>diminutive</td>
</tr>
<tr>
<td>DS</td>
<td>different subject</td>
</tr>
<tr>
<td>EVID</td>
<td>evidential</td>
</tr>
<tr>
<td>FREQ</td>
<td>frequentative</td>
</tr>
<tr>
<td>HAB</td>
<td>habitual</td>
</tr>
<tr>
<td>INDEF</td>
<td>indefinite</td>
</tr>
<tr>
<td>INSTR</td>
<td>instrumental</td>
</tr>
<tr>
<td>INTER</td>
<td>interrogative</td>
</tr>
<tr>
<td>ITER</td>
<td>iterative</td>
</tr>
<tr>
<td>LOC</td>
<td>locative</td>
</tr>
<tr>
<td>NEG</td>
<td>negative</td>
</tr>
<tr>
<td>NOM</td>
<td>nominalizer</td>
</tr>
<tr>
<td>OBJ</td>
<td>object</td>
</tr>
<tr>
<td>OBL</td>
<td>oblique</td>
</tr>
<tr>
<td>PL</td>
<td>plural</td>
</tr>
<tr>
<td>POS</td>
<td>possessive</td>
</tr>
<tr>
<td>POST</td>
<td>postposition</td>
</tr>
<tr>
<td>PROG</td>
<td>progressive</td>
</tr>
<tr>
<td>PRT</td>
<td>particle</td>
</tr>
<tr>
<td>QUOT</td>
<td>quotative</td>
</tr>
<tr>
<td>SB3</td>
<td>subject</td>
</tr>
<tr>
<td>SG</td>
<td>singular</td>
</tr>
<tr>
<td>SS</td>
<td>same subject</td>
</tr>
<tr>
<td>SUB</td>
<td>subordinating suffix</td>
</tr>
<tr>
<td>1s</td>
<td>first person singular</td>
</tr>
<tr>
<td>2s</td>
<td>second person singular</td>
</tr>
<tr>
<td>3s</td>
<td>third person singular</td>
</tr>
</tbody>
</table>
Volume 9

KANSAP

WORKING

PAPERS

in

LINGUISTICS

edited by

Lotta Strantzali

© Partial funding for this journal is provided by the Graduate Student Council from the Student Activity Fee

STUDIES

IN

NATIVE

AMERICAN

LANGUAGES

III

Linguistics Graduate Student Association
University of Kansas, 1984
<table>
<thead>
<tr>
<th>Articles</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul Voorhis</td>
<td>Catawba Morphology in the Texts of Frank Speck and of Matthew's Red Thunder Cloud</td>
<td>1</td>
</tr>
<tr>
<td>Nile R. Thompson</td>
<td>Lexical Representation of Salish Verb Roots: A Preliminary Examination</td>
<td>31</td>
</tr>
<tr>
<td>Douglas W. Issacson</td>
<td></td>
<td></td>
</tr>
<tr>
<td>John E. McLaughlin</td>
<td>A Revised Approach to Southern Paiute Phonology</td>
<td>47</td>
</tr>
<tr>
<td>John E. McLaughlin</td>
<td>JEBGY: An Interactive Program in Basic for Analyzing Comanche (and Other) Texts (With Sample Text)</td>
<td>81</td>
</tr>
<tr>
<td>Danna Barrager</td>
<td>Description of a Pikean Field Matrix Permutation Program</td>
<td>95</td>
</tr>
<tr>
<td>David L. Sheaul</td>
<td>Kselen Linguistic Materials</td>
<td>127</td>
</tr>
<tr>
<td>Katherine Turner</td>
<td></td>
<td></td>
</tr>
<tr>
<td>James Collins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joseph F. Keen</td>
<td>The Structure and Function of Nootkan Baby Talk</td>
<td>141</td>
</tr>
<tr>
<td>Anita Copeland</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contents of Previous Volumes</td>
<td></td>
<td>165</td>
</tr>
</tbody>
</table>
DESCRIPTION OF A PIKEAN FIELD MATRIX PERMUTATION PROGRAM

Dana Barrager

Abstract: This paper presents a program which may be used to perform Pikean field matrix permutations and other matrix operations. The use of the program is also demonstrated. This program can be a useful tool for analyzing data from highly inflected languages.

Pike (1963) suggested a matrix treatment of formatives as a tool for analyzing morphological data. His technique involves compiling the data in a matrix format and permuting the matrix to other matrices. These matrices may be analyzed and information abstracted from them. Intuitively, this is a very appealing approach. Permuting the matrices is, however, a very tedious process, and it is doubtful that a human could produce very useful results performing these manipulations by hand. Also, hand permutation tends to propagate errors in the matrices.

With the advent of computer availability to researchers, fortunately, these problems are easily solved. Computers are perfectly suited for manipulating bodies of data, such as matrices. Pikean field matrix permutations may be easily performed using a computer program. This paper describes one such program. This program is a development of another program presented in Miner and Taghva (1983). Readers should refer to the earlier paper for a more detailed discussion of how to interpret particular matrix configurations and why these matrix permutations are useful. The program presented in Miner and Taghva is not very powerful. The program presented here is more powerful. In particular, this program uses input and output files so that the matrices may be stored as computer files and do not have to be typed in every time the program is run. In addition, the individual cells may be much larger, and there are several new commands which the earlier program did not have. This program is presented as a tool linguists may wish to use to assist in analyzing morphological data. The savings in time and frustration are proportional to the complexity of the data. Data from highly inflected languages, therefore, are very good candidates for input to this program.

One of the greatest difficulties of matrix analysis is entering the data correctly. The program makes this much easier by reading the data from a file in the computer. Once the data is entered, it may be inspected, modified, and verified. After this process is complete, the data is ready to be used with the program. The data need never be entered again. It may be manipulated, and sections of the matrix may be

Kansas Working Papers in Linguistics, 1984, Volume 9, Pages 95-126
removed, and the resulting matrices may be stored on other files which can also be used as input to the program. The use of input and output files makes this a very useful tool in morphological analysis of highly inflected languages.

The program provides other useful features which will be described in the subsequent pages. These features are summarized here. The data is entered as "cells" of the matrix. A cell of the matrix is an element which may be referenced by one row name and one column name. For example, the table on the following page gives a subset of the Chukchi verb paradigms, provided by Scott R. Krause (1983). This data will be used for examples in this paper. In this matrix, the cell referenced by row name 3g and column name 1pl is the unit "mak". The coding scheme is given after the data. The program manipulates rows or columns of cells, and in some cases, blocks of cells. Rows or columns may be moved, merged into other rows or columns, or removed from the matrix.

The matrix is presented here exactly as it is stored in the computer file which is used as input to the program. The numbers at the top of the matrix provide formatting information for the program which will be explained shortly.

The cells may vary in size from one row per cell to five rows per cell, and from one column per cell to ten columns per cell. This permits a broad range of data to be handled. The matrix itself has a limit of 50 rows and 150 columns. This seems like a reasonable size for manipulating language data, but this size may be expanded if necessary by changing constants in the program and recompiling the program. If very large matrices are used, a problem exists in displaying the data on a standard Cathode Ray Tube (CRT) terminal. Normally only twenty-four rows and eighty columns may be displayed. There are two solutions to this problem: use a terminal which may display more rows or columns, or use the output files the program provides and have the output files printed on a line printer. Either of these methods will permit larger matrices to be examined.

The program provides a degree of assistance in using it. The Inventory procedure, for example, will give the user a list of commands which may be used, if "I" or "i" is entered in response to the COMMAND prompt. Also, most of the commands request specific information which is required for processing the command. This means that complex command syntax need not be memorized. Since the commands are all single letters, and there are relatively few commands, memorizing them should not be a problem.
<table>
<thead>
<tr>
<th></th>
<th>1sg</th>
<th>2sg</th>
<th>3sg</th>
<th>1pl</th>
<th>2pl</th>
<th>3pl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1sg</td>
<td>-gat</td>
<td>-n-an</td>
<td>-n/N</td>
<td>-N-an-t</td>
<td>-N-an-t</td>
<td></td>
</tr>
<tr>
<td>2sg</td>
<td>-g?e</td>
<td>-N-an</td>
<td>-tlen</td>
<td>-g?e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3sg</td>
<td>-g?e</td>
<td>-g/Na</td>
<td>-ni-n</td>
<td>-mak</td>
<td>-n/Na</td>
<td>-tak</td>
</tr>
<tr>
<td>1pl</td>
<td>-gat</td>
<td>-N-an</td>
<td>-mak</td>
<td>-n/Na</td>
<td>-tak</td>
<td>-N-an-t</td>
</tr>
<tr>
<td>2pl</td>
<td>-n/Na</td>
<td>-Na-tak</td>
<td>-tku-n</td>
<td>-Na-tka</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3pl</td>
<td>-gam</td>
<td>-g/Na</td>
<td>-mak</td>
<td>-n/Na</td>
<td>-tak</td>
<td>-N-an-t</td>
</tr>
</tbody>
</table>

Symbol | IPA Equivalent
---|---
a | ø
? | ?
g | Y
N | ø

Chukchi Verb Paradigm and Coding Scheme

The program is written in Pascal programming language, which has nice features, such as advanced file-handling capability, for this application. The program is standard Pascal with the exception of the ELSE clause in the CASE statement in the main program. Many compilers support the ELSE in a CASE statement, but if not, this line may be left out and the program will still run. However, without this line, execution will terminate if an illegal command is entered. There are ways of preventing this problem without the use of the ELSE in the CASE statement, such as using nested IF-THEN-ELSE statements in place of the CASE statement.

The program was written to make modification as easy as possible. The program is, due to the broad range of matrices it will accept, rather complex. However, the program is divided up into a large number
of procedures which perform specialized functions. This means that the program may be customized for specific applications with comparative ease. Interior documentation is provided to assist the user in figuring out how the program works. The program modularity also permits new features to be added using the procedures which already exist. The main program consists largely of a series of procedure calls, built around a list of commands. New commands may be inserted into the list relatively easily.

There are several commands which are not implemented in the program which might be useful. Some ideas have been: a command to transpose the matrix and a command to remove submatrices in a single step. A particularly interesting suggestion is to add another display mode. Rather than printing the entire matrix every time the display command is entered, only as much of the matrix as could be displayed would be printed. This would provide a "window" for examining the matrix which could be moved around to view different parts of the matrix. This would make examining large matrices considerably easier.

Another command which would be very useful, but would be rather difficult to implement, is a command to automatically perform permutations and produce the "best" matrix according to some pattern recognition scheme. The difficulty with this lies in the problem of defining an algorithm which can identify desirable matrix configurations. It is an interesting problem, however, which merits further investigation.

Using the Matrix Permutation Program

This section illustrates the use of the matrix permutation program. The information presented in this section is independent of the system the program is implemented on.

1. The Data File: The distinction between logical and physical rows and columns is vital in understanding how to use this program. A physical row consists of one line of the matrix, whereas a logical row may refer to one, or several, physical rows. A logical row can be referenced by a name, as in the following examples: lag refers to the physical row on which the letters "lag" occur, the next row, which is a continuation of the first, and a blank row. In this matrix, all logical rows consist of three physical rows.

The same distinction applies to logical and physical columns. In this matrix there are nine physical columns per logical column. A "cell" is a space which may be referenced by one logical row and column name, in this matrix a three by nine submatrix. The number of physical columns or rows per logical row or column is determined by the user, who should base this decision upon the data. In this program, up to five physical rows per logical row
and up to ten physical columns per logical column are permitted. One cell may therefore contain up to 50 characters. Usually, however, a blank physical row and a blank physical column between logical rows and columns is desirable to avoid confusion. This means that usually a maximum of 40 out of a possible 50 character positions in a cell will be filled. The program will permit 50 total physical rows and 150 total physical columns. These limitations may be changed quite easily by changing constants in the program before compiling the program.

When creating data files, the blank spaces must be kept in mind. The program will not automatically generate blank spaces between cells; they must be incorporated into the data file itself. The data file which this program runs on is called by default "datafile". When the program is run, it looks for a file called "datafile", if no alternate data file is specified. An alternate data file may be specified in some system-dependent manner. A data file, in order to function properly with this program, must consist of the following: the first non-blank line of the file should list, in this order: 1) the number of physical rows per logical row 2) the number of physical columns per logical column 3) the total number of physical rows in the matrix, including blank rows 4) the total number of physical columns in the matrix, including blank columns. The next line should begin the listing of the data. It is suggested that one logical row and column be reserved for headings. This is not necessary, but helpful.

After the data file has been prepared, you are ready to manipulate the data. The rest of this section explains the use of the commands within the program. Henceforth, all uses of the terms row and column refer to logical rows and columns unless explicitly stated otherwise. All of the commands described here refer to logical rows or columns. A row or column is referenced simply by typing the first element in that row or column. That is why headings are not strictly necessary.

2. Inventory--I: The commands all consist of one letter, which may be upper- or lower-case. All commands are designed to be as mnemonic as possible, and the I command, which stands for Inventory, incorporates the mnemonics. When you run the program, you will first get a listing of the data. Then you will receive a message telling you that you may obtain a listing of all commands by typing I. Then you will be given the COMMAND: prompt. You will be given this prompt whenever the program is ready for a command to be entered. You need not be concerned about upper- and lower-case distinctions in command names. All command names may be entered in either case. In text, I am using upper-case letters for commands to emphasize them. In examples, they are lower-case. Row or column names, however, must be entered exactly as they appear. For example, the sequence "R 1 ng 2 sg"
is exactly equivalent to "r 1sg 2sg" but not to "R 1SG 2SG". Here is an example of the I command:

<table>
<thead>
<tr>
<th>Command</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Display</td>
<td>Print matrix at terminal</td>
</tr>
<tr>
<td>P</td>
<td>Permanent</td>
<td>Write matrix to permfile</td>
</tr>
<tr>
<td>T</td>
<td>Temporary</td>
<td>Write matrix to tempfile</td>
</tr>
<tr>
<td>C y1 y2</td>
<td>Column</td>
<td>Move column y1 to y2 position</td>
</tr>
<tr>
<td>R x1 x2</td>
<td>Row</td>
<td>Move row x1 to x2 position</td>
</tr>
<tr>
<td>S</td>
<td>Strip</td>
<td>Create submatrix</td>
</tr>
<tr>
<td>M</td>
<td>Merge</td>
<td>Combine two rows or columns</td>
</tr>
<tr>
<td>Q</td>
<td>Quit</td>
<td>End Session</td>
</tr>
<tr>
<td>I</td>
<td>Inventory</td>
<td>Display inventory of commands</td>
</tr>
</tbody>
</table>

Example 1. - Inventory of Commands

This listing tells you what commands are available, a mnemonic for remembering that command, and a short description of what that command does. The command I may be entered at any COMMAND: prompt, enabling you to review what commands are available at any time.

3. Display--D: When you run the program, it retrieves the data from a file and stores it in memory. It automatically displays the data once. From then on, it is your responsibility to tell the program when to display the data. This is done by entering the D (or d) command on the COMMAND: line. The program will then display the data.

A number of commands may be entered on one line before hitting RETURN, which causes the command line to be processed. Commands on the command line are separated by blanks. As you become
familiar with the way the commands work, you will want to place several commands on a line. Frequently, you will want the last command on the line to be D. This will cause the matrix to be displayed after all other commands are processed. You may place the D command before other commands, but this is not very useful. However, if you make a mistake typing a command, that command and all subsequent commands on that line will be ignored. You may then have to enter the D command to find out how far the program got before it found an error. You will see numerous examples of the D command.

4. Moving Columns and Rows—R and C: These two commands are discussed together since they work in a similar fashion. What they do is provide the basic matrix permutations as described by Pike. They are the only commands which accept arguments. All other commands will query you for the necessary information. They each require two arguments: two row names or two column names. A row or column name is simply the first item in that row or column. They each move the first argument to the position of the second argument, and move everything between the first and second arguments, including the second argument, in the direction of the first argument to fill up the hole. Here are a couple of examples to demonstrate how this works. All examples involve manipulations of the previous matrix displayed, in this case the Chukchi verb paradigm matrix as shown at the beginning of the paper.
| Command: r 1sg 3pl
| Command: c 2sg 3sg
| Command: d

<table>
<thead>
<tr>
<th>1sg</th>
<th>3sg</th>
<th>2sg</th>
<th>1pl</th>
<th>2pl</th>
<th>3pl</th>
</tr>
</thead>
<tbody>
<tr>
<td>-g?e</td>
<td>-N-an</td>
<td>-tlen</td>
<td>-g?e</td>
<td>-N-ane-t</td>
<td></td>
</tr>
<tr>
<td>-g?e</td>
<td>-g/Na</td>
<td>-gat</td>
<td>-mak</td>
<td>-n/Na</td>
<td>-g/Na-mi</td>
</tr>
<tr>
<td></td>
<td>-ni-n</td>
<td></td>
<td>-tak</td>
<td></td>
<td>-ne-t</td>
</tr>
<tr>
<td>-N-an</td>
<td>-gat</td>
<td></td>
<td>-n/Na</td>
<td></td>
<td>-N-ane-t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-tak</td>
<td></td>
<td>-N-ane-t</td>
</tr>
<tr>
<td>-n/Na</td>
<td>-Na-tka</td>
<td>-tku-n</td>
<td></td>
<td></td>
<td>-Na-tka</td>
</tr>
<tr>
<td>-tak</td>
<td></td>
<td>-tak</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-gam</td>
<td>-N-an</td>
<td>-gat</td>
<td>-mak</td>
<td>-n/Na</td>
<td>-N-ane-t</td>
</tr>
<tr>
<td></td>
<td>-n-an</td>
<td>-gat</td>
<td></td>
<td>-n/N</td>
<td>-N-ane-t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-tak</td>
<td></td>
</tr>
</tbody>
</table>

Example 2. - Moving Rows and Columns

In this example, row 1sg was moved to the position of 3sg, then 3sg and 2sg were moved up one position. Then columns 2sg and 3sg were switched, and the matrix was displayed using the D command.

Since a large number of commands can be entered on one line, you can do a number of permutations at once, and display the final result. In this example, the previous matrix is permuted using several commands simultaneously, and then displayed using a D command at the end of the line.
command: r 1sg 3sg c 3pl 2sg r 1pl 2pl d
s\o 1sg 3sg 3pl 2sg 1pl 2pl

2sg -g?e -N-an -N-an-e-t -tlen -g?e
1sg -n-an -N-an-e-t -gat -n/N -tak
3sg -g?o -g/Na -g/Na-ni -gat -mak -n/Na -tak
-ni-n -ne-t -tak
2pl -n/Na -Na-tka -Na-tka -tku-n -tak
- tkak
1pl -N-an -N-an-e-t -gat -n/Na -tak
- -tak
3pl -gam -N-an -N-an-e-t -gat -mak -n/Na -tak
- -tak

command:

Example 3. - Moving Rows and Columns

5. Writing Data to Files -- T and P: A useful feature of this program is that you may at any point write the matrix you have obtained out to a file. This file may then be printed or used in subsequent runs of the program. This means that you can save your work at any point and return to it at that point.

Two files, called "permfile" and "tempfile" are provided for this purpose. Regardless of the names, both of them are temporary files. The names were chosen because it is likely that you will wish to write numerous versions of the matrix out to the file called "tempfile", which you can then print, and save a final version of the program on "permfile" for subsequent use. This is entirely up to the user, however. Since they are temporary files, they will be lost when the time-sharing session ends. Steps must be taken to preserve them, if they are going to be needed later. The use of output files is a feature of Pascal which adds significant power to the program. Once a body of data has been placed in a matrix on a file, it need never be entered again, but may be manipulated using the program and restored.
The commands used to write a matrix to a file are P and T. As their names suggest, P writes the matrix to the file called "perm-file", and T writes the matrix to the file called "temp-file". The matrix may be written to a file several times during a single session; the matrices will simply be placed one after the other on the file. If you exit the program and then run the program again during the same time-sharing session, anything stored on "tempfile" or "permfile" will cease to exist. Here is an example of writing to a temporary file:

command: t
Matrix written to temporary file called tempfile.
Save or print tempfile.
command:

Example 4. - Writing to Temporary Files

6. Merging Columns and Rows: In some cases, it may be desirable to merge two columns or rows. This is accomplished with the M command. The program will ask you if you wish to merge rows, columns, or nothing, to which you respond R, C, or N. M permits you to change your mind. If you respond with R or C, you will be asked which columns or rows should be merged, to which you should respond with two row or column names. The second row or column you specify will then be merged into the first.

The order you specify the names is important. The first row or column you specify will retain its integrity in the merge. That is, no information that was in the first row or column will be lost. Information may be lost in the second row or column if there is data in the same cell in both the first and second. The first one specified always takes precedence over the second. After the merge, the program tries to find a blank spot in the heading to place the second name you specified. This will usually be in the same column and in the following physical row as the first name you gave. Here is an example using M:
In this example, rows 1sg and 1pl were merged, with no loss of information in either row, since the two rows were identical before the merge. This is a way to reduce a matrix when redundancy occurs in rows and columns.

7. Removing Elements--S: The most complicated command is the S command, which stands for Submatrix or Strip. This command will remove blocks of the matrix, of any size, so long as they are rectangular in shape. Irregular shapes may be removed by using the command several times. To use this command, type S on the command line. The program will ask you for the first and last columns you wish to affect. To this you may respond with a single column name, two column names, or N for none. If you respond with a single column name, the program will assume the S command to affect only that column. If you respond with two column names, a range between, and including, those columns will be affected. If you enter N, none of the columns will be affected.
The program will then ask you which rows you wish to be affected. Again, you may respond with one row name, two row names, or N. The program will then mark the rows and columns you specified in response to these queries.

The next step will be to remove all of those columns and rows in the boundaries you have instructed the program to use. In the following example, a block consisting of four cells is removed by specifying two row names and two column names.

<table>
<thead>
<tr>
<th>command: s</th>
<th>Enter first and last columns (if any) to be stripped. (S = none)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3pl 2sg</td>
<td>Enter first and last rows (if any) to be stripped. (N = none)</td>
</tr>
<tr>
<td>1sg 3sg</td>
<td></td>
</tr>
</tbody>
</table>

**command: d**

<table>
<thead>
<tr>
<th>s\o</th>
<th>1sg</th>
<th>3sg</th>
<th>3pl</th>
<th>2sg</th>
<th>1pl</th>
<th>2pl</th>
</tr>
</thead>
<tbody>
<tr>
<td>2sg</td>
<td>-g?e</td>
<td>-N-an</td>
<td>-N-ane-t</td>
<td>-tlcn</td>
<td>-g?e</td>
<td></td>
</tr>
<tr>
<td>1sg</td>
<td>-n-an</td>
<td>-n/N</td>
<td>-tak</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/1pl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3sg</td>
<td>-g?e</td>
<td>-g/Na</td>
<td>-ni-n</td>
<td>-mak</td>
<td>-n/Na</td>
<td>-tak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2pl</td>
<td>-n/Na</td>
<td>-Na-tka</td>
<td>-Na-tka</td>
<td>-tku-n</td>
<td>-tak</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3pl</td>
<td>-gam</td>
<td>-N-an</td>
<td>-N-ane-t</td>
<td>-gat</td>
<td>-mak</td>
<td>-n/Na</td>
</tr>
</tbody>
</table>

**command:**

Example 6. - Removing Columns or Rows

If an N response is given to both questions, nothing will be changed. This permits you to change your mind after entering S on the command line.

A subtlety of this command is that it permits you to remove entire rows or columns if you specify N to one of the prompts, but not to both. Suppose, for example, you answer with one row name for the
row prompt and \( N \) for the column prompt. The row you specified will be removed from the matrix. If you enter \( N \) for the row prompt and two column names for the column prompt, those columns and any columns between them will be removed from the matrix. Here is an example of removing entire columns. Note that the \( B \) command placed after the \( S \) command causes the matrix to be automatically displayed after the \( S \) command is processed. In general, any combination of commands may be placed upon the command line and they will be processed until the end of the line or until an erroneous command is encountered.

\[
\begin{array}{cccc}
\text{command:} & s & d \\
\text{Enter first and last columns (if any) to be stripped. (N = none)} & 2sg & 3sg \\
\text{n} & n/o & lsg & lpi & 2pl \\
\text{2sg} & \text{-g7e} & \text{-tlen} & \text{-g7e} \\
\text{lsg} & \text{-n/N} & \text{-tak} \\
\text{lpi} & \text{-tak} \\
\text{3sg} & \text{-g7e} & \text{-mak} & \text{-n/Na} & \text{-tak} \\
\text{lpl} & \text{-n/Na} & \text{-tku-n} & \text{-tak} \\
\text{lm} & \text{-tak} \\
\text{3pl} & \text{-gsm} & \text{-mak} & \text{-n/Na} & \text{-tak} \\
\end{array}
\]

command:

Example 7. - Removing Columns or Rows

8. Exiting the Program--\( Q \): Only one command remains to be discussed, the \( Q \) command. This command causes the program to terminate and return you to system level.
Program Listing

Following is a listing of the program, which is written in
Pascal. The ELSE in the CASE statement in the main program is non-
standard. Many compilers support it, but if yours does not, it should
be omitted from the program.

program permute(input, datafile, tempfile, permfile, output);

(* Author: Dana Barrager
Date: 10/83
Purpose: To perform matrix manipulations upon a matrix
read in from a data file, and write to two
output files. *)

const
linelen = 80; (* length of command line *)
maxcellsize = 10; (* maximum number of columns per cell *)
maxlen = 5; (* maximum number of rows per cell *)
rowmax = 50; (* maximum number of rows in matrix *)
colmax = 150; (* maximum number of columns in matrix *)
blank = ' '; (* blank constant *)
type
commandtype = packed array[1..linelen] of char;
celltype = packed array[1..maxcellsize] of char;
columntype = array[1..rowmax] of celltype;
linetype = packed array[1..colmax] of char;
rowtype = array[1..maxlen] of linetype;
matrixinfo = record
  linenum : integer; (* # of lines per cell *)
  cellsize : integer; (* # of columns per cell *)
  rownum : integer; (* # total # of rows in matrix *)
  colnum : integer; (* # total # of columns in matrix *)
  flag : boolean; (* trace variable *)
  data : array[1..rowmax, 1..colmax] of char;
  matrix ; (* matrix *)
end; (* record *)

var
matrix : matrixinfo; (* matrix from input file *)
head : boolean; (* switch variable *)
tempfile : text; (* file to be printed *)
datafile : text; (* matrix data file *)
permfile : text; (* file to be saved *)
command : commandtype; (* input command line *)
i : integer; ( * loop index variable *)
error : boolean; ( * error flag *)
done : boolean; ( * termination flag *)

procedure buildmatrix(var matrix : matrixinfo);
( * Procedure to read data from a data file. *)

var
  i, j, k, l, m : integer; ( * loop index variables *)

begin ( * procedure buildmatrix *)

matrix.flag := false; ( * set to true for trace of procedure calls. *)
if matrix.flag then writeln ('now in buildmatrix');
with matrix do
begin
  reset(datafile);
  readln(datafile, linenumber, cellsize, rownumber, colnumber);
  colnum := colnum div cellsize;
i := 1;
while (i <= rownumber) and (not eof(datafile))) do
begin
  j := 1;
k := 1;
while ((j <= colnumber) and (not eof(datafile))) do
begin
  l := 1;
  while ((l <= cellsize) and (not eof(datafile))) do
begin
    read(datafile, data[i, k]);
k := k + 1;
l := l + 1;
  end; ( * while *)
j := j + 1;
end; ( * while *)
readln(datafile);
for m := k to colmax do data[i, m] := ' ';
i := i + 1;
end ( * while *)
end ( * with *)
end; ( * procedure buildmatrix *)

procedure inventory(flag : boolean; var i : integer);
( * Procedure to print a list of valid commands *)
begin (* procedure inventory *)
if flag then writeln('now in inventory');
writeln('Command Mnemonic Function');
writeln;
writeln(' D - Display - Print matrix at terminal');
writeln;
writeln(' P - Permanent - Write matrix to permfile');
writeln;
writeln(' T - Temporary - Write matrix to tempfile');
writeln;
writeln(' C y1 y2 - Column - Move column y1 to y2 position');
writeln;
writeln(' R x1 x2 - Row - Move row x1 to x2 position');
writeln;
writeln(' S - Strip - Create submatrix');
writeln;
writeln(' M - Merge - Combine two rows or columns');
writeln;
writeln(' Q - Quit - End Session');
writeln;
writeln(' I - Inventory - Display inventory of commands');
writeln;
end; (* procedure inventory *)

procedure writematrix(var outfile: text; var listpos: integer;
var matrix: matrixinfo; head: boolean);

(* Procedure to write matrix to output file. The output file
is passed in as a parameter. If the output file is not the
terminal, the formatting information is printed at the beginning
of the file, so the output files may be read in as input files.*)

var
i, j, k: integer; (* loop index variables *)
begin (* procedure writematrix *)
if matrix.flag then writeln('now in writematrix');
with matrix do
begin
k := (column * cellsize);
if head then writeln(outfile, linenum, cellsize, rownum, k);
i := 1;
end;
while i <= rownum do
  begin
    for j := 1 to (colnum * cellsize) do
      write(outfile, data[i, j]);
    writeln(outfile);
    i := i + 1
  end; (* while *)
end; (* with *)
listpos := listpos + 1
end; (* procedure writematrix *)

procedure getarg(var argcell : ceulltype; var listpos : integer;
                  command : commandtype);

(* Procedure to return a row or column name argument from the
  command line *)

var
  i : integer; (* loop index variable *)
begin
  if matrix.flg then writeln('Now in getarg');
  while (command[listpos] = blank) and (listpos < linelength) do
    listpos := listpos + 1;
  i := 1;
  while((command[listpos] < blank) and (listpos < linelength))
    and (i <= maxcellsize) do
    begin
      argcell[i] := command[listpos];
      i := i + 1;
      listpos := listpos + 1
    end; (* while *)
  while (i <= maxcellsize) do
    begin
      argcell[i] := blank;
      i := i + 1;
    end; (* while *)
end; (* procedure getarg *)

procedure locatecol(var arg : integer; argcell : celltype;
                     var matrix : matrixinfo; var found : boolean);

(* Procedure to locate the position in the matrix of a column
  argument on the command line *)
var
  i, j : integer; (* loop index variables *)
  tempcell : celltype; (* temporary storage variable *)
begin (* procedure locatecol *)
  if matrix.flag then writeln('Now in locatecol');
  j := 1;
  found := false;
  while ((not found) and (j <= (matrix.colnum * matrix.cellsizel))) do begin
    for i := 1 to matrix.cellsizel do begin
      tempcell[i] := matrix.data[i,j];
      j := j + 1
    end; (* for *)
    for i := (matrix.cellsizel + 1) to maxcellsizel do
      tempcell[i] := blank;
    if argcell = tempcell then begin
      found := true;
      arg := (j - matrix.cellsizel);
    end (* then *)
  end (* while *)
end; (* procedure locatecol *)

procedure getcolarg(var arg1, arg2 : integer; var matrix: matrixinfo;
  var listpos : integer; command : commandtype;
  argnum : integer; var found : boolean);
(* Procedure to process command line arguments, returning integer values for column names in commands using two previous procedures *)

var
  argcell : celltype; (* temporary storage variable *)
begin
  if matrix.flag then writeln('Now in getcolarg');
  getarg(argcell, listpos, command);
  locatecol(arg1, argcell, matrix, found);
  if found then begin
    if argnum = 2 then begin
      getarg(argcell, listpos, command);
      locatecol(arg2, argcell, matrix, found)
    end (* then *)
  end; (* procedure getcolarg *)
procedure movecolumn (var matrix : matrixinfo; var error : boolean;
                 var listpos : integer; command : commandtype);

(* Procedure to move columns specified by the C command.
It uses the previous three procedures to determine which
columns to move. *)

var
  arg1, arg2 : integer; (* the column positions of args *)
  argnum : integer; (* switch variable for getcolarg *)
  dif : integer; (* positive or negative direction vbl. *)
  tempcol : columntype; (* temporary storage for columns *)
  offset : integer; (* offset variable *)
  current, next : integer; (* column number indicators *)
  i, j : integer; (* loop index variables *)
  found : boolean; (* boolean variable for valid arguments *)
begin (* procedure movecol *)
  if matrix.flag then writeln('Now in movecolumn.');
  listpos := listpos + 1;
  argnum := 2;
  getcolarg(arg1, arg2, matrix, listpos, command, argnum, found);
  if not found then error := true
  else begin
    with matrix do begin
      if arg1 < arg2 then dif := -cellsize
      else if arg1 > arg2 then dif := cellsize
      else error := true;
      if not error then begin
        offset := cellsize - 1;
        for i := 1 to rowmax do
          for j := 0 to offset do tempcol[i,(j + 1)] := data[i, (arg1 + j)];
        current := arg1;
        next := arg1 + dif;
        while current < arg2 do begin
          for i := 1 to rowmax do
            for j := 0 to offset do data[i, (current + j)] := data[i, (next + j)];
          current := current + dif;
          next := next + dif;
        end
      end
    end
  end
end
end; (* while *)
for i := 1 to rowmax do
  for j := 0 to offset do
    data[i,(arg2 + j)] := tempcol[i,(j + 1)]
  end (* then *)
end (* with *)
end (* else *)
end; (* procedure movecol *)

procedure locaterow(var arg : integer; argcell : celttype;
  var matrix : matrixinfo; var found : boolean);

(* Procedure to locate the position in the matrix of a row
argument on the command line. *)

var
  i, j : integer; (* loop index variables *)
  tempcell : celttype; (* temporary storage variable *)

begin (* procedure locaterow *)
  if matrix.flag then writeln('Row in locaterow');
  j := 0;
  found := false;
  while ((not found) and (j < matrix.rownum)) do begin
    j := j + 1;
    for i := 1 to matrix.cellsize do
      tempcell[i] := matrix.data[j,i];
    for i :=(matrix.cellsize + 1) to maxcellssize do
      tempcell[i] := blank;
    if (argcell = tempcell) and (tempcell < ' ') then begin
      found := true;
      arg := j;
    end (* if *)
  end (* while *)
end; (* procedure locaterow *)

procedure getrowarg(var argl, arg2 : integer; var matrix: matrixinfo;
  var listpos : integer; command : commandtype;
  argnum : integer; var found : boolean);

(* Procedure to process command line arguments, returning integers
for row names in commands using previous procedure and getarg *)
var
argcell : celltype; (* temporary storage variable *)

begin
if matrix.flag then writeln('Now in getrowarg');
getarg(argcell, listpos, command);
locaterow(arg1, argcell, matrix, found);
if found
then if argnum = 2
then begin
getrow(argcell, listpos, command);
locaterow(arg2, argcell, matrix, found)
end (* then *)
end; (* procedure getrowarg *)

procedure moverow (var matrix : matrixinfo; var error : boolean;
var listpos : integer; command : commandtype);

(* Procedure to move rows specified by the R command.
It uses the previous two procedures and getarg to
determine which rows to move. *)

var
arg1, arg2 : integer; (* the column positions of args to R *)
arginum : integer; (* switch variable for getrowarg *)
dif : integer; (* positive or negative direction vbl *)
temprow : rowtype; (* temporary storage for rows *)
current, next : integer; (* row number indicators *)
i, j, k : integer; (* loop index variables *)
found : boolean; (* boolean variable for valid arguments *)

begin (* procedure moverow *)
if matrix.flag then writeln('Now in moverow.');
listpos := listpos + 1;
arginum := 2;
getrowarg(arg1, arg2, matrix, listpos, command, argnum, found);
if not found
then error := true
else begin
with matrix do
begin
if arg1 < arg2
then dif := 1
else if arg1 > arg2
then dif := -1
else error := true;
if not error
then begin
  k := 1;
  for j := arg1 to (arg1 + (linenum -1)) do begin
    for i := 1 to (cellsize * colnum) do
      temprow[k, i] := data[j, i];
    k := k + 1
  end; (* for *)
  if dif = 1 then k := linenum - 1
else k := 0;
  j := linenum;
  while j <= 0 do begin
    if dif = 1 then current := arg1
else current := arg1 + (linenum - 1);
  next := current + dif;
  while current < (arg2 + k) do begin
    for i := 1 to (cellsize * colnum) do
      data[current, i] := data[next, i];
    current := current + dif;
    next := next + dif;
  end; (* while *)
  k := k - dif;
  j := j - 1;
end; (* while *)
  for j := 0 to (linenum - 1) do begin
    for i := 1 to (cellsize * colnum) do
      data[(arg2 + j), i] := temprow[(j + 1), i];
  end (* then *)
end (* with *)
end (* else *)

end; (* procedure moverow *)

procedure strip(arg1, arg2, arg3, arg4 : integer;
  var matrix : matrixinfo; moverow, movecol : boolean);

(* Procedure to remove portions of the matrix *)

var
  i, j : integer; (* loop index variables *)
begin (* procedure stripcol *)
with matrix do
begin
if flag then writeln('now is stripcol');
if (not moverow) and (not movecol)
then begin
for i := arg3 to (arg4 + (linenum - 1)) do
for j := arg1 to (arg2 + (cellsize - 1)) do
data[i, j] := blank;
end (* then *)
else begin
if movecol then
for i := arg3 to arg4 do
for j := arg1 to ((column * cellsize) - (arg2 - arg1 + cellsize))
do
data[i, j] := data[i, (j + cellsize + arg2 - arg1)];
if moverow then
for i := arg3 to (roumnun - (arg4 - arg3 + linenum - 1)) do
for j := arg1 to (arg2 + cellsize) do
data[i, j] := data[(i + (arg4 - arg3) + linenum), j]
end (* else *)
end (* with *)
end; (* procedure strip *)

procedure submatrix(var matrix : matrixinfo; var listpos : integer);

(* Procedure to determine which portions of a matrix should be
removes when the S command is used. The user is queried
for necessary information. *)

var
moverow, movecol : boolean; (* switch vbls. for strip *)
command : commandtype; (* user input *)
i, argnum : integer; (* counter variables *)
arg1, arg2, arg3, arg4, dummy : integer; (* row arguments *)
done : boolean; (* dummy variable *)
sound : boolean; (* loop termination variable *)
errcount : integer; (* valid row or column vbl *)

procedure swap(var arg1, arg2 : integer);
(* procedure to swap two values if arg1 arg2 *)

var
    i : integer; (* temporary storage variable *)
begin (* procedure swap *)
    if arg2 < arg1 then begin
        i := arg2;
        arg2 := arg1;
        arg1 := i;
    end (* then *)
end; (* procedure swap *)

begin (* procedure submatrix *)
    if matrix.flag then writeln('now in submatrix');
    errorcount := 0;
    argnum := 1;
    listpos := listpos + 1;
    done := false;
    moverow := false;
    movercol := false;
    i := 1;
    while (not done) and (errorcount < 3) do begin
        writeln
            ('Enter first and last columns (if any) to be stripped. (N = none)');
        readin(command);
        getcolarg(arg1, dummy, matrix, i, command, argnum, found);
        if (not found) and ((command[1] = 'n') or (command[1] = 'N'))
            then begin
                moverow := true;
                arg1 := i;
                arg2 := (matrix.colnum - 1) * matrix.cellsize;
                done := true;
            end (* then *)
        else begin
            if found
                then begin
                    getcolarg(arg2, dummy, matrix, i, command, argnum, found);
                    if not found then arg2 := arg1;
                    done := true;
                    swap(arg1, arg2);
                end (* then *)
        end (* if *)
    end (* while *)
    errorcount := errorcount + 1;
writeln('Column not found--ignored.')
end (* else *)
end; (* while *)
i := 1;
done := false;
while (not done) and (errorcount < 3) do
begin
writeln
('Int=6 first and last rows (if any) to be stripped. (N = none)' );
radin(command);
getrowarg(arg3, dummy, matrix, i, command, argnum, found);
if (not found) and ((command[1] = 'n') or (command[1] = 'N'))
then begin
movecol := true;
arg3 := 1;
arg4 := matrix.rownum;
done := true;
end; (* then *)
else begin
if found
then begin
getrowarg(arg4, dummy, matrix, i, command, argnum, found);
if not found then arg4 := arg3;
swap(arg3, arg4);
done := true
end; (* then *)
else begin
errorcount := errorcount + 1;
writeln('Row not found--ignored.' );
end; (* else *)
end; (* else *)
end; (* while *)
if done
then begin
if (not moverow) or (not movecol)
then begin
strip(arg1, arg2, arg3, arg4, matrix, moverow, movecol);
if moverow then
matrix.rownum :=
(matrix.rownum - (arg4 - arg3 + matrix.linenum));
if movecol then
matrix.colnum :=
(matrix.colnum - ((arg2 + matrix.cellsiz - arg1))
div matrix.cellsiz);
end; (* then *)
end; (* then *)
else writeln('Too many errors.')
procedure mergeset(var matrix : matrixinfo; var errorcount : integer);

(* Procedure to merge two columns, as instructed by the user
   in response to query. *)

var
  command : commandtype; (* input type from asser *)
  i, j, k : integer; (* loop index variables *)
  argnum : integer; (* argument number for getcolarg *)
  arg1, arg2 : integer; (* column arguments *)
  blanknum : integer; (* number of blanks in column heading *)
  tempcell : celltype; (* temporary storage cell *)
  found : boolean; (* valid argument variable *)

begin (* procedure mergeset *)
  if matrixflag then writeln('Now in mergeset');
  write('Enter columns to be merged: ');
  argnum := 2;
  readln(command);
  i := 1;
  getcolarg(arg1, arg2, matrix, i, command, argnum, found);
  if (arg1 < arg2) and (found) then begin
    with matrix do
      begin
        j := 3;
        while j <= rownum do
          begin
            i := 1;
            for k := arg1 to (arg1 + (cellsizex - 1)) do
              begin
                tempcell[i] := data[j, k];
                i := i + 1;
              end;
            for k := (cellsizex + 1) to maxcellsize do
              tempcell[k] := blank;
            if tempcell = ' ' then for k := 0 to (cellsizex - 1) do
              begin
                data[j, (arg1 + k)] := data[j, (arg2 + k)];
                j := j + 1
              end;
            end; (* while *)
          for i := 0 to (cellsizex - 1) do
            data[j, (arg1 + i)] := data[1, (arg2 + i)];
        for k := 1 to rownum do
          begin
            for i := 0 to (cellsizex - 1) do
              data[j, (arg1 + i)] := data[j, (arg2 + i)];
            end;
          end;
        end;
      end;
    end;
end; (* procedure mergeset *)

end; (* procedure submatrix *)
for j := arg2 to (colnum * cellsize) do
  data[k, j] := data[k, (j + cellsize)];
colnum := colnum - 1
end (* with *)
end (* then *)
else begin
  writeln('Column not recognized--ignored.');
  errorcount := errorcount + 1
end (* else *)
end; (* procedure mergecol *)

procedure mergerow(var matrix : matrixinfo; var errorcount : integer);
(* Procedure to merge two rows, as instructed by the user
  in response to query. *)
var
  command : commandtype; (* input line from user *)
  i, j, k : integer; (* loop index variables *)
  argnum : integer; (* argument number for getrowarg *)
  arg1, arg2 : integer; (* row arguments *)
  blanknum : integer; (* number of blanks in row heading *)
  tempcell : celltype; (* temporary storage cell *)
  found : boolean; (* valid argument variable *)
begin (* procedure mergerow *)
  if matrix.flag then writeln('Now in mergerow');
  i := 1;
  argnum := 2;
  writeln('Enter rows to be merged: ');
  readln(command);
  getrowarg(arg1, arg2, matrix, i, command, argnum, found);
  if (arg1 < arg2) and (found)
    then begin
      with matrix do
        begin
          j := cellsize;
          while j < (colnum * cellsize) do begin
            k := 1 to cellsize do
              tempcell[k] := data[arg1, (k + j)];
            for k := (cellsizel + 1) to maxcellsize do
              tempcell[k] := blank;
            if tempcell = '
              then for i := 0 to (linenum - 1) do
                for k := (j + 1) to (j + cellsize) do
                  data[(arg1 + i), k] := data[(arg2 + i), k];
          end
          i := i + 1
        end
    end
end; (* procedure mergerow *)
j := j + cellsize
end; (* while *)
if linenum = 1 then begin
data[(arg1 + 1), 1] := blank;
data[(arg1 + 1), 2] := '/'
for i := 3 to cellsize do
  data[(arg1 + 1), i] := data[arg2, (i - 2)]
end (* then *)
else begin
  blanknum := -2;
j := cellsize;
while (j < 1) and (data[arg1, j] = blank) do
  begin
    blanknum := blanknum + 1;
j := j - 1
end; (* while *)
k := 1;
j := j + 1;
data[arg1, j] := '/';
j := j + 1;
while (j < cellsize) and (blanknum > 0) do
  begin
    data[arg1, j] := data[arg2, k];
k := k + 1;
j := j + 1;
    blanknum := blanknum - 1
end; (* while *)
end; (* else *)
for k := arg2 to (rownum - linenum) do
  for i := 1 to (colnum * cellsize) do
    data[k, i] := data[(k + linenum), i];
rownum := rownum - linenum
end (* with *)
end (* then *)
else begin
  writeln('Row not recognized--ignored.');
  errorcount := errorcount + 1
end (* else *)
end; (* procedure mergerow *)

procedure merge(var matrix : matrixinfo; var i : integer;
                var error : boolean);
(* procedure to merge rows or columns, as specified by the
  user. User is queried for which he wishes to merge. *)
var
ch : char; (* input character variable *)
done : boolean; (* loop termination variable *)
errorcount : integer; (* alternate loop termination variable *)

begin (* procedure merge *)

if matrix.flag then writeln('now in merge');
errorcount := 0;
i := i + 1;
done := false;
while(not done) and (errorcount <= 3) do
begin
    writeln('Merge columns(C) or rows(R) or none(N)');
    readln(ch);
    if (ch = 'C') or (ch = 'c')
        then begin
        mergerow(matrix, errorcount);
        done := true
        end (* then *)
    else if (ch = 'R') or (ch = 'r')
        then begin
        mergecol(matrix, errorcount);
        done := true
        end (* then *)
    else if (ch = 'N') or (ch = 'n')
        then begin
        writeln('Invalid input. Enter C or R or N.');
        errorcount := errorcount + 1;
        readln;
        end (* else *)
end (* while *)
end; (* procedure merge *)

(* --------------------------------------------------------------- *)

begin (* program permute *)

(* This begins the main program. It is mostly a list of procedure *)
calls. The strategy of the main program is simple. A matrix is read
in from a file using Buildmatrix, then displayed using Writematrix.
Then the user is instructed that a list of commands may be obtained
by typing 1, one of the valid commands. This provides some self-
documentation. Then a while loop is entered which processes a line
of commands each time through. Several commands, separated by spaces,
may be entered on one line. A command line is limited to 80 character
positions. If an erroneous command is entered, that command and all
subsequent commands are ignored. The commands are processed through a
case statement which recognizes valid commands and makes the
appropriate procedure calls. *)

buildmatrix(matrix); (* call procedure to input matrix *)
rewrite(tempfile); (* open file for writing *)
rewrite(permfile); (* open file for writing *)
writematrix(output, i, matrix, false);
writeln('If you need an inventory of commands, type "I".');
done := false;
while (not done) do
begin (* begin processing commands *)
if (not done) then writeln('command: ');
readln(command);
i := 1;
error := false;
while (((not done) and (i < linelength)) and (not error)) do
begin (* process input line *)
while (((command[i] = ' ') and (i < linelength)) do i := i + 1;
if i < linelength
then begin
case command[i] of
'I': inventory(matrix.flag, i);
'i': inventory(matrix.flag, i);
'D': writematrix(output, i, matrix, false);
'd': writematrix(output, i, matrix, false);
'P': begin
writematrix(permfile, i, matrix, true);
writeln('Matrix written to temporary file called permfile.');
writeln('Save or print permfile.');
end;
'p': begin
writematrix(permfile, i, matrix, true);
writeln('Matrix written to temporary file called permfile.');
writeln('Save or print permfile.');
end;
'T': begin
writematrix(tempfile, i, matrix, true);
writeln('Matrix written to temporary file called tempfile.');
writeln('Save or print tempfile.');
end;
end;
end;
end;
't': begin
  writematrix(tempfile, i, matrix, true);
  writeln('Matrix written to temporary file called tempfile.');
  writeln('Save or print tempfile.');
end;
'C': movecolumn(matrix, error, i, command);
'C': movecolumn(matrix, error, i, command);
'R': moverow(matrix, error, i, command);
'R': moverow(matrix, error, i, command);
'S': submatrix(matrix, i);
'S': submatrix(matrix, i);
'H': merge(matrix, i, error);
'H': merge(matrix, i, error);
'Q': done := true;
'Q': done := true;
else error := true
end (* case #)
end; (* then #)
if error
  then begin
    writeln('Invalid command or parameter, ignored.');
  end (* then #)
end (* while #)
end. (* program permute *)
REFERENCES


