Volume 9
1984

Kansas Working Papers in Linguistics

edited by

Letta 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

Studies in Native American Languages III

Kansas Working Papers in Linguistics

Volume 9, 1984

Articles

Paul Voorhis	Catawba Morphology in the Texts of Frank Speck and of Matthews Red Thunder Cloud	1
Nile R. Thompson Douglas W. Isaacson	Lexical Representation of Salish Verb Roots: A Preliminary Examination	31
John E. McLaughlin	A Revised Approach to Southern Paiute Phonology	47
John E. McLaughlin	JENNY: An Interactive Program in Basic for Analyzing Comanche (and Other) Texts (With Sample Text)	81
Danna Barrager	Description of a Pikean Field Matrix Permutation Program	95
David L. Shaul Katherine Turner James Collins	Esselen Linguistic Materials	127
Joseph F. Kess Anita Copeland	The Structure and Function of Nootkan Baby Talk	141
Contents of Previous Volumes		165

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 3sg 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.

3 9 21 63

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

Symbol	IPA Equivalent
a	ə
?	ʔ
g	ɣ
N	ŋ

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 comands. 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.

<u>Using the Matrix Permutation Program</u>

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: lsg refers to the physical row on which the letters "lsg" 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 1sg 2sg"

is exactly equivalent to "r 1sg 2sg" but not to "R 1SG 2SG". Here is an example of the I command:

```
If you need an inventory of commands, type "I".
command: i
Command     Mnemonic    Function


 D       - Display   - Print matrix at terminal

 P       - Permanent - Write matrix to permfile

 T       - Temporary - Write matrix to tempfile

 C y1 y2 - Column    - Move column y1 to y2 position

 R x1 x2 - Row       - Move row x1 to x2 position

 S       - Strip     - Create submatrix

 M       - Merge     - Combine two rows or columns

 Q       - Quit      - End Session

 I       - Inventory - Display inventory of commands

command:
```

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
s\o     1sg     3sg     2sg     1pl     2pl     3pl


2sg     -g?e    -N-an           -tlen           -N-ane-t
                                -g?e

3sg     -g?e    -g/Na   -gat    -mak    -n/Na   -g/Na-ni
                -ni-n                   -tak    -ne-t

1pl             -N-an   -gat            -n/Na   -N-ane-t
                                        -tak

2pl     -n/Na   -Na-tka         -tku-n          -Na-tka
        -tak                    -tak

3pl     -gam    -N-an   -gat    -mak    -n/Na   -N-ane-t
                                        -tak

1sg             -n-an   -gat            -n/N    -N-ane-t
                                        -tak

command:
```

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-ane-t		-tlen -g?e	
1sg		-n-an	-N-ane-t	-gat		-n/N -tak
3sg	-g?e	-g/Na -ni-n	-g/Na-ni -ne-t	-gat	-mak	-n/Na -tak
2pl	-n/Na -tak	-Na-tka	-Na-tka		-tku-n -tak	
1pl		-N-an	-N-ane-t	-gat		-n/Na -tak
3pl	-gam	-N-an	-N-ane-t	-gat	-mak	-n/Na -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 "permfile", and T writes the matrix to the file called "tempfile". 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--M: 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. N 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:

```
command: m
Merge columns(C) or rows(R) or none(N)
r
Enter rows to be merged:  1sg 1pl
command: d
s\o      1sg      3sg      3pl      2sg      1pl      2pl


2sg      -g?e     -N-an    -N-ane-t          -tlen
                                             -g?e

1sg               -n-an    -N-ane-t -gat              -n/N
 /1pl                                                 -tak

3sg      -g?e     -g/Na    -g/Na-ni -gat     -mak     -n/Na
                  -ni-n    -ne-t                      -tak

2pl      -n/Na    -Na-tka  -Na-tka           -tku-n
         -tak                                -tak

3pl      -gam     -N-an    -N-ane-t -gat     -mak     -n/Na
                                                      -tak

command:
```

Example 5. - Merging Columns or Rows

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.

```
command: s
Enter first and last columns (if any) to be stripped. (N = none)
3pl 2sg
Enter first and last rows (if any) to be stripped. (N = none)
1sg 3sg
command: d
s\o      1sg      3sg      3pl      2sg      1pl      2pl


2sg      -g?e     -N-an    -N-ane-t          -tlen
                                             -g?e

1sg               -n-an                               -n/N
 /1pl                                                 -tak

3sg      -g?e     -g/Na                      -mak     -n/Na
                  -ni-n                               -tak

2pl      -n/Na    -Na-tka  -Na-tka           -tku-n
         -tak                                -tak

3pl      -gam     -N-an    -N-ane-t -gat     -mak     -n/Na
                                                      -tak

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 D 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.

```
command: s d
Enter first and last columns (if any) to be stripped. (N = none)
2sg 3sg
Enter first and last rows (if any) to be stripped. (N = none)
n
s\o      1sg      1pl      2pl


2sg      -g?e     -tlen
                  -g?e

1sg                        -n/N
 /1pl                      -tak

3sg      -g?e     -mak     -n/Na
                           -tak

2pl      -n/Na    -tku-n
         -tak     -tak

3pl      -gam     -mak     -n/Na
                           -tak

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
  linelength  = 80;   (* length of command line *)
  maxcellsize = 10;   (* maximum number of columns per cell *)
  maxlinenum  = 5;    (* maximum number of rows per cell *)
  rowmax      = 50;   (* maximum number or rows in matrix *)
  colmax      = 150;  (* maximum number of columns in matrix *)
  blank       = ' ';  (* blank constant *)

type
  commandtype = packed array[1..linelength] 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..maxlinenum] 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 *)
                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, linenum, cellsize, rownum, colnum);
colnum := colnum div cellsize;
i := 1;
while((i <= rownum) and (not eof(datafile))) do
  begin
  j:= 1;
  k:= 1;
  while ((j <= colnum) and (not eoln(datafile))) do
    begin
      l := 1;
      while ((l <= cellsize) and (not eoln(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;
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;
i := i + 1
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 := (colnum * cellsize);
if head then writeln(outfile, linenum, cellsize, rownum, k);
i := 1;
```

```
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 : celltype; 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.flag 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.cellsize))) do
  begin
  for i := 1 to matrix.cellsize do
      begin
      tempcell[i] := matrix.data[1,j];
      j := j + 1
      end;  (* for *)
  for i :=(matrix.cellsize + 1) to maxcellsize do
      tempcell[i] := blank;
  if argcell = tempcell
     then begin
       found := true;
       arg := (j - matrix.cellsize);
       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 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 varible 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;  (* 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 : celltype;
                    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 : celltype;  (* temporary storage variable *)

begin  (* procedure locaterow *)
if matrix.flag then writeln('Now 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 maxcellsize do
      tempcell[i] := blank;
  if (argcell = tempcell) and (tempcell <  '          ')
     then begin
       found := true;
       arg := j;
       end  (* then *)
  end  (* while *)
end;  (* procedure locaterow *)


procedure getrowarg(var arg1, 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
             getarg(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 *)
  argnum        : integer;  (* switch varible 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;
argnum := 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
             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 in 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
                    ((colnum * cellsize) - (arg2 - arg1 + cellsize))
           do
                data[i, j] := data[i, (j + cellsize + arg2 - arg1)];
     if moverow then
        for i := arg3 to (rownum - (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
   removed 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,                        (* column arguments *)
  arg3, arg4,                        (* row arguments *)
  dummy              : integer;      (* dummy variable *)
  done               : boolean;      (* loop termination variable *)
  found              : boolean;      (* valid row or column vbl *)
  errorcount         : integer;      (* count of errors *)

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;
movecol := false;
i := 1;
while(not done) and (errorcount < 3) do
  begin
writeln
('Enter first and last columns (if any) to be stripped. (N = none)');
  readln(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 := 1;
       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 *)
          else begin
            errorcount := errorcount + 1;
```

```
            writeln('Column not found--ignored.')
            end  (* else *)
        end  (* else *)
  end;  (* while *)
i := 1;
done := false;
while (not done) and (errorcount < 3) do
  begin
writeln
('Enter first and last rows (if any) to be stripped. (N = none)');
  readln(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.cellsize - arg1)
             div matrix.cellsize);
             end  (* then *)
     end  (* then *)
   else writeln('Too many errors.')
```

```
end;  (* procedure submatrix *)


procedure mergecol(var matrix : matrixinfo; var errorcount : integer);

(* Procedure to merge two columns, 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 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 mergecol *)
if matrix.flag then writeln('Now in mergecol');
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 + (cellsize - 1)) do
           begin
           tempcell[i] := data[j, k];
           i := i + 1
           end;  (* for *)
       for k := (cellsize + 1) to maxcellsize do
           tempcell[k] := blank;
       if tempcell = '           '
          then for k := 0 to (cellsize - 1) do
                    data[j, (arg1 + k)] := data[j, (arg2 + k)];
       j := j + 1
       end;  (* while *)
     for i := 0 to (cellsize - 1) do
         data[2, (arg1 + i)] := data[1, (arg2 + i)];
     for k := 1 to rownum do
```

```
        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;
write('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
       for k := 1 to cellsize do
           tempcell[k] := data[arg1, (k + j)];
       for k := (cellsize + 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];
```

```
        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
       mergecol(matrix, errorcount);
       done := true
       end  (* then *)
     else if (ch = 'R') or (ch = 'r')
             then begin
               mergerow(matrix, errorcount);
               done := true
               end  (* then *)
             else if (ch = 'N') or (ch = 'n')
                     then begin
                       done := true
                       end  (* then *)
                     else 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 I, 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 write('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;
      '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);
      'M':  merge(matrix, i, error);
      'm':  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  (* while *)
  end.  (* program permute *)
```

REFERENCES

Comrie, Bernard. 1980. Inverse Verb Forms in Siberia: Evidence from Chukchee, Koryak, and Kamchadal. Folia Linguistica Historica I:1.61-74.

Grogono, Peter. 1980. Programming in Pascal. Reading, Massachusetts: Addison-Wesley.

Krause, Scott R.. 1980. Topics in Chukchee Phonology and Morphology. University of Illinois Ph. D. Dissertation.

Miner, Kenneth L., and Barbara Lynn Taghva. 1983. Computerized Permutation of Pikean Field Matrices. Kansas Working Papers in Linguistics 8:1.97-106.

Miner, Kenneth L., and Dana Barrager. February, 1984. A Pascal Program for Manipulation of Pikean Field Matrices. Newsletter for the Society for the Study of the Indigenous Languages of the Americas III:1.12-13.

Pike, Kenneth L.. 1962. Dimensions of Grammatical Constructions. Language 38.221-323.

__________. 1963. Theoretical Implications of Matrix Permutation in Fore (New Guinea). Anthropological Linguistics 5.1-23.

__________. 1975. On Describing Languages. In Robert Austerlitz, ed., The Scope of American Linguistics. Lisse: Peter de Ridder Press pp.9-38.

__________, and Barbara E. Erikson. 1964. Conflated Field Structures in Potawatomi and Arabic. International Journal of American Linguistics 30.201-212.