The |STAT Handbook
Data Analysis Programs
on UNIX and MSDOS

Gary Perlman

This handbook was typeset by the author using the **troff** text formatting system on the UNIX operating system. The cover design, also by the author, shows data flowing though pipes and filters to produce displays, summaries, and inferences.

Dedicated to Caroline.

First Edition:	March 1986
Second Edition:	September 1986
Third Edition:	March 1987

Printed in the United States of America

|STAT Handbook
Table of Contents

CHAPTER 0

Preface

Purpose and Intended Audience of the Handbook

This handbook is meant to be an introduction to the |STAT programs. It is not written to teach students how to do data analysis, although it has been used as a supplementary text in courses. |STAT users should be familiar with using the hardware and utility programs (e.g., a text editor) on their systems.

Comparison With Other Packages

|STAT has advantages and disadvantages compared to other statistical packages. |STAT is not a comprehensive package because it was developed as needs arose. So there are deficits in many areas of analysis: no multivariate analysis other than regression, and only simple graphics. Independent of these limitations, the programs are not designed for use with large data sets or large values; the programs are usually adequate for data up to a few thousand points. Also, |STAT is unsupported, so if you have problems installing or using the programs, you may be on your own. Despite these limitations, |STAT provides you with most analyses reported in research. |STAT programs run on UNIX and MSDOS, operating systems popular in educational and research institutions, government, and industry. The liberal copyright of the programs allows free copies to be made for multiple machines provided the programs are not copied for material gain. |STAT programs integrate easily with other programs, and this makes it possible for new programs to be added later.

Distribution Conditions

References

Bradley, J. V. (1968) *Distribution-Free Statistical Tests*. Englewood Cliffs, NJ: Prentice-Hall.

Coombs, C. H., Dawes, R. M., & Tversky, A. (1970) *Mathematical Psychology: An Elementary Introduction*. Englewood Cliffs, NJ: Prentice-Hall.

Dixon, W. J. (1975) *BMD-P Biomedical Computer Programs*. Berkeley, CA: University of California Press.

Guilford, J. P., & Fruchter, B. (1978) *Fundamental Statistics in Psychology and Education*. (6th Edition). New York: McGraw-Hill.

Hays, W. L. (1973) *Statistics for the Social Sciences*. (2nd Edition). New York, NY: Holt Rinehart Winston.

Hemenway, K., & Armitage, H. (1984) Proposed Syntax Standard for UNIX System Commands. In *Summer USENIX Conference*. El Cerito, CA: Usenix Association. (Washington, DC.)

Keppel, G. (1973) *Design and Analysis: A Researcher's Handbook*. Englewood Cliffs, NJ: Prentice-Hall.

Kerlinger, F. N., & Pedhazur, E. J. (1973) *Multiple Regression in Behavioral Research*. New York, NY: Holt Rinehart Winston.

Kernighan, B. W., & Ritchie, D. M. (1979) *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall.

Nie, H. H., Jenkins, J. G., Steinbrenner, K., & Bent, D. H. (1975) *SPSS: Statistical Package for the Social Sciences*. New York: McGraw-Hill.

Perlman, G. (1980) Data Analysis Programs for the UNIX Operating System. *Behavior Research Methods & Instrumentation*, **12:5**, 554-558.

Perlman, G. (1982) Data Analysis in the UNIX Environment: Techniques for Automated Experimental Design Specification. In K. W. Heiner, R. S. Sacher, & J. W. Wilkinson (Eds.), *Computer Science and Statistics: Proceedings of the 14th Symposium on the Interface*.

Perlman, G., & Horan, F. L. (1986) Report on |STAT Release 5.1 Data Analysis Programs for UNIX and MSDOS. *Behavior Research Methods, Instruments, & Computers*, **18.2**, 168-176.

Perlman, G., & Horan, F. L. (1986) |STAT: Compact Data Manipulation and Analysis Programs for MSDOS and UNIX - A Tutorial Overview. Tyngsboro, MA: Wang Institute of Graduate Studies.

Ritchie, D. M., & Thompson, K. (1974) The UNIX Time-Sharing System. *Communications of the Association for Computing Machinery*, **17:7**, 365-375.

Ryan, T. A., Joiner, B. L., & Ryan, B. F. (1976) *MINITAB Student Handbook*. North Scituate, MA: Duxbury Press.

Siegel, S. (1956) *Nonparametric Methods for the Behavioral Sciences*. New York: McGraw-Hill.

CHAPTER 1

Introduction

The purpose, environment, and philosophy of the |STAT programs are introduced.

Section 1.1 Capabilities and Requirements

|STAT is a small statistical package I have developed on the UNIX operating system (Ritchie & Thompson, 1974) at the University of California San Diego and at the Wang Institute of Graduate Studies. Over twenty programs allow the manipulation and analysis of data and are complemented by this documentation and manual entries for each program. The package has been distributed to hundreds of UNIX sites and the portability of the package, written in C (Kernighan & Ritchie, 1979), was demonstrated when it was ported from UNIX to MSDOS at Cornell University on an IBM PC using the Lattice C compiler. This handbook is designed to be a tutorial introduction and reference for the most popular parts of release 5.3 of |STAT (January, 1987) and updates through February, 1987. Full reference information on the programs is found in the online manual entries and in the online options help available with most of the programs.

Dataset Sizes

|STAT programs have mostly been run on small datasets, the kind obtained in controlled psychological experiments, not the large sets obtained in surveys or physical experiments. The programs' performances on datasets with more than about 10,000 points is not known, and the programs should not be used for them.

System Requirements

The programs run on almost any version of UNIX. They are compatible with UNIX systems dating back to Version 6 UNIX (circa 1975). On MSDOS, the programs run on versions 2.X through 3.X. MSDOS versions earlier than 2.0 may not support the pipes often used with |STAT programs, and MSDOS version 4.0 formats are not compatible. Space requirements for MSDOS are about 1 megabyte of disk space, and at least 96 kilobytes of main memory. Hard disk storage is preferred, but not mandatory.

Availability and Distribution

Please take care to follow all the instructions below.

- Please indicate the items that you would like to order.
- Orders must be prepaid. Purchase orders are not acceptable.
- Make your check/(postal)money order payable to G. Perlman.
- Checks must be in US funds drawn on a US bank.
- Please include a delivery address label to speed service.
  International orders: please indicate your country name.

UNIX C Source Version of |STAT: $20/30
C Language Source Code & Online Manual Entries
1/2 inch 9 track mag tape, 1600 bpi tar format ($20)
1/4 inch cartridge tape, tar format ($30)
DOS Executable Version of |STAT: $15
Preformatted Manuals & Executables (without Source Code) ($15)
2S/2D (360K) DOS 5.25" floppy diskettes
HD (1.2M) DOS 5.25" or 3.5" floppies (by special request)
DOS Turbo C Source Code Version for |STAT: $10
Turbo C Language Source Code, Project Files, Online Manual
HD (1.2M) DOS 5.25"inch or 3.5" floppy diskette
Handbook (highly recommended for new users): $10
Examples, Ref. Materials, CALC & DM Manuals, Manual Entries
Typeset Manual (over 100 8.5 x 11 inch pages)
Prices include cost of media and airmail delivery worldwide.

Section 1.2 Design Philosophy

|STAT programs promote a particular style of data analysis. The package is interactive and programmable. Data analysis is typically not a single action but an iterative process in which a goal of understanding some data is approached. Many tools are used to provide several analyses of data, and based on the feedback provided by one analysis, new analyses are suggested.

The design philosophy of |STAT is easy to summarize. |STAT consists of several separate programs that can be used apart or together. The programs are called and combined at the command level, and common analyses can be saved in files using UNIX shell scripts or MSDOS batch files.

Understanding the design philosophy behind |STAT programs makes it easier to use them. |STAT programs are designed to be tools, used with each other, and with standard UNIX and MSDOS tools. This is possible because the programs make few assumptions about file formats used by other programs. Most of the programs read their inputs from the standard input (what is typed at the keyboard, unless redirected from a file), and all write to the standard output (what appears on the screen, unless saved to a file or sent to another program). The data formats are readable by people, with fields (columns) on lines separated by white space (blank spaces or tabs). Data are line-oriented, so they can be operated on by many programs. An example of a filter program on UNIX and MSDOS that can be used with the |STAT programs is the **`sort`** utility, which puts lines in numerical or alphabetical order. The following command sorts the lines in the file **`input`** and saves the result in the file **`sorted`**.

```
sort  <  input  >  sorted
```

The **`<`** symbol causes **`sort`** to read from **`input`** and the **`>`** causes **`sort`** to write to the file **`sorted`**. Because **`sort`** exists on UNIX and MSDOS, it is not necessary to duplicate its function in |STAT, which does not duplicate existing tools. (In all following examples, **`this font will be used`** to show text (e.g., commands and program names) that would be seen by people using the programs.

User efficiency is supported over program efficiency. That does not mean the programs are slow, but ease-of-use is not sacrificed to save computer time. Input formats are simple and readable by people. There is extensive checking to protect against invalid analyses. Output formats of analysis programs are designed to be easy to understand. Data manipulation programs are designed to produce uncluttered output that is ready for input to other programs.

On UNIX and MSDOS, a *filter* is a program that reads from the standard input, also called **`stdin`** (the keyboard, unless redirected from a file) and writes to the standard output, also called **`stdout`** (the screen, unless redirected to a file). Most |STAT programs are filters. They are small programs that can be used alone, or with other programs. |STAT users typically keep their data in a *master data file*. With data manipulation programs, extractions from the master data file are transformed into a format suitable for input to an analysis program. The original data do not change, but copies are made for transformations and analysis. Thus, an analysis consists of an extraction of data, optional transformations, and some analysis. Pictorially, this can be shown as:

```
data | extract | transform | format | analysis | results
```

where a copy a subset of the data has been extracted, transformed, reformatted, and analyzed by chaining several programs. Data manipulation functions, sometimes built into analysis programs in other packages, are distinct programs in |STAT. The use of pipelines, signaled with the pipe symbol, **|**, is the reason for the name |STAT.

Section 1.3 Table of |STAT Programs

|STAT programs are divided into two categories. There are programs for data manipulation: data generation, transformation, formatting, extraction, and validation. And there are programs for data analysis: summary statistics, inferential statistics, and data plots. The data manipulation programs can be used for tasks outside of statistics.

Data Manipulation Programs

`abut`	join data files beside each other
`colex`	column extraction/formatting
`dm`	conditional data extraction/transformation
`dsort`	multiple key data sorting filter
`linex`	line extraction
`maketrix`	create matrix format file from free-format input
`perm`	permute line order randomly, numerically, alphabetically
`probdist`	probability distribution functions
`ranksort`	convert data to ranks
`repeat`	repeat strings or lines in files
`reverse`	reverse lines, columns, or characters
`series`	generate an additive series of numbers
`transpose`	transpose matrix format input
`validata`	verify data file consistency

Data Analysis Programs

`anova`	multi-factor analysis of variance
`calc`	interactive algebraic modeling calculator
`contab`	contingency tables and chi-square
`desc`	descriptions, histograms, frequency tables
`dprime`	signal detection d’ and beta calculations
`features`	display features of items
`oneway`	one-way anova/t-test with error-bar plots
`pair`	paired data statistics, regression, scatterplots
`rankind`	rank order analysis for independent conditions
`rankrel`	rank order analysis for related conditions
`regress`	multiple linear regression and correlation
`stats`	simple summary statistics
`ts`	time series analysis and plots

Section 1.4 Table of UNIX and MSDOS Utilities

The UNIX and MSDOS environments are similar, at least as far as |STAT is concerned, but many command names differ. The following table shows the pairing of UNIX names with their MSDOS equivalents.

UNIX	**MSDOS**	**Purpose**
`cat`	`type`	print files to stdout
`cd,pwd`	`cd`	change/print working directory
`cp`	`copy`	copy files
`diff`	`comp`	compare and list file differences
`echo`	`echo`	print text to standard output
`grep`	`find`	search for pattern in files
`ls`	`dir`	list files in directory
`mkdir`	`mkdir`	create a new directory
`more`	`more`	paginate text on screen
`mv`	`rename`	move/rename files
`print`	`print`	print files on printer
`rm`	`del,erase`	remove/delete files
`rmdir`	`rmdir`	remove an empty directory
`sort`	`sort`	sort lines in files
`shell-script`	`batch-file`	programming language
`$1,$2`	`%1,%2`	variables
`/dev/tty`	`con`	terminal keyboard/screen
`/dev/null`	`nul`	empty file, infinite sink

CHAPTER 2

Annotated Example

A concrete example with several |STAT programs is worked in detail. The example shows the style of analysis in |STAT. New users of |STAT should not try to understand all the details in the examples. Details about all the programs can be found in on-line manual entries and more examples of program use appear in following chapters. Explanations about features common to all |STAT programs can be found in the next chapter.

Section 2.1 A Familiar Problem

To show the |STAT style of interactive data analysis, I will work through a concrete example. The example is based on a familiar problem: grades in a course based on two midterm exams and a final exam. Scores on exams will be broken down by student gender (male or female) and by the lab section taught by one of two teaching assistants: John or Jane. Assume the following data are in the file **`exam.dat`**. Each line in the file includes a student identification number, the student's section's teaching assistant, the student's gender, and the scores (out of 100) on the two midterm exams and the final.

```
S-1     john    male    56      42      58
S-2     john    male    96      90      91
S-3     john    male    70      59      65
S-4     john    male    82      75      78
S-5     john    male    85      90      92
S-6     john    male    69      60      65
S-7     john    female  82      78      60
S-8     john    female  84      81      82
S-9     john    female  89      80      68
S-10    john    female  90      93      91
S-11    jane    male    42      46      65
S-12    jane    male    28      15      34
S-13    jane    male    49      68      75
S-14    jane    male    36      30      48
S-15    jane    male    58      58      62
S-16    jane    male    72      70      84
S-17    jane    female  65      61      70
S-18    jane    female  68      75      71
S-19    jane    female  62      50      55
S-20    jane    female  71      72      87
```

We are interested in computing final grades based on the exam scores, and comparing the performances of males versus females, and of the different teaching assistants. The following analyses can be tried by typing in the above file and running the commands in the examples. Minor variations on the example commands will help show how the programs work.

Section 2.2 Computing Final Scores

Computing final scores is easy with the data manipulation program **dm**. Assume that the first midterm is worth 20 percent, the second 30 percent, and the final exam, 50 percent. The following command tells **dm** to repeat each input line with **INPUT**, and then print the weighted sum of columns 4, 5, and 6, treated as *numbers*. To print numbers, **dm** uses an **x** before the column number. The input to **dm** is read from the file **exam.dat** and the result is saved in the file **scores.dat**. Once all the original data and the final scores are in **scores.dat**, only that file will be used in following analyses.

```
dm INPUT ".2*x4 + .3*x5 + .5*x6" < exam.dat > scores.dat
```

The standard input is redirected from the file **exam.dat** with the **<** on the command line. Similarly, the standard output, which would ordinarily go to the screen, is redirected to the file **scores.dat** with the **>** on the command line. The second expression for **dm** is in quotes. This allows the insertion of spaces to make the expression more readable, and to make sure that any special characters (e.g., ***** is special to UNIX shells) are hidden from the command line interpreter. The output from the above command, saved in the file **scores.dat**, would begin with the following.

```
S-1     john    male    56      42      58      52.8
S-2     john    male    96      90      91      91.7
S-3     john    male    70      59      65      64.2
S-4     john    male    82      75      78      77.9
S-5     john    male    85      90      92      90
S-6     john    male    69      60      65      64.3
etc.
```

This could be sorted by final grade by reversing the columns and sending the output to the standard UNIX or MSDOS **sort** utility program using the ‘‘pipe’’ symbol **|**.

```
reverse -f < scores.dat | sort
```

The above command would produce the following output.

```
27.1    34      15      28      male    jane    S-12
40.2    48      30      36      male    jane    S-14
52.8    58      42      56      male    john    S-1
54.7    65      46      42      male    jane    S-11
54.9    55      50      62      female  jane    S-19
 ...
79.3    87      72      71      female  jane    S-20
82.1    82      81      84      female  john    S-8
90      92      90      85      male    john    S-5
91.4    91      93      90      female  john    S-10
91.7    91      90      96      male    john    S-2
```

To restore the order of the fields, **reverse** could be called again. Another way, more efficient, would be to use the **dsort** filter to sort based on column 7:

```
dsort 7 < scores.dat
```

Section 2.3 Summary of Final Scores

desc prints summary statistics, histograms, and frequency tables. The following command takes the final scores (the weighted average from the previous section).

```
dm  s7  <  scores.dat
```

Summary order statistics are printed with the **-o** option and the distribution is tested against the passing grade of 75 with the **-t 75** option. **desc** makes a histogram (the **-h** option) with 10 point intervals (the **-i 10** option) starting at a minimum value of 0 (the **-m 0** option).

```
dm  s7  <  scores.dat | desc  -o  -t 75  -h  -i 10  -m 0

------------------------------------------------------------
 Under Range    In Range  Over Range     Missing         Sum
           0          20           0           0    1359.200
------------------------------------------------------------
        Mean      Median    Midpoint   Geometric    Harmonic
      67.960      68.750      59.400      65.564      62.529
------------------------------------------------------------
          SD   Quart Dev       Range     SE mean
      16.707      10.575      64.600       3.736
------------------------------------------------------------
     Minimum  Quartile 1  Quartile 2  Quartile 3     Maximum
      27.100      57.450      68.750      78.600      91.700
------------------------------------------------------------
        Skew     SD Skew    Kurtosis     SD Kurt
      -0.586       0.548       2.844       1.095
------------------------------------------------------------
   Null Mean           t    prob (t)           F    prob (F)
      75.000      -1.884       0.075       3.551       0.075
------------------------------------------------------------
       Midpt    Freq
       5.000       0
      15.000       0
      25.000       1 *
      35.000       0
      45.000       1 *
      55.000       4 ****
      65.000       5 *****
      75.000       5 *****
      85.000       2 **
      95.000       2 **
```

Section 2.4 Predicting Final Exam Scores

The next analysis predicts final exam scores with those of the two midterm exams. The **regress** program assumes its input has the predicted variable in column 1 and the predictors in following columns. **dm** can extract the columns in the correct order from the file **scores.dat**. The command for **dm** looks like this.

```
dm x6 x4 x5 < scores.dat
```

The output from **dm** looks like this.

```
58    56    42
91    96    90
65    70    59
78    82    75
92    85    90
65    69    60
60    82    78
```
etc.

This is the correct format for input for **regress**, which is given mnemonic names for the columns. The **-e** option tells **regress** to save the regression equation in the file **regress.eqn** for a later analysis.

```
dm x6 x4 x5 < scores.dat | regress -e final midterm1 midterm2
```

The output from **regress** includes summary statistics for all the variables, a correlation matrix (e.g., the correlation of **midterm1** and **midterm2** is .9190), the regression equation relating the predicted variable, and the significance test of the multiple correlation coefficient. The squared multiple correlation coefficient of 0.7996 shows a strong relationship between midterm exams and the final.

```
Analysis for 20 cases of 3 variables:
Variable        final   midterm1   midterm2
Min           34.0000    28.0000    15.0000
Max           92.0000    96.0000    93.0000
Sum         1401.0000  1354.0000  1293.0000
Mean          70.0500    67.7000    64.6500
SD            15.3502    18.6720    20.4303

Correlation Matrix:
final          1.0000
midterm1       0.7586     1.0000
midterm2       0.8838     0.9190     1.0000
Variable        final   midterm1   midterm2

Regression Equation for final:
final  =  -0.2835 midterm1  +  0.9022 midterm2  +  30.9177

Significance test for prediction of final
    Mult-R  R-Squared      SEest    F(2,17)   prob (F)
    0.8942     0.7996     7.2640    33.9228     0.0000
```

Predicted Plot

We can look at the predictions from the regression analysis. From the analysis above, the file **regress.eqn** contains a regression equation for **dm**.

```
s1
(x2 * -0.283512...) + (x3 * 0.902182...) + 30.9177...
```

Extra precision is used in **regress.eqn**, compared to the equation in the output from **regress** to allow more accurate calculations. These two expressions, one on each line, are the obtained and predicted final exam scores, respectively. To plot these against each other, we duplicate the input used to **`regress`**, and process **`regress`**'s output with **`dm`**, reading its expressions from the expression file **`regress.eqn`** that follows the letter **`E`**. The result is passed through a pipe to the paired data analysis program **`pair`** with the plotting option **`-p`**, options to control the height and width of the plot, the **`-h`** and **`-w`** options, and **`-x`** and **`-y`** options to label the plot.

```
dm x6 x4 x5 < scores.dat | dm Eregress.eqn |
      pair -p -h 10 -w 30 -x final -y predicted
```

```
|------------------------------|89.3045
|                             3|
|                   1    1     |
|             1   1   11  1 1  |
|                              |
|              1 2 1           |predicted
|          1     1             |
|            1                 |
|       1                      |
|                              |
|1                             |
|------------------------------|36.5121
34.000                    92.000
        final  r= 0.894
```

Residual Plot

To plot the residuals (deviations) from prediction, you can run the data through another pass of **`dm`** to subtract the predicted scores from the obtained. Note that **`r`** must be zero.

```
dm x6 x4 x5 < scores.dat | dm Eregress.eqn | dm x2 x1-x2 |
      pair -p -h 10 -w 30 -x predicted -y residuals
```

```
|------------------------------|11.2546
|                     11       |
|                           1  |
|         1   1   1    1      1|
|      1        1        1    1|
|1               1      1      |residuals
|            1    1            |
|                        1     |
|                       1      |
|                              |
|                       1      |
|------------------------------|-18.0399
36.512                    89.304
      predicted  r= 0.000
```

Section 2.5 Failures by Assistant and Gender

Now suppose the passing grade in the course is 75. To see how many people of each sex in the two sections passed, we can use the **contab** program to print contingency tables. First **dm** extracts the columns containing teaching assistant, gender, and the final grade (the weighted average computed earlier). Rather than include the final grade, a label indicating pass or fail is added, as appropriate.

```
dm  s2  s3  "if x7 >= 75 then 'pass' else 'fail'"  1  <  scores.dat
```

The huge third expression says ‘‘if the final grade is greater than or equal to 75, then insert the string **pass**, else insert the string **fail**.’’ Such expressions can be placed in files rather than be typed on the command line, and usually **dm** is used for simpler expressions. The fourth expression is the constant **1** used to tell **contab** that there was one replication for each combination of factor levels. Part of the output from **dm** follows.

```
john    male    fail    1
john    male    pass    1
john    male    fail    1
    ...
jane    female  fail    1
jane    female  fail    1
jane    female  pass    1
```

This is used as input to **contab**, which is given mnemonic factor names.

```
dm  s2  s3  "if x7 >= 75 then 'pass' else 'fail'"  1  <  scores.dat |
      contab assistant gender success count
```

Parts of the output from this command follow. First, there is a summary of the input, which contained three factors, each with 2 levels, and a sum of observation counts.

```
FACTOR:  assistant     gender    success      count
LEVELS:          2          2          2         20
```

The first contingency table does not provide new information. It shows that both Jane’s section and John’s section had 6 male and 4 female students.

```
SOURCE: assistant gender
            male  female  Totals
john           6       4      10
jane           6       4      10
Totals        12       8      20
```

The second contingency table tells us that 12 of 20 students failed the course--4 in John’s section and 8 in Jane’s. A significance test follows, and the warning about small expected frequencies suggests that the chi-square test for independence might be invalid. No matter, the Fisher exact test applies because we are dealing with a 2x2 table and total frequencies less than 100. It does not show a significant association of factors (ie. Jane’s section did not do significantly better than John’s).

```
SOURCE: assistant success
            fail    pass  Totals
john           4       6      10
jane           8       2      10
Totals        12       8      20
```

```
Analysis for assistant x success:
NOTE: Yates’ correction for continuity applied
WARNING: 2 of 4 cells had expected frequencies < 5
chisq       1.875000    df   1    p  0.170904
Fisher Exact One-Tailed Probability  0.084901
Fisher Exact Two-Tailed Probability  0.169802
phi Coefficient == Cramer’s V        0.306186
Contingency Coefficient              0.292770
```

The third contingency table shows that 8 male students and 4 female students failed the course.

```
SOURCE: gender success
            fail    pass  Totals
male           8       4      12
female         4       4       8
Totals        12       8      20
```

The final table, the three-way interaction, shows all the effects listed above, but no significance test is computed by **contab**. Some hints about the reason for the poorer performance of Jane’s section follow from the next section’s analysis of variance.

```
SOURCE: assistant gender success
assistan  gender success
    john    male    fail       3
    john    male    pass       3
    john  female    fail       1
    john  female    pass       3
    jane    male    fail       5
    jane    male    pass       1
    jane  female    fail       3
    jane  female    pass       1
```

Section 2.6 Effects of Assistant and Gender

We now want to compare the performance of the two teaching assistants and of male versus female students. We are interested to see how an assistant's students progress through the term. **anova**, the analysis of variance program, is the program to analyze these data, but we have to get the data into the correct format for input to **anova**. **anova** assumes that there is only one datum per line, preceded by the levels of factors under which it was obtained. This is unlike the format of **scores.dat**, which has the three exam scores after the student number, teaching assistant name, and gender. Several transformations are needed to get the data in the correct format. As an example, the data for student 1:

```
S-1      john     male     56       42       58
```

must be transformed to:

```
S-1      john     male     m1       56
S-1      john     male     m2       42
S-1      john     male     final    58
```

This is made up of three replications of the labels with new labels, **m1**, **m2**, and **final**, for the exams inserted. First, **dm** extracts and inserts the desired information. The result is a 15 column output, one for each expression. Note that on UNIX, it is necessary to quote the quotes of the labels for the exam names. To insert the newlines, so that each datum is on one line, the program **maketrix** reformats the input to **anova** into 5 columns. Finally, mnemonic labels for factor names are given to **anova**.

```
dm  s1  s2  s3  "'m1'"     s4 ...
    s1  s2  s3  "'m2'"     s5 ...
    s1  s2  s3  "'final'"  s6 < scores.dat |
    maketrix 5 | anova student assistant gender exam score
```

Only parts of the output are shown below. First, John's students did better than Jane's students (F(1,16)=8.311, p=.011).

```
john       76.7000
jane       58.2333
```

Female students scored better than males, although the effect is not statistically significant (F(1,16)=3.102, p=.097).

```
male       62.8611
female     74.3750
```

There was no interaction between these two factors (F(1,16)=.289), but there were some interactions between section assistant and gender and the different exam grades. If we look at the interaction of section assistant and exam, we get a better picture of the performances of John and Jane.

```
SOURCE: assistant exam
assista exam       N       MEAN        SD         SE
john    m1        10    80.3000    11.9355     3.7743
john    m2        10    74.8000    16.3761     5.1786
john    final     10    75.0000    13.4247     4.2453
jane    m1        10    55.1000    15.5167     4.9068
jane    m2        10    54.5000    19.5973     6.1972
jane    final     10    65.1000    16.2101     5.1261
```

This is the first full cell-means table shown. It contains the names of factors and levels, cell counts, means, standard deviations, and standard errors. The results show that John's students started higher than Jane's (80.3 versus 55.1), and that over the term, Jane's students improved while John's got worse. The significance test for the interaction looks like this.

```
SOURCE          SS   df         MS       F      p
=================================================
ae        610.4333    2   305.2167   9.502  0.001 ***
es/ag    1027.8889   32    32.1215
```

A Scheffe confidence interval around the difference between two means of this interaction can be found with the following formula.

```
sqrt (df1 * critf * MSerror * 2 / N)
```

`df1` is the degrees of freedom numerator, `critf` is the critical F-ratio given the degrees of freedom and confidence level desired, `MSerror` is the mean-square error for the overall F-test, and `N` is the number of scores going into each cell. The critical F ratio for a 95% confidence interval based on 2 and 32 degrees of freedom can be found with the `probdist` program.

```
probdist  crit  F  2  32  .05
3.294537
```

Then, the calculator program `calc` can be used interactively to substitute the values.

```
CALC: sqrt (2 * 3.294537 * 32.1215 * 2 / 10)
sqrt(((((2 * 3.29454) * 32.1215) * 2) / 10)) = 6.50617
```

Any difference of two means in this interaction greater than 6.5 is significant at the .05 level.

There was a similar pattern of males versus females on the three exams. Males started out lower than females, and males improved slightly while females stayed about the same.

```
SOURCE: gender exam
gender  exam       N       MEAN        SD         SE
male    m1        12    61.9167   20.7822     5.9993
male    m2        12    58.5833   22.5931     6.5221
male    final     12    68.0833   17.1329     4.9459
female  m1         8    76.3750   11.1475     3.9413
female  m2         8    73.7500   13.1557     4.6512
female  final      8    73.0000   12.7167     4.4960
```

After the cell means in the output from `anova` is a summary of the design, followed by an F-table, parts of which were seen above.

```
FACTOR:    student  assistant     gender       exam      score
LEVELS:         20          2          2          3         60
TYPE  :     RANDOM    BETWEEN    BETWEEN     WITHIN       DATA
```

The results of the analysis show that John’s section did better than Jane’s. That must be qualified because it seems that Jane’s students may not have been as good as John’s. To Jane’s credit, her students improved more than John’s during the term.

CHAPTER 3

Conventions

Features common to all the |STAT programs are covered. This information makes it easier to learn about new |STAT programs, and serves as a reference for experienced users.

Section 3.1 Command Line Interpreters

|STAT analyses consist of a series of commands, each on a single line, hence the name *command line*. Commands are typed by users into a command line interpreter, itself a program that runs the commands typed in. On MSDOS, there is no special name given to the command line interpreter. On UNIX, the command line interpreters are called *shells*, and there are several of them. Users are expected to know the conventions of their command line interpreters. Some of the examples in this handbook and in the manual entries will not work because of differences in how command lines are formatted. Minor modifications to the examples are sometimes needed.

Some command line interpreters support in-line editing, which is useful when running |STAT analyses because data analysis is an iterative process in which minor changes in analyses, and hence commands, are common.

Special Characters

Command line interpreters have special characters to perform special tasks. On both MSDOS and UNIX, there are special characters for file input, output, and pipe redirection:

<	redirect standard input from the following file
>	redirect standard output to the following file
|	redirect standard output to the following command

UNIX and MSDOS both have patterns (sometimes called ‘‘wildcards’’) to match file names. For example, **`*.c`** matches all files that end with a **`c`** suffix. Also, the **`?`** can be used in patterns to match any one character. An important difference between UNIX and MSDOS command line interpreters is that on UNIX, the pattern matching is part of the shell, and so is available to every program, while on MSDOS, it is part of only some programs.

It is sometimes necessary to quote the special meaning of special characters so that they are not seen by the command line interpreter. For example, an expression for **`dm`** might contain the symbols **`*`** for multiplication or **`<`** for comparison. Both these characters are special to UNIX shells, while only **`<`** is special to MSDOS. The blank space and tab characters are special on both UNIX and MSDOS, and are used to separate command line arguments. Special characters can be quoted by enclosing command line arguments in double quotes. For example, **`dm`** expressions may contain special characters, and strings may contain spaces.

```
dm  "if x1 > 10 then 'Large number on line:' else SKIP"  INLINE
```

Section 3.2 Command Formats

|STAT programs are run on UNIX and MSDOS by typing the name of the program, program options, and program operands (e.g., expressions or file names). Program names, options, and operands, are separated or *delimited* by blank space. On UNIX, program names are lower case, while on the case-insensitive MSDOS, they are always upper case, although users can type the names in lower case. Program options and operands can be complex, so it is sometimes useful to insert spaces into an option value or an operand, either to modify the output or to make the command line more readable. This is done by quoting (with double quotes) the parts that should be kept together.

Simple Commands

A simple command consists of a program name, program options delimited with minus signs, and program operands, such as file or variable names. Here are some examples:

```
dm  x1+x2  x3/x4
calc  model
regress  -p  age  height  weight
desc  -h  -i 1  -m 0  -cfp
series  1  100  .5
probdist  random  normal  100
```

Pipelines of Commands

A pipeline of commands is a series of simple commands joined by the pipe symbol, **|**. In a pipeline, the output from one simple command is the input to the next command in the pipeline. The following pipeline creates a series of numbers from 1 to 100, transforms it by using the **dm** logarithm function, and then makes a histogram of the result.

```
series 1 100  |  dm logx1  |  desc -h
```

The following pipeline abuts three files beside one another, and passes the result to the **regress** program, which prints their correlation matrix.

```
abut age height weight  |  regress -r age height weight
```

Note that the operands to **abut** are file names, while those for **regress** are variable names, which could be different if desired. If they were always supposed to be the same, then this constraint could be encoded in a shell script or batch file.

Batch Files and Shell Scripts

Because the |STAT programs work well together, and because most data analysis is routine, it is often advantageous to save a series of commands in a file for later analyses. Both UNIX and MSDOS support this, MSDOS with *batch* files and UNIX with *shell* scripts. Batch files and shell scripts also support variables, some set by command line calls and some set inside the command file. They provide |STAT with a simple but effective programming facility.

Section 3.3 Program Options

Program options allow the user to control how a program works by requesting custom or extra analysis. Without options, |STAT programs provide the simplest or most common behavior by default. Program options conform to the standard UNIX option parsing convention (Hemenway & Armitage, 1984) by using the **`getopt`** option parser. In this standard, all program options are single characters preceded by a minus sign. For example, **`-a`** and **`-X`** are both options. All program options must precede operands (such as file names, variable names, or expressions). Some options require values, and these should follow the option. For example, the **`pair`** plotting function allows setting the height of the plot with the **`-h`** option: **`-h 30`** would set the plot height to 30 lines. There should be a space between an option and its value. Options that do not take values (logical options) can be grouped or ‘‘bundled’’ to save typing. For example, the descriptive statistics program, **`desc`**, has options for requesting a histogram, a table of frequencies, and a table of proportions. These can be requested with the bundle of options: **`-hfp`** instead of the longer: **`-h -f -p`**.

There are some special conventions used with the **`getopt`** option parser. A double dash, **`--`**, by itself signals the end of the options, which can be useful when the first operand begins with **`-`** and it would be misinterpreted as an option. For programs that take files as operands (e.g., **`abut`**, **`calc`**), a solitary **`-`** means to read from the standard input, which can be useful to insert the output of a pipeline in a set of files. For example, the **`abut`** program can read several files with the standard input inserted with the following command line.

```
series 1 20  |  abut file1 file2 - file3
```

The output would be four columns, the third of which would be the series 1 to 20.

The same options can usually be specified more than once on a command line. For logical options (those that turn on or off a feature), repetition usually has no effect. For options that take values, such as the width of a plot, respecifying an option resets it to a new value. Exceptions to these rules for specific options are mentioned in program manual entries.

Table of Option Rules

`-x`	options are single letters preceded by minus
`-h 30`	option values must follow the option after a space
`-nve`	logical options can be bundled
`--`	signals the end of the options
`-`	insert standard input to operands of file-reading program

Standard Options

All |STAT programs using the standard option parser, **`getopt`**, have standard options to get information online. The information reported by the program is always accurate, while the printed documentation may not be up to date, or may not apply to the particular version (e.g., limits on MSDOS may be smaller than on UNIX).

`-L`	prints a list of program limits
`-O`	prints a summary of program options
`-V`	prints version information

Section 3.4 File Inputs and Outputs

Most of the |STAT programs are filters. That means they read from the standard input and write to the standard output. By default, the standard input is the keyboard, and the standard output is the screen. The standard input and output can independently be ‘‘redirected’’ using the special characters: **<**, to redirect the standard input from an immediately following file name, **>**, to redirect the standard output to a file. Also, the pipe character **|**, can connect the output from one program to the input to another. (Some of these features are not available on early versions of MSDOS (before version 2.0).) The following command says for the **anova** program to read from the file **anova.in**.

```
anova  <  anova.in
```

The output would go to the screen, by default. The following command saves the above output to the file **anova.out**.

```
anova  <  anova.in  >  anova.out
```

Never do this:

```
anova  <  data  >  data          # Never Do This!
```

Never make the input file the same as the output file, or you will lose the file; the output file is created (and zeroed) by the command line interpreter before the input file is read. Temporary files should be used instead. Here is an example of output redirection to save 50 random normal numbers.

```
probdist random normal 50  >  numbers
```

In English, this is read: ‘‘A random sample of 50 numbers is created and saved in the file **numbers**. This file of numbers could be used as input to the descriptive statistics program, **desc**. The intermediate file, **numbers**, could be avoided by using a pipeline.

```
probdist random normal 50  |  desc
```

To save the result of the above analysis in a file called **results**, output redirection would be used.

```
probdist random normal 50  |  desc  >  results
```

Although pipes are supported on MSDOS, they are not efficient and they require that there is enough space for temporary files to hold the contents of the pipes (temporary files with names like **PIPE%1.$$$**). This can make input and output redirection without pipes a better choice for speed, especially in command scripts, called ‘‘batch files’’ on MSDOS.

Keyboard Input

If a program is expecting input from the keyboard (ie. the standard input has not been redirected from a file or pipe), a prompt will be printed on the screen. Often, input from the keyboard is a mistake; most people do not type directly into an analysis program but prepare a file with their preferred editor and use that file as input.

```
prompt: desc
desc: reading input from terminal:
```
user types input, followed by end of file: `^D` *on UNIX,* `^Z` *on MSDOS*

In all examples of keyboard input, the sequence **^X** will be used for control characters like control-x (hold down the **CTRL** key and type the letter **X**). On UNIX, end of input from the keyboard is signaled by typing **^D**. MSDOS users type **^Z**.

Section 3.5 Input Formats

|STAT programs have simple input formats. Program input is read until the end of file, *EOF*, is found. End of file in disk files is done by the system; no special marking characters are needed nor allowed.

Input **fields** (visibly distinguishable **words**) are separated by **whitespace** (blank spaces, tabs, newlines). For most programs, fields in lines with embedded spaces can be enclosed by single or double quotes. Most |STAT analysis programs ignore blank input lines used to improve the human-readability of the data. However, blank lines are meaningful to some data manipulation programs, so when there are unexpected results, it is often instructive to run a file through **`validata`**.

Suggestion: Staged Analysis

It is usually a good idea to build a complex command, such as a pipeline, in stages. At each stage, a quick visual inspection of the output catches most errors you might make.

Data Types

|STAT programs recognize several types of data: label and variable names, numbers (integers and real numbers), and some programs can deal with missing values, denoted by **`NA`**. Label and variable names begin with an alphabetic character (a-z or A-Z), and can be followed by any number of alphanumerics (a-z, A-Z, 0-9) and underscores. There are three types of numbers: integers, real numbers with a decimal point, and numbers in exponential scientific notation. Integers are positive or negative numbers with no decimal point, or if they have a decimal point, they have no non-zero digits after the decimal point. Exponential notation numbers are numbers of the form **`xxx.yyyEzz`**. They may have digits before an optional decimal point or after it, and the number after the **`E`** or **`e`** is a power of ten multiplier. For example, **`1.2e-6`** is 1.2 times the inverse of one million.

Caveat: Appearances Can Be Deceiving

Inputs that look like they line up might not appear so to |STAT programs. For example, the following data might appear to have four columns, but have a variable number. Also, the columns that look like they line up to a person, do not line up to |STAT programs.

```
a     b     c     d
e           f     g
h     i           j
```

Here is how |STAT programs see this input:

```
a     b     c     d
e     f     g
h     i     j
```

This difference could be found with the **`validata`** utility program, which would report for both formats above:

```
validata: Variable number of columns at line 2
Col   N  NA alnum alpha   int float other  type   min   max
  1   3   0     3     3     0     0     0 alnum     0     0
  2   3   0     3     3     0     0     0 alnum     0     0
  3   3   0     3     3     0     0     0 alnum     0     0
  4   1   0     1     1     0     0     0 alnum     0     0
```

Section 3.6 Limits and Error Messages

There is a system-dependent limit on the count of characters in an input line: on small systems, 512 characters, and on large ones, 1024. Many programs use dynamic memory allocation so the memory available on a machine will determine the size of data sets that can be analyzed. Integer overflow is not checked, so numbers like data counts are limited on 16 bit machines to 32767; in practice, this has not presented problems. All calculations are done with double precision floating point numbers, but overflow (exceeding the maximum allowed double precision number, about 10 to the 38th power) and underflow (loss of precision of a tiny non-zero result being rounded to 0.0) are not checked. Program specific limits can be found in most programs with the **`-L`** option. The programs are not robust when used on highly variable data (differences of several orders of magnitude), very large numbers, or large datasets (more than 10,000 values).

All error and warning messages (1) identify the program detecting the problem (useful when pipelines or command scripts are used), (2) print diagnostic information, (3) sound a bell, and for errors, (4) cause an exit. All error and warning messages are printed on the diagnostic output (that is **`stderr`** for C lovers), so they will be seen even if the standard output is redirected to a file. All |STAT programs exit with a non-zero exit status on error and a zero exit status after a successful run.

Common Error Messages

Some errors and messages are common to several programs. They are explained below. Other messages should be self-explanatory.

`Not enough (or no) input data`
: There were no data points read, or not enough to make sense

`Too many xxxx’s; at most N allowed`
: Too many of something were in the input (e.g., columns or variables)

`Cannot open ’file’`
: The named file could not be opened for reading

`No storage space left for xxxx`
: The program has run out of dynamic memory for internal storage

`´string’ (description) is not a number`
: The described object whose input value was ’string’ was non-numerical

`N operand(s) ignored on command line`
: Operands (e.g., files) on the command line are ignored by this program

`VALUE is an illegal value for the TYPE`
: The provided value was out of the legal range for the given type

`Ragged input file`
: The program expects a uniform number of input columns

Section 3.7 Manual Entries

|STAT manual entries contain detailed information about each of the programs. They describe the effects of all the options.

On-Line Manuals

On UNIX systems, the manual entries for |STAT programs are available online with the **manstat** program. UNIX system administrators might prefer to install the |STAT manuals in a public place, so they might be available with the standard UNIX **man** program. On MSDOS systems, manual entries might be available online with a batch file that types pre-formatted manuals. The following will print the online manual for the **anova** program.

```
manstat anova
```

Most programs print a summary of their options with the **-O** option. The following will print a summary of the options available with the **desc** descriptive statistics program.

```
desc -O
```

UNIX Manual Conventions

UNIX manual entries are often considered cryptic, especially for new users. It helps to know the conventions used in writing manual entries. In the following table, the contents of the different manual entry sections are summarized.

ALGORITHMS	sources or descriptions of algorithms
BUGS	limitations or known deficiencies in the program
DESCRIPTION	details about the workings of the program, and information about operands
EXAMPLES	examples of command lines showing expected use of the program
FILES	files used by the program (e.g., temporary files)
LIMITS	limits built into the program should be determined with the -L option
NAME	the name and purpose of the program
OPTIONS	detailed information about command line options (see the -O option)
SYNOPSIS	a short summary of the option/operand syntax for the program (items enclosed in square brackets are optional)

CHAPTER 4

Data Manipulation

All data manipulation programs are introduced, showing some of their options. Full documentation is in the manual entries. |STAT data manipulation tools allow users to generate, transform, format, extract, and validate data. **dm**, the data manipulator, is the most important tool for use with other |STAT programs. A detailed manual for **dm** is the last section of this chapter.

There are several classes of data manipulation programs. **Generation** programs produce more data than their inputs by repeating data, numbering data, or by creating new data. **Transformation** programs allow algebraic conversion of data. **Formatting** programs change the shape or order of the data. **Extraction** programs produce subsets of datasets. **Validation** programs check the consistency, data types, and ranges of data.

Section 4.1 Data Generation/Augmentation

repeat: repeat a string or file

`repeat` can repeat strings or lines in a file as many times as requested. It helps generate labels for datasets, or feed a program like **`dm`** that needs input to produce output. The following will repeat the file **`data`** 10 times.

```
repeat  -n 10  data
```

The following will repeat its input series of 20 numbers 15 times.

```
series  1 20  |  repeat  -n 15
```

Strings can be repeated using the **`echo`** command. The following will repeat the string **`hello`** 100 times.

```
echo hello | repeat -n 100
```

series: generate a linear series

`series` generates a linear series of numbers between two values. By default, its values change by units, but this can be modified. The following produces a series of 10 numbers, 1 to 10, one per line.

```
series 1 10
```

The following produces the same series, but in reverse order; the start of the series can be greater than the end.

```
series 10 1
```

Non-integral series can be created by supplying an optional increment.

```
series 0 1 .1
```

produces the series:

```
0  .1  .2  .3  .4  .5  .6  .7  .8  .9  1
```

except that each value is on its own line. The output from series can be transformed with **`dm`** to produce other than linear series. Here is an exponential series:

```
series 1 10 | dm "exp(x1)"
```

probdist: generate random numbers

`probdist` can generate random numbers for several probability distributions. The following will generate 100 random numbers from the uniform distribution (between 0 and 1).

```
probdist random uniform 100
```

This can be transformed using **`dm`** to get random numbers with other ranges. The following will produce 100 random integers uniformly distributed between 10 and 29.

```
probdist random uniform 100 | dm "floor(x1*20+10)"
```

The following generates numbers from a one-trial binomial distribution with probability 0.5.

```
probdist random uniform 100 | dm "if x1 > .5 then 1 else 0"
```

`probdist` also has a binomial distribution built in, so the following would be equivalent to the previous example:

```
probdist rand binomial 1 1/2 100
```

The random number generator can be seeded. The following will seed the random number generator with 143 and generate 100 normally distributed z values.

```
probdist -s 143 random normal 100
```

The seeding option is useful when a random sequence must be repeated. The random normal numbers have a mean of 0 and a standard deviation of 1, so **dm** can help create different random normal distributions. The following samples a normal distribution with mean 100 and standard deviation 15.

```
probdist random normal 100  |  dm  "x1*15+100"
```

abut: number lines, recycle files

abut can number input lines in files using the **-n** option, or cycle through input files as many times as is necessary to match the length of longer files. The latter case is common in creating input files for programs like **anova** and **contab**, which have input data tagged with regular patterns of labels.

File1	*File2*	*Data*
large	easy	12
small	easy	23
	hard	34
	hard	45
		56
		67
		78
		89

For the above input file configuration, the command

```
abut -nc File1 File2 Data
```

would produce the following by recycling the smaller files.

```
1           large       easy        12
2           small       easy        23
3           large       hard        34
4           small       hard        45
5           large       easy        56
6           small       easy        67
7           large       hard        78
8           small       hard        89
```

dm: number lines

dm can number its input lines with its special variables **INLINE**, which always contains the input line number, and **INPUT**, which always contains the current input line.

```
dm INLINE INPUT < data
```

Section 4.2 Data Transformation

dm: conditional algebraic combinations of columns

dm can produce algebraic combinations of columns. The following command reads from **data** and produces the ratio of columns 2 and 1 with column 3 added on.

```
dm  x2/x1+x3  <  data
```

Transformations can be based on conditions. For example, if **x1**, the value in column 1, in the above example is 0, then **dm** will exit after producing an error message like:

```
dm: division by zero. input line 12  expr[1].
```

To avoid this problem, the following will do the division only if **x1** is non-zero.

```
dm "if x1 then x2/x1+x3 else 0" < data
```

probdist: probability/statistic conversion

probdist can convert probabilities to distribution statistics and *vice versa* as seen in tables at the end of most statistics textbooks. Many distributions are supported, including: the normal z, binomial, chi-square, F, and t. The following will print the two-tailed probability of an obtained t statistic of 2.5 with 20 degrees of freedom.

```
probdist prob t 20 2.5
0.021234
```

Similarly, the following will print the two-tailed probability of an F ratio of 6.25 with 1 and 20 degrees of freedom.

```
probdist prob F 1 20 6.25
0.021234
```

These results are the same because of the relationship between the t and F distributions.

The following prints the critical value (also called the quantile) in the chi-square distribution with 5 degrees of freedom to obtain a significance level of .05.

```
probdist crit chisq 5 .05
11.070498
```

Both probabilities and critical values in the normal z distribution use the lower one tail $-\infty$ to $+\infty$ distribution, so the z value that produces the .05 level is obtained with the following.

```
probdist crit z .05
-1.644854
```

The critical value for the 99th percentile is found with the following.

```
probdist crit z .99
2.326348
```

Binomial distribution critical values are treated differently than the other continuous distributions. For the binomial distribution based on five trials, and a probability of success of one half, The critical value for a one-tailed test at the .05 level is:

```
probdist crit binomial 5 1/2 .05
5
```

even though the probability of 5 successes is proportionally much less than .05:

```
probdist prob binomial 5 1/2 5
0.031250
```

This is because the binomial distribution is discrete. Not only are critical values conservative, sometimes there may be no possible value; there is no way to get a less probable event than five out of five successes:

```
      probdist crit binomial 5 1/2 .01
      6
```

Here, **`probdist`** is returning an impossible value (one with zero probability).

ranksort: convert data to ranks

`ranksort` can rank order data from numerical data columns. For the input:

```
      1     95    4.3
      2     113   5.2
      3     89    4.5
      4     100   5.0
      5     89    4.5
```

`ranksort` would produce:

```
      1     3     1
      2     5     5
      3     1.5   2.5
      4     4     4
      5     1.5   2.5
```

The ties in the second and third columns get the average rank of the values for which they are tied. Once data are ranksorted, further ranksorting has no effect. With rank orders within columns, rank order statistics (e.g., Spearman rank order correlation, average group rank) can be computed by parametric programs like **`pair`** or **`regress`**.

Section 4.3 Data Formatting

maketrix: form a matrix format file

maketrix reads its data, one whitespace separated string at a time from its free format input, and produces a multicolumn output.

```
series 1 20 | maketrix 5
```

The above produces a five column output.

```
1     2     3     4     5
6     7     8     9     10
11    12    13    14    15
16    17    18    19    20
```

perm: permute lines

perm, with no options, randomizes its input lines. It can randomize output from programs like **series**.

```
series 1 20 | perm
```

A subset of this permutation is a sample without replacement. The following is a sample of size 10 from the file **data**.

```
perm < data | dm "if INLINE <= 10 then INPUT else EXIT"
```

perm can be supplied a seed for its random number generator, to replicate a random permutation.

```
series 1 20 | perm -s 5762 | maketrix 5
```

The above produces (with my system's random number generator):

```
18    7     10    13    2
14    11    19    15    20
1     3     9     6     16
8     17    12    5     4
```

perm can also put its lines in alphabetical or numerical order. For example, the output from the previous example could be put into ascending order (according to the first number on each line) with:

```
series 1 20 | perm -s 5762 | maketrix 5 | perm -n
```

This produces:

```
1     3     9     6     16
8     17    12    5     4
14    11    19    15    20
18    7     10    13    2
```

dsort: sort data lines by multiple keys

The last example of the **perm** filter showed how lines can be ordered according to the numerical value in the first column. **dsort** can sort lines based on numerical or alphabetical values in any column. For example, the following command sorts the previous example matrix in ascending order of the values in the third column.

```
series 1 20 | perm -s 5762 | maketrix 5 | dsort -n 3
```

This produces:

```
1     3     9     6     16
18    7     10    13    2
8     17    12    5     4
```

```
14    11    19    15    20
```

If there were ties in a column, `dsort` could sort by additional key columns.

transpose: transpose matrix format file

`transpose` flips rows and columns in its input. For the input:

```
1     2     3     4
5     6     7     8
9     10    11    12
```

`transpose` produces:

```
1     5     9
2     6     10
3     7     11
4     8     12
```

The input to `transpose` does not have to be regular, nor does it have to be numerical.

```
one         two         three
four        five
six
seven       eight
nine        ten         eleven
```

For the above input, `transpose` produces the following.

```
one         four        six         seven       nine
two         five                    eight       ten
three                                           eleven
```

Note that with regular inputs, the transposition of a transposition yields the original. This is not necessarily so with data as in the above input and output. The above output piped through another pass of `transpose` produces a result different from the original input.

```
one         two         three
four        five        eleven
six         eight
seven       ten
nine
```

reverse: reverse lines, columns, characters

`reverse` can reverse the lines, fields, or characters in its input. It can provide easier access to the last lines in a file, or the last columns on lines. To get the last 10 lines in a file, we can reverse the file, get the first 10 lines, and then reverse those 10 lines.

```
reverse < data | dm "if INLINE GT 10 then EXIT else INPUT" | reverse
```

To get the last two *columns* in a file is easier.

```
reverse -f < data | dm s2 s1
```

Here, `dm` is used for column extraction, and rather than call `reverse` a second time, what were the last two columns before reversal are listed in the opposite order.

colex: reorder columns, reformat columns

colex is a column extraction program that shares some of the functionality of **dm** and **reverse**. **colex** is faster and has a simpler syntax than **dm** and has data formatting capabilities. Suppose a matrix dataset with 10 columns is created with the following.

```
series 1 50 | maketrix 10
```

colex can extract the last five columns followed by the first five with the command:

```
series 1 50 | maketrix 10 | colex 6-10 1 2 3 4 5
```

Either ranges of columns or single columns can be given. The above command produces:

```
6     7     8     9     10    1     2     3     4     5
16    17    18    19    20    11    12    13    14    15
26    27    28    29    30    21    22    23    24    25
36    37    38    39    40    31    32    33    34    35
46    47    48    49    50    41    42    43    44    45
```

Note in the previous example how the numbers line up on the left, rather than the customary format to line up the unit digits. This is because **colex** puts tabs between columns, and it is not a problem because |STAT programs read data in free-format. **colex** can print its columns in several numerical formats as well as the default string format. The numerical formatting can round values to some number of decimal places (like zero, for whole numbers). The option: **-F 4i** would tell **colex** to format all the columns as integers, each four spaces wide, and the **-t** option would tell **colex** to not place a tab between columns. The format of columns can be assigned to individual columns by placing the format before each range of columns. For example, the following variation on the previous command would print columns 6-10 in a money format with two digits after the decimal place, and columns 1-5 as integers four wide.

```
series 1 50 | maketrix 10 | colex -t 6.2n6-10 4i1-5

  6.00  7.00  8.00  9.00 10.00   1   2   3   4   5
 16.00 17.00 18.00 19.00 20.00  11  12  13  14  15
 26.00 27.00 28.00 29.00 30.00  21  22  23  24  25
 36.00 37.00 38.00 39.00 40.00  31  32  33  34  35
 46.00 47.00 48.00 49.00 50.00  41  42  43  44  45
```

dm: reorder columns

dm, like **colex**, can reorder columns. However, it does not allow the specification of ranges of columns. The above example of **colex** could be done with **dm** with similar results.

```
series 1 50 | maketrix 10 | dm s6 s7 s8 s9 s10 s1 s2 s3 s4 s5
```

abut: paste corresponding lines from files

abut can join data in separate files beside one another. In the usual case, **abut** takes N files with K lines and produces 1 file with K lines. Suppose the files **height** and **weight** contain the respective heights and weights of the same people. Each line in each file contains one height or weight. These could be plotted with the plotting option on the **pair** program with the following command.

```
abut height weight | pair -p
```

Section 4.4 Data Extraction

dm: conditional data extraction

dm can extract subsets of its input, either by columns or by lines. To extract columns of data, each extracted column is specified with the number of the column preceded by the letter **s**. The following extracts columns 8, 2, and 11, in that order.

```
dm s8 s2 s11
```

dm can extract lines of data by using its built-in line skipping expression **SKIP**. The following will extract lines 50 to 100.

```
dm "if INLINE >= 50 & INLINE <= 100 then INPUT else SKIP"
```

It is more awkward than column extraction, but the latter is common.

colex: quick column extraction

colex can extract individual columns, or ranges of columns. For column extraction, it is easier to use and faster than **dm**. The following extracts, in order, columns 8, 2, and 11.

```
colex  8  2  11
```

linex: line extraction

linex can extract individual lines (by number), or ranges of lines. The following extracts, in order, lines 8, 2, and 11.

```
linex  8  2  11
```

To extract lines 50 to 100, you could type:

```
linex  50-100
```

or you could even extract them in reverse order:

```
linex  100-50
```

Section 4.5 Data Validation

validata: data validation

validata will report for its input the number of columns, data-types of columns, and for columns with numerical values, the maxima and minima. **validata** reports any inconsistencies in the number of columns in its input. Floating point numbers can be entered in scientific notation. For the input:

```
1     2     3
4     5     6
7     2E2   end
5           1e-3
```

validata's output is:

```
validata: Variable number of columns at line 4
Col   N  NA alnum alpha   int float other  type   min   max
  1   4   0     4     0     4     4     0   int     1     7
  2   4   0     3     0     2     4     0 float 0.001   200
  3   3   0     3     1     2     2     0 alnum     3     6
```

dm: conditional data validation

dm can find exceptional cases in its input. A simple case is non-numerical input, which can be checked with **dm**'s **number** function.

```
dm  "if !number(s1) then 'bad input on line' else SKIP"  INLINE
```

dm can check for specific values, ranges of values, or specific relations of values. The following prints all lines in **data** with the string **bad** in them.

```
dm "if 'bad' C INPUT then INPUT else SKIP"
```

The input line number could be prepended.

```
dm INLINE "if 'bad' C INPUT then INPUT else SKIP"
```

This is possible because **dm** will produce no output for skipped lines, regardless of expression order. The following prints all lines where column 3 is greater than column 2.

```
dm "if x3 > x2 then INPUT else SKIP"
```

dm can print lengths of strings and check for numerical fields:

```
dm  len(s1)  number(s1)
```

will print the length of column 1 strings, and report if they are numerical (0 for non-numbers, 1 for integers, 2 for real numbers, 3 for exponential scientific notation numbers).

Section 4.6 DM: Tutorial and Manual

dm is a data manipulating program with many operators for manipulating columnated files of numbers and strings. **dm** helps avoid writing little BASIC or C programs every time some transformation to a file of data is wanted. To use **dm**, a list of expressions is entered, and for each line of data, **dm** prints the result of evaluating each expression.

Introductory Examples. Usually, the input to **dm** is a file of lines, each with the same number of fields. Put another way, **dm**'s input is a file with some set number of columns.

Column Extraction: **dm** can be used to extract columns. If **data** is the name of a file of five columns, then the following will extract the 3rd string followed by the 1st, followed by the 4th, and print them to the standard output.

```
dm  s3  s1  s4  <  data
```

Thus **dm** is useful for putting data in a correct format for input to many programs, notably the |STAT data analysis programs. **Warning:** If a column is missing (e.g., you access column 3 and there is no third column in the input), then the value of the access will be taken from the previous input line. This *feature* must be considered if there are blank lines in the input; it may be best to remove blank lines, with **dm** or some other filter program.

Simple Expressions: In the preceding example, columns were accessed by typing the letter **s** (for string) followed by a column number. The numerical value of a column can be accessed by typing **x** followed by a column number. This is useful to form simple expressions based on columns. Suppose **data** is a file of four numerical columns, and that the task is to print the sum of the first two columns followed by the difference of the second two. The easiest way to do this is with:

```
dm  x1+x2  x3-x4  <  data
```

Almost all arithmetic operations are available and expressions can be of arbitrary complexity. Care must be taken because many of the symbols used by **dm** (such as ***** for multiplication) have special meaning when used in UNIX (though not MSDOS). Problems can be avoided by putting expressions in quotes. For example, the following will print the sum of the squares of the first two columns followed by the square of the third, a simple Pythagorean program.

```
dm  "x1*x1+x2*x2"  'x3*x3'  <  data
```

Line Extraction Based on Conditions: **dm** allows printing values that depend on conditions. The **dm** call

```
dm  "if x1 >= 100 then INPUT else NEXT"  <  data
```

will print only those lines that have first columns with values greater than or equal to 100. The variable **INPUT** refers to the whole input line. The special variable **NEXT** instructs **dm** to stop processing on the current line and go to the next.

Data Types

String Data. To access or print a column in a file, the string variable, **s**, is provided. **s**i (the letter **s** followed by a column number, such as **5**) refers to the ith column of the input, treated as a string. The most simple example is to use an **s**i as the only part of an expression.

```
dm  s2  s3  s1
```

will print the second, third and first columns of the input. One special string is called **INPUT**, and is the current input line of data. String constants in expressions are delimited by single or double quotes. For example:

```
"I am a string"
```

Numerical Data. Constant numbers like **123** or **14.6** can be used alone or with other expressions. Two general numerical variables are available To refer to the input columns, there is **x**i and for the result of evaluated expressions, there is **y**i. **x**i refers to the ith column of the input, treated as a number. **x**i is the result of converting **s**i to a number. If **s**i contains non-numerical characters, **x**i may have strange values. A common use of the **x**i is in algebraic expressions.

```
dm  x1+x2  x1/x2
```

will print out two columns, first the sum of the first two input columns, then their ratio.

The value of a previously evaluated expression can be accessed to avoid evaluating the same sub-expression more than once. **y**i refers to the numerical value of the ith expression. Instead of writing:

```
dm  x1+x2+x3  (x1+x2+x3)/3
```

the following would be more efficient:

```
dm  x1+x2+x3     y1/3
```

y1 is the value of the first expression, **x1+x2+x3**. String values of expressions are unfortunately inaccessible.

Indexing numerical variables is usually done by putting the index after **x** or **y**, but if value of the index is to depend on the input, such as when there are a variable number of columns, and only the last column is of interest, the index value will depend on the number of columns. If a computed index is desired for **x** or **y** the index should be an expression in square brackets following **x** or **y**. For example, **x[N]** is the value of the last column of the input. **N** is a special variable equal to the number of columns in **INPUT**. There is the option to use **x1** or **x[1]** but **x1** will execute faster so computed indexes should not be used unless necessary.

Special Variables. **dm** offers some special variables and control primitives for commonly desired operations. Many of the special variables have more than one name to allow more readable expressions. Many can be abbreviated, and the short forms will be shown in square brackets.

N	the number of columns in the current input line
SUM	the sum of the numbers on the input line
INLINE	the line number of the input (initially 1.0)
OUTLINE	the number of lines so far output (initially 0.0)
RAND [R]	a random number uniform in [0,1) (may be followed by a seed)
INPUT [I]	the original input line, all spaces, etc. included
NIL	the empty expression (often used with a test)
KILL [K]	stop processing the current line and produce no output
NEXT	synonym for **KILL**
SKIP	synonym for **KILL**
EXIT [E]	exit immediately (useful after a search)

User Interface

Expressions. Expressions are written in common computer language syntax, and can have spaces or underscores inserted for readability anywhere except (1) in the middle of constants, and (2) in the middle of multicharacter operators such as **<=** (less than or equal to). Four modes are available for specifying expressions to **dm**. They provide the choice of entering expressions from the terminal or a file, and the option to use **dm** interactively or in batch mode.

Argument Mode: The most common but restrictive mode is to supply expressions as arguments on the command line call to **dm**, as featured in previous examples. The main problem with this mode is that many special characters in UNIX and MSDOS are used as operators, requiring that many expressions be quoted. The main advantage is that this mode is most useful in constructing pipelines and shell scripts.

Expression File Mode: Another non-interactive method is to supply **dm** with a file with expressions in it (one to each line) by calling **dm** with:

```
dm  Efilename
```

where **filename** is a file of expressions. This mode makes it easier to use **dm** with pipelines and redirection.

Interactive Mode: **dm** can also be used interactively by calling **dm** with no arguments. In interactive mode, **dm** will first ask for a file of expressions. If the expressions are not in a file, type **RETURN** when asked for the expression file, and they can be entered interactively. A null filename tells **dm** to read expressions from the terminal. In terminal mode, **dm** will prompt with the expression number, and print out how it interprets what is typed in if it has correct syntax, otherwise it allows corrections. When the last expression has been entered, an empty line informs **dm** there are no more. If the expressions are in a file, type in the name of the file, and **dm** will read them from there.

Input. If **dm** is used in interactive mode, it will prompt for an input file. A file name can be supplied or a **RETURN** without a file name tells **dm** to read data from the terminal. Out of interactive mode, **dm** will read from the standard input.

dm reads data a line at a time and stores that line in a string variable called **INPUT**. **dm** then takes each column in **INPUT**, separated by spaces or tabs, and stores each in the string variables, **s**i. **dm** then tries to convert these strings to numbers and stores the result in the number variables, **x**i. If a column is not a number (e.g., it is a string) then its numerical value will be inaccessible, and trying to refer to such a column will cause an error message. The number of columns in a line is stored in a special variable called **N**, so variable numbers of columns can be dealt with gracefully. The general control structure of **dm** is summarized in the following display.

```
read in n expressions; e1, e2, ..., en.
repeat while there is some input left
      INPUT  = <next line from input file>
      N      = <number of fields in INPUT>
      SUM    = 0
      RAND   = <a new random number in [0,1)>
      INLINE = INLINE + 1
      for i  = 1 until N do
            si  = <ith string in INPUT>
            xi  = <si converted to a number>
            SUM = SUM + xi
      for i = 1 until n do
            switch on <value of ei>
                  case EXIT: <stop the program>
                  case KILL: <go to get new INPUT>
                  case NIL : <go to next expression>
                  default  :
                        OUTLINE = OUTLINE + 1
                        yi = <value of ei>
                        if (ei not X'd) print yi
      <print a newline character>
```

Output. In interactive mode, **dm** will ask for an output file (or pipe, on UNIX only).

```
Output file or pipe:
```

A filename, a ‘‘pipe command,’’ or just **RETURN** can be entered. A null filename tells **dm** to print to the terminal. If output is being directed to a file, the output file should be different from the input file. **dm** will ask permission to overwrite any file that contains anything, but that does not mean it makes sense to write the file it is reading from.

On UNIX, the output from **dm** can be redirected to another program by having the first character of the output specification be a pipe symbol, the vertical bar: **|**. For example, the following line tells **dm** to pipe its output to **tee** which prints a copy of its output to the terminal, and a copy to the named file.

```
Output file or pipe: | tee dm.save
```

Out of interactive mode, **dm** prints to the standard output.

dm prints the values of all its expressions in **%.6g** format for numbers (maintaining at most six digits of precision and printing in the fewest possible characters), and **%s** format for strings. A tab is printed after every column to insure separation.

Operations

dm offers many numerical, logical, and string operators. The operators are evaluated in the usual order (e.g., times before plus) and expressions tend be evaluated from left to right. Parentheses can be used to make the order of operations clear. The way **dm** interprets expressions can be verified by entering them interactively on UNIX, in which case **dm** prints a fully parenthesized form.

An assignment operator is not directly available. Instead, variables can be evaluated but not printed by using the expression suppression flag, **X**. If the first character of an expression is **X**, it will be evaluated, but not printed. The value of a suppressed expression can later be accessed with the expression value variable, **y**i.

String Operations. Strings can be lexically compared with several comparators: **<** or **LT** (less-than), **<=** or **LE** (less-than or equal), **=** or **EQ** (equal), **!=** or **NE** (not equal), **>=** or **GE** greater-than or equal), and **>** or **GT** (greater than). They return **1.0** if their condition holds, and **0.0** otherwise. For example,

```
"abcde"  <=  'eeek!'
```

is equal to **1.0.** The length of strings can be found with the **len** operator.

```
len  'five'
```

evaluates to four, the length of the string argument. The character **#** is a synonym for the **len** operator. The numerical type of a string can be checked with the **number** function, which returns 0 for non-numerical strings, 1 for integer strings, and 2 for real numbers (scientific notation or strings with non-zero digits after the decimal point).

Individual characters inside strings can be accessed by following a string with an index in square brackets.

```
"abcdefg"[4]
```

is the ASCII character number (164.0) of the 4th character in **abcdefg**. Indexing a string is mainly useful for comparing characters because it is not the character that is printed, but the character number. A warning is appropriate here:

```
s1[1]  =  '*'
```

will result in an error because the left side of the **=** is a number, and the right hand side is a string. The correct (although inelegant) form is:

```
s1[1]  =  '*'[1]
```

A substring test is available. The expression:

```
string1  C  string2
```

will return **1.0** if **string1** is somewhere in **string2**. This can be used as a test for character membership if string1 has only one character. Also available is **!C** which returns **1.0** if **string1** is NOT in **string2**.

Numerical Operators. The numerical comparators are:

```
<  <=  =  !=  >=  >
LT LE  EQ NE  GE GT
```

and have the analogous meanings as their string counterparts.

The binary operators, `+` (addition), `-` (subtraction or "change-sign"), `*` (multiplication), and `/` (division) are available. Multiplication and division are evaluated before addition and subtraction, and are all evaluated left to right. Exponentiation, `^`, is the binary operator of highest precedence and is evaluated right to left. Modulo division, `%`, has the same properties as division, and is useful for tests of even/odd and the like. NOTE: Modulo division truncates its operands to integers before dividing.

Several unary functions are available: `l` (natural log `[log]`), `L` (base ten log `[Log]`), `e` (exponential `[exp]`), `a` (absolute value `[abs]`), `f` (floor `[floor]`), `c` (ceiling `[ceil]`). Their meaning can be verified in the UNIX Programmer's Manual. Single letter names for these functions or the more mnemonic strings bracketed after their names can be used. Also available are trigonometric functions that work on degrees in radians: `sin cos tan asin acos atan`.

Logical Operators. Logical operators are of lower precedence than any other operators. Both logical AND, `&` and OR `|` can be used to form complicated tests. For example, to see if the first three columns are in either increasing or decreasing order, one could test if `x2` was between `x1` and `x3`:

```
x1<x2  &  x2<x3  |  x1>x2  &  x2>x3
```

would equal `1.0` if the condition was satisfied. Parentheses are unnecessary because `<` and `>` are of higher precedence than `&` which is of higher precedence than `|`. The above expression could be written as:

```
x1 LT x2  AND  x2 LT x3  OR  x1 GT x2  AND  x2 GT x3
```

by using synonyms for the special character operators. This is useful to avoid the special meaning of characters in command lines. The unary logical operator, `!` (NOT), evaluates to 1.0 if its operand is `0.0`, otherwise it equals `0.0`. Many binary operators can be immediately preceded by `!` to negate their value. `!=` is "not equal to," `!|` is "neither," `!&` is "not both," and `!C` is "not in."

Conditional Expressions. The expressions:

```
if expression1 then expression2 else expression3
   expression1  ?   expression2   :  expression3
```

evaluate to `expression2` if `expression1` is non-zero, otherwise they evaluate to `expression3`. The first form is more mnemonic than the second which is consistent with C syntax. Upper case names can be used in their stead. Both forms have the same meaning. `expression1` has to be numerical, `expression2` or `expression3` can be numerical or string. For example, The following expression will filter out lines with the word `bad` in them.

```
if  'bad'  C  INPUT  then  KILL  else  INPUT
```

As another example, the following expression will print the ratio of columns two and three if (a) there are at least three columns, and (b) column three is not zero.

```
if  (N >= 3)  &  (x3 != 0)  then  x2/x3  else  'bad line'
```

These are the only expressions, besides `s`i or a string constant that can evaluate to a string. If a conditional expression does evaluate to a string, then it CANNOT be used in some other expression. The conditional expression is of lowest precedence and groups left to right, however parentheses are recommended to make the semantics obvious.

Expression Syntax

Arithmetic expressions may be formed using variables (with **x**i and **y**i) and constants and can be of arbitrary complexity. In the following table, unary and binary operators are listed along with their precedences and a brief description. All unary operators are prefix except string indexing, **[]**, which is postfix. All binary operators are infix.

Operators of higher precedence are executed first. All binary operators are left associative except exponentiation, which groups to the right. An operator, **O**, is left associative if **xOxOx** is parsed as **(xOx)Ox**, while one that is right associative is parsed as **xO(xOx)**.

Unary Operators:

op	*prec*	*description*
`sin`	10	sine of argument degrees in radians
`cos`	10	cosine of argument degrees in radians
`tan`	10	tangent of argument degrees in radians
`asin`	10	arc (inverse) sine function
`acos`	10	arc (inverse) cosine function
`atan`	10	arc (inverse) tangent function
`sqrt`	10	square root function
`log`	10	base e logarithm [`l`]
`Log`	10	base 10 logarithm [`L`]
`exp`	10	exponential [`e`]
`abs`	10	absolute value [`a`]
`ceil`	10	ceiling (rounds up to next integer) [`c`]
`floor`	10	floor (rounds down to last integer) [`f`]
`len`	10	number of characters in string [#]
`number`	10	report if string is a number (0 non, 1 int, 2 real)
`[]`	10	ASCII number of indexed string character
`-`	9	change sign
`!`	4	logical not (also `NOT`, `not`)

Binary Operators:

op	*prec*	*description*
`^`	8	exponentiation
`*`	7	multiplication
`/`	7	division
`%`	7	modulo division
`+`	6	addition
`-`	6	subtraction
`=`	5	test for equality (also `EQ`; opposite `!=`, `NE`)
`>`	5	test for greater-than (also `GT`; opposite `<=`, `LE`)
`<`	5	test for less-than (also `LT`; opposite, `>=`, `GE`)
`C`	5	substring (opposite `!C`)
`&`	4	logical AND (also `AND`, `and`; opposite `!&`)
`|`	3	logical OR (also `OR`, `or`; opposite `!|`)

Some Examples

To print lines 10-20 from an input file **dm.dat**, you could run the following command (note that **x** is the same as **x0**, which is the same as **INLINE**, the input line number).

```
dm  "if x >= 20 and x <= 20 then INPUT else SKIP"  < dm.dat
```

To print all the lines longer than 100 characters, you could run the following:

```
dm  "if len(INPUT) > 100 then INPUT else SKIP"  < dm.dat
```

To print the running sums of values in a column, you need to use the **y** variables. The following will print the running sum of values in the first column.

```
dm y1+x1
```

To print the running sum of the data in the 5th column is a bit unintuitive. **y1** is the value from the previous line of the first expression, and **x5** is the value of the fifth column on the current line. To get the running sum of column 5, you would type:

```
dm y1+x5
```

If the running sum is to come out in the third column, then you would run:

```
dm <something> <something> y3+x5
```

dm is good at making tables of computed values. In the following example, the **echo** command prints headings for the columns, and **colex** reformats the output of **dm**. **colex** sets the default format to 10.3n (numbers 10 wide, with 3 decimal places), and prints column 1 in 2i format (2-wide integer) and column 6 in 6i format (6-wide integer). The **-t** option to **colex** stops the printing of tabs after columns.

```
echo " x       1/x  x**2   sqrt(x)    log(x)"
series 1 10 | dm x1 1/x1 "x1*x1" "sqrt(x1)" "log(x1)" |
      colex -t -F 10.3n 2i1 2 6i3 4-5

 x       1/x  x**2   sqrt(x)    log(x)
 1     1.000     1     1.000     0.000
 2     0.500     4     1.414     0.693
 3     0.333     9     1.732     1.099
 4     0.250    16     2.000     1.386
 5     0.200    25     2.236     1.609
 6     0.167    36     2.449     1.792
 7     0.143    49     2.646     1.946
 8     0.125    64     2.828     2.079
 9     0.111    81     3.000     2.197
10     0.100   100     3.162     2.303
```

CHAPTER 5

Data Analysis

Each of the analysis programs are introduced, showing some, but not all of their options. Full documentation can be found in the manual entries. Details about the procedures and assumptions are found in the references in the ALGORITHM sections of the manual entries. Most analysis programs allow summary statistics, inferential statistics. and simple graphics. In general, a program consists of all the analyses for a specific type of data. There are programs for univariate (single) distributions, multilevel, and multifactor analysis. Some simple analyses are possible by combining data manipulation and analysis programs. For example, Scheffe confidence intervals can be computed for means using the **`probdist`** and **`calc`** programs. A tutorial and reference manual for **`calc`** is the final section of this chapter.

Section 5.1 Table of Analysis Programs

	Descriptive	Inferential	Graphical
Univariate			
`stats`	simple stats		
`desc`	many stats	t-test	histogram
`ts`	summary	auto-correlation	bar plot
Multilevel			
`oneway`	group stats	(un)weighted between anova	error barplots
`rankind`	rank stats	Mann-Whitney Kruskal-Wallis	fivenum plots
Bivariate			
`pair`	column stats differences correlation	paired t-test simple regression	scatter plot
Multivariate			
`regress`	variable stats correlation	linear regression partial correlation	residual output
`rankrel`	rank stats correlation	Wilcoxon Friedman	
Multifactor			
`anova`	cell stats	mixed model ANOVA	
`contab`	crosstabs	chi-square fisher exact test	

Section 5.2 stats: print summary statistics

`stats` prints summary statistics for its input. Its input is a free format series of strings from which it extracts only numbers for analysis. When a full analysis is not needed, the following names request specific statistics.

```
n  min  max  sum  ss  mean  var  sd  skew  kurt  se  NA

prompt: stats
stats: reading input from terminal
1 2 3 4 5 6 7 8 9 10
EOF
n     =     10
NA    =     0
min   =     1
max   =     10
sum   =     55
ss    =     385
mean  =     5.5
var   =     9.16667
sd    =     3.02765
se    =     0.957427
skew  =     0
kurt  =     1.43836

prompt: series 1 100 | dm logx1 | stats min mean max
0     3.63739     4.60517
```

Section 5.3 desc: descriptions of a single distribution

desc describes a single distribution. Summary statistics, modifiable format histograms and frequency tables, and single distribution *t*-tests are supported. **desc** reads free format input, with numbers separated by any amount of white space (blank spaces, tabs, newlines). When order statistics are being printed, or when a histogram or frequency table is being printed, there is a limit to the number of input values that must be stored. Although system dependent, usually several thousand values can be stored.

An example input to **desc** is shown below.

```
3 3 4 4 7 7 7 7 8 9 1 2 3 4 5 6 7
8 9 9 8 7 6 5 4 3 2 4 5 6 1 2 3 4 3  1  7 7
```

The call to **desc** includes many options: **-o** for Order statistics, **-hcfp** respectively for a Histogram, and a table with Cumulative Frequencies and Proportions, **-m 0.5** to set the Minimum allowed value to 0.5, **-M 8** to set the Maximum allowed value to 8, **-i 1** to set the Interval width in the histogram and table to 1, and **-t 5** to request a t-test with null mean equal to 5.

```
desc  -o  -hcfp  -m 0.5  -M 8  -i 1  -t 5
```

The output follows.

```
------------------------------------------------------------
 Under Range    In Range  Over Range     Missing         Sum
           0          35           3           0     164.000
------------------------------------------------------------
        Mean      Median    Midpoint   Geometric    Harmonic
       4.686       4.000       4.500       4.055       3.296
------------------------------------------------------------
          SD   Quart Dev       Range     SE mean
       2.193       2.000       7.000       0.371
------------------------------------------------------------
     Minimum  Quartile 1  Quartile 2  Quartile 3     Maximum
       1.000       3.000       4.000       7.000       8.000
------------------------------------------------------------
        Skew     SD Skew    Kurtosis     SD Kurt
      -0.064       0.414       1.679       0.828
------------------------------------------------------------
   Null Mean           t    prob (t)           F    prob (F)
       5.000      -0.848       0.402       0.719       0.402
------------------------------------------------------------

Midpt    Freq     Cum    Prop     Cum
1.000       3       3   0.086   0.086 ***
2.000       3       6   0.086   0.171 ***
3.000       6      12   0.171   0.343 ******
4.000       6      18   0.171   0.514 ******
5.000       3      21   0.086   0.600 ***
6.000       3      24   0.086   0.686 ***
7.000       8      32   0.229   0.914 ********
8.000       3      35   0.086   1.000 ***
```

Section 5.4 ts: time series analysis and plots

ts performs simple analyses and plots for time series data. Its input is a free format stream of at most 1000 numbers. It prints summary statistics for the time series, allows rescaling of the size of the time series so that time series of different lengths can be compared, and optionally computes auto-correlations of the series for different lags. Auto-correlation analysis detects recurring trends in data. For example, an auto-correlation of lag 1 of a time series pairs each value with the next in the series. **ts** is best demonstrated on an oscillating sequence, the output from which is shown below. The call to **ts** includes several options: **-c 5** requests autocorrelations for lags of 1 to 5, the **-ps** options request a time-series plot and statistics, and the **-w 40** option sets the width of the plot to 40 characters.

```
ts  -c 5  -ps  -w 40

1 2 3 4 5 4 3 2 1 2 3 4 5 4 3 2 1

n       = 17
sum     = 49
ss      = 169
min     = 1
max     = 5
range   = 4
midpt   = 3
mean    = 2.88235
sd      = 1.31731
Lag      r    r^2    n'           F    df      p
  0  0.000  0.000    17       0.000    15  1.000
  1  0.667  0.444    16      11.200    14  0.005
  2 -0.047  0.002    15       0.028    13  0.869
  3 -0.701  0.491    14      11.590    12  0.005
  4 -1.000  1.000    13       0.000    11  0.000
  5 -0.698  0.487    12       9.507    10  0.012
-----+------------|------------+--------
------------------|
         ---------|
                  |-
                  |-----------
                  |---------------------
                  |-----------
                  |-
         ---------|
------------------|
         ---------|
                  |-
                  |-----------
                  |---------------------
                  |-----------
                  |-
         ---------|
------------------|
-----+------------|------------+--------
1.000                              5.000
```

Section 5.5 oneway: one way analysis of variance

oneway performs a between-groups one-way analysis of variance. It is partly redundant with **anova**, but is provided to simplify analysis of this common experimental design. The input to **oneway** consists of each group's data, in free input format, separated by a special value called the *splitter*. Group sizes can differ, and **oneway** uses a weighted or unweighted (Keppel, 1973) means solution. At most 20 groups can be compared. When two groups are being compared, the **-t** option prints the significance test as a *t*-test. The program is based on a method of analysis described by Guilford and Fruchter (1978).

An example interactive session with **oneway** is shown below. The call to **oneway** includes the **-s** option with 999 as the value of the Splitting value between groups. The **-u** option request the *unweighted* means solution rather than the default *weighted* means solution. The **-w 40** option requests an error bar plot of width 40. Meaningful names are given to the groups.

```
prompt: oneway  -s 999  -u  -w 40  less equal more
oneway: reading input from terminal:
1 2 3 4 5 4 3 2 1
999
3 4 5 4 3 4 5 4 3
999
7 6 5 7 6 5
EOF

Name          N     Mean       SD      Min      Max
less          9    2.778    1.394    1.000    5.000
equal         9    3.889    0.782    3.000    5.000
more          6    6.000    0.894    5.000    7.000
Total        24    4.000    1.642    1.000    7.000

less      |<-======(==#==)======---->             |
equal     |            <===(=#)====->             |
more      |                         <===(==#=)===>|
           1.000                            7.000

Unweighted Means Analysis:
Source           SS    df         MS        F     p
Between      41.333     2     20.667   17.755 0.000 ***
Within       24.444    21      1.164
```

Section 5.6 rankind: rank-order analysis of independent groups

rankind prints rank-order summary statistics and compares independent group data using non-parametric methods. It is the non-parametric counterpart to the normal theory **oneway** program, and the input format to **rankind** is the same as for **oneway**. Each group's data are in free input format, separated by a special value, called the *splitter*. Like **oneway**, there are plots of group data, but **rankind**'s show the minimum, 25th, 50th, and 75th percentiles, and the maximum. Significance tests include the median test, Fisher's exact test, Mann-Whitney U test for ranks, and the Kruskal-Wallis analysis of variance of ranks.

The following example is for the same data as in the example with **oneway**. The options to set the splitter and plot width are the same for both programs. Meaningful names are given to the groups.

```
prompt: rankind  -s 999  -w 40  less equal more
rankind: reading input from terminal:
1 2 3 4 5 4 3 2 1
999
3 4 5 4 3 4 5 4 3
999
7 6 5 7 6 5
EOF

               N    NA      Min      25%   Median      75%      Max
less           9     0     1.00     1.75     3.00     4.00     5.00
equal          9     0     3.00     3.00     4.00     4.25     5.00
more           6     0     5.00     5.00     6.00     7.00     7.00
Total         24     0     1.00     3.00     4.00     5.00     7.00

less      |<   ---------#------      >             |
equal     |            <-----#--     >             |
more      |                          <------#----->|
           1.000                               7.000

Median-Test:
                 less  equal   more
        above       1      2      6      9
        below       6      3      0      9
                    7      5      6     18
        WARNING: 6 of 6 cells had expected frequencies less than 5
        chisq       9.771429     df   2      p  0.007554

Kruskal-Wallis:
        H (not corrected for ties)                 13.337778
        Tie correction factor                       0.965652
        H (corrected for ties)                     13.812197
        chisq      13.812197     df   2      p  0.001002
```

Section 5.7 pair: paired points analysis and plots

pair analyzes paired data by printing summary statistics and significance tests for two input variables in columns and their difference, correlation and simple linear regression, and plots. The test of the difference of the two columns against zero is equivalent to a paired t-test. The input consists of a series of lines, each with two paired points. When data are being stored for a plot, at most 1000 points can be processed.

An example input to **pair** is generated using the **series** and **dm** programs connected by pipes. The input to **pair** are the numbers 1 to 100 in column 1, and the square roots of those numbers in column 2. The **dm** built-in variable **INLINE** is used in a condition to switch the sign of the second column for the second half of the data. **pair** reads X-Y points and predicts Y (in column 2) with X (in column 1). The significance test of the difference of the columns against 0.0 is equivalent to a paired-groups *t*-test. The call to **pair** includes several options: **-sp** requests Statistics and a Plot, **-w 40** sets the Width of the plot to 40 characters, and **-h 15** sets the Height of the plot to 15 characters.

```
series 1 100 | dm x1 "(INLINE>50?-1:1)*x1^.5" | pair -sp -w 40 -h 15

                Column 1   Column 2   Difference
Minimums          1.0000   -10.0000       0.0000
Maximums        100.0000     7.0711     110.0000
Sums           5050.0000  -193.3913    5243.3913
SumSquares   338350.0000  5049.9989  395407.6303
Means            50.5000    -1.9339      52.4339
SDs              29.0115     6.8726      34.8845
t(99)            17.4069    -2.8140      15.0307
p                 0.0000     0.0059       0.0000

Correlation    r-squared      t(98)            p
    -0.8226       0.6767   -14.3219       0.0000
  Intercept        Slope
     7.9070      -0.1949

|----------------------------------------|7.07107
|              323232                    |
|        123232                          |
|     2322                               |
|  223                                   |
|221                                     |
|1                                       |
|                                        |
|                                        |Column 2
|                                        |
|                                        |
|                                        |
|                                        |
|                    2322                |
|                       123232321        |
|                               223232323|
|----------------------------------------|-10
1.000                              100.000
            Column 1  r=-0.823
```

Section 5.8 rankrel: rank-order analysis of related groups

rankrel prints rank-order summary statistics and compares data from related groups. It is the non-parametric counterpart to parts of the normal theory **pair** and **regress** programs. Each group's data are in a column, separated by whitespace. Instead of normal theory statistics like mean and standard deviation, the median and other quartiles are reported. Significance tests include the binomial sign test, the Wilcoxon signed-ranks test for matched pairs, and the Friedman two-way analysis of variance of ranks.

The following (transposed) data are contained in the file **siegel.79**, and are based on the example on page 79 of Siegel (1956). The astute analyst will notice that the last datum in column 2 in Siegel's book is misprinted as 82.

```
82  69    73    43    58    56    76    65
63  42    74    37    51    43    80    62
```

When the output contains a suggestion to consult a table of computed exact probability values, it is because the continuous chi-square or normal approximation may not be adequate. Siegel (1956) notes that the normal approximation for the probability of the computed Wilcoxon *T* statistic is *excellent* even for small samples such as the one above. Once again, the astute analyst will see the flaw in Siegel's analysis when he uses a normal approximation; he fails to use a correction for continuity.

```
prompt: rankrel control prisoner < siegel.79
               N   NA      Min      25%   Median      75%      Max
control        8    0    43.00    57.00    67.00    74.50    82.00
prisoner       8    0    37.00    42.50    56.50    68.50    80.00
Total         16    0    37.00    47.00    62.50    73.50    82.00

Binomial Sign Test:
        Number of cases control is above prisoner:   6
        Number of cases control is below prisoner:   2
        One-tail probability (exact)           0.144531

Wilcoxon Matched-Pairs Signed-Ranks Test:
    Comparison of control and prisoner
        T (smaller ranksum of like signs)      4.000000
        N (number of signed differences)       8.000000
        z                                      1.890378
        One-tail probability approximation     0.029354
        NOTE: Yates' correction for continuity applied
        Check a table for T with N = 8

Friedman Chi-Square Test for Ranks:
        Chi-square of ranks                    2.000000
        chisq       2.000000     df   1      p  0.157299
        Check a table for Friedman with N = 8

Spearman Rank Correlation (rho) [corrected for ties]:
        Critical r (.05) t approximation       0.706734
        Critical r (.01) t approximation       0.834342
        Check a table for Spearman rho with N = 8
        rho                                    0.785714
```

Section 5.9 regress: multiple correlation/regression

`regress` performs a multiple linear correlation and regression analysis. Its input consists of a series of lines, each with an equal number of columns, one column per variable. In the regression analysis, the first column is predicted with all the others. There are options to print the matrix of sums of squares and the covariance matrix. There is also an option to perform a partial correlation analysis to see the contribution of individual variables to the whole regression equation. The program is based on a method of analysis described by Kerlinger & Pedhazur (1973). Non-linear regression models are possible using transformations with |STAT utilities like `dm`. The program can handle up to 20 input columns, but the width of the output for more than 10 is awkward.

The following artificial example predicts a straight line with a log function, a quadratic, and an inverse function. The input to `regress` is created with `series` and `dm`. The call to `regress` includes the `-p` option to request a partial correlation analysis and meaningful names for most of the variables in the analysis. The output from `regress` includes summary statistics for each variable, a correlation matrix, the regression equation and the significance test of the multiple correlation coefficient, and finally, a partial correlation analysis to examine the contribution of individual predictors, after the others are included in the model.

```
series 1 20 | dm x1 logx1 x1*x1 1/x1 | regress -p linear log quad inverse

Analysis for 20 cases of 4 variables:
Variable       linear        log       quad    inverse
Min            1.0000     0.0000     1.0000     0.0500
Max           20.0000     2.9957   400.0000     1.0000
Sum          210.0000    42.3356  2870.0000     3.5977
Mean          10.5000     2.1168   143.5000     0.1799
SD             5.9161     0.8127   127.9023     0.2235

Correlation Matrix:
linear         1.0000
log            0.9313     1.0000
quad           0.9713     0.8280     1.0000
inverse       -0.7076    -0.9061    -0.5639     1.0000
Variable       linear        log       quad    inverse

Regression Equation for linear:
linear  =  5.539 log  +  0.02245 quad  +  6.764 inverse  +  -5.66305

Significance test for prediction of linear
    Mult-R  R-Squared      SEest    F(3,16)   prob (F)
    0.9996     0.9993     0.1707  7603.7543     0.0000

Significance test(s) for predictor(s) of linear
Predictor     beta         b        Rsq        se     t(16)         p
log         0.7609    5.5389    0.9684    0.2709   20.4478    0.0000
quad        0.4854    0.0225    0.8795    0.0009   25.4555    0.0000
inverse     0.2555    6.7638    0.9314    0.6688   10.1139    0.0000
```

Section 5.10 anova: multi-factor analysis of variance

anova performs analysis of variance with one random factor and up to nine independent factors. Both within-subjects and unequal-cells between-subjects factors are supported. Nested factors, other than those involving the random factor, are not supported. The input format is simple: each datum is preceded by a description of the conditions under which the datum was obtained. For example, if subject 3 took 325 msec to respond to a loud sound on the first trial, the input line to **anova** might be:

```
s3  loud  1  325
```

From input lines of this format, **anova** infers whether a factor is within- or between-subjects, prints cell means for all main effects and interactions, and prints standard format *F* tables with probability levels. The computations done in **anova** are based on a method of analysis described by Keppel (1973), however, for unequal cell sizes on between-groups factors, the weighted-means solution is used instead of Keppel's preferred unweighted solution. The weighted-means solution requires that sample sizes must be in constant proportions across rows and columns in interactions of between-subjects factors or else the analysis may be invalid.

An example input to **anova** is shown below. The call to **anova** gives meaningful names to the columns of its input. The output from **anova** contains cell statistics for all systematic sources (main effects and interactions), a summary of the design, and an F-table.

```
anova subject noise trial RT

s1   loud   1   259
s1   loud   2   228
s2   soft   1   526
s2   soft   2   480
s3   loud   1   325
s3   loud   2   315
s4   soft   1   418
s4   soft   2   397

SOURCE: grand mean
noise  trial     N       MEAN         SD         SE
                 8   368.5000   104.8713    37.0776

SOURCE: noise
noise  trial     N       MEAN         SD         SE
loud             4   281.7500    46.1257    23.0629
soft             4   455.2500    58.8749    29.4374

SOURCE: trial
noise  trial     N       MEAN         SD         SE
       1         4   382.0000   116.0603    58.0302
       2         4   355.0000   108.1943    54.0971

SOURCE: noise trial
noise  trial     N       MEAN         SD         SE
loud   1         2   292.0000    46.6690    33.0000
loud   2         2   271.5000    61.5183    43.5000
soft   1         2   472.0000    76.3675    54.0000
soft   2         2   438.5000    58.6899    41.5000
```

```
FACTOR:    subject      noise      trial         RT
LEVELS:          4          2          2          8
TYPE  :     RANDOM    BETWEEN     WITHIN       DATA

SOURCE            SS  df            MS        F      p
=====================================================
mean   1086338.0000   1  1086338.0000  145.111  0.007 **
s/n      14972.5000   2     7486.2500

noise    60204.5000   1    60204.5000    8.042  0.105
s/n      14972.5000   2     7486.2500

trial     1458.0000   1     1458.0000   10.942  0.081
ts/n       266.5000   2      133.2500

nt          84.5000   1       84.5000    0.634  0.509
ts/n       266.5000   2      133.2500
```

Section 5.11 contab: contingency tables and chi-square

contab supports the analysis of multifactor designs with categorical data. Contingency tables (also called crosstabs) and chi-square test of independence are printed for all two-way interactions of factors. The method of analysis comes from several sources, especially Bradley (1968), Hays (1973), and Siegel (1956). The input format is similar to that of **anova**: each cell count is preceded by labels indicating the level at which that frequency count was obtained. Below are fictitious data of color preferences of boys and girls:

```
boys        red         3
boys        blue        17
boys        green       4
boys        yellow      2
boys        brown       10
girls       red         12
girls       blue        10
girls       green       5
girls       yellow      8
girls       brown       1
```

The output from the following command includes of a summary of the input design, tables, and statistical analyses.

```
contab  sex  color

FACTOR:         sex      color       DATA
LEVELS:           2          5         72

color      count
red           15
blue          27
green          9
yellow        10
brown         11
Total         72
        chisq      15.222222     df   4      p  0.004262

SOURCE: sex color
             red    blue   green  yellow   brown  Totals
boys           3      17       4       2      10      36
girls         12      10       5       8       1      36
Totals        15      27       9      10      11      72
Analysis for sex x color:
        WARNING: 2 of 10 cells had expected frequencies < 5
        chisq      18.289562     df   4      p  0.001083
        Cramer’s V                              0.504006
        Contingency Coefficient                 0.450073
```

Section 5.12 dprime: d'/beta for signal detection data

dprime computes *d'* (a measure of discrimination of stimuli) and *beta* (a measure of response bias) using a method of analysis discussed in Coombs, Dawes, & Tversky (1970). The input to **dprime** can be a series of lines, each with two paired indicators: the first tells if a signal was present and the second tells if the observer detected a signal. From that, **dprime** computes the *hit-rate* (the proportion of times the observer detected a signal that was present), and the *false-alarm-rate* (the proportion of times the observer reported a signal that was not present). If the hit-rate and the false-alarm-rate are known, then they can be supplied directly to the program:

```
prompt: dprime .7 .4
  hr       far     dprime      beta
0.70      0.40       0.78      0.90
```

The input in raw form, with the true stimulus (Was a signal present or just noise?) in column 1 and the observer's response (Did the observer say there was a signal?) in column 2, is followed by the output.

```
signal      yes
signal      yes
signal      yes
signal      yes
signal      yes
signal      yes
signal      yes
signal      no
signal      no
signal      no
noise       yes
noise       yes
noise       no
noise       no
noise       no
```

dprime would produce for the above data:

```
          signal   noise
yes         7        2
 no         3        3

  hr       far     dprime      beta
0.70      0.40       0.78      0.90
```

Section 5.13 CALC: Tutorial and Manual

calc is a program for mathematical calculations for which you might use a hand-held calculator. **calc** supplies most of the operations common to programming languages and variables with constraint properties much like those in spreadsheets.

The arithmetical operators **calc** offers are

+	addition
-	subtraction and change-sign
*****	multiplication
/	division
%	modulo division
ˆ	exponentiation

Arithmetical expressions can be arbitrarily complex and are generally evaluated left to right. That is,

```
a + b - c
```

is the same as

```
(a + b) - c
```

Exponentiation is evaluated before multiplication and division which are evaluated before addition and subtraction. For example, the expression

```
a + b - c * d / e ˆ 2
```

is parsed as

```
(a + b) - ((c * d) / (e ˆ 2))
```

This default order of operations can be overridden by using parentheses.

calc supplies some transcendental functions: **sqrt**, **log**, **exp**, and **abs**, and the following trigonometric functions: **sin**, **asin**, **cos**, **acos**, **tan**, and **atan**, for which degrees are measured in radians.

Using CALC

To use **calc**, begin by typing

```
calc
```

at the command level, and **calc** will prompt you with

```
CALC:
```

You can supply inputs to **calc** from files specified by command line arguments. For example, typing

```
calc foo
```

will read from the file **foo** and then ask for input from you. Type in each of your expressions followed by **RETURN** and **calc** will respond with how it parsed your expression followed by the result. In all following examples, what you would type in is preceded by the **calc** prompt

```
CALC:
```

and what **calc** responds with is immediately after. A simple calculation is:

```
CALC: sqrt (12ˆ2 + 5ˆ2)
sqrt(((12 ˆ 2) + (5 ˆ 2)))     = 13
```

Expressions can be stored by assigning them to variables. For example you could type:

```
CALC: pi = 22/7
(22 / 7)      = 3.14286
CALC: pi
pi          = 3.14286
```

Variables can be used in expressions.

```
CALC: area = pi * r^2
(pi * (r ^ 2))      = UNDEFINED
CALC: area
area     = UNDEFINED
```

area is undefined because **r** has not been set. Once **r** is set, **area** will have a value because **area** is set to an equation rather than a particular value. This can be observed by printing all the variables so far introduced with **^V**, which may have to be typed twice as **^V** is used in some UNIX versions to quote characters.

```
CALC: ^V
pi       =      3.14286 = (22 / 7)
area     =    UNDEFINED = (pi * (r ^ 2))
r        =    UNDEFINED =
```

The variable table is formatted so that each variable's name is on the left, followed by its current value, followed by its current definition. If **r** is set to 5, the value of **area** is now defined.

```
CALC: r = 5
5        = 5
CALC: ^V
pi       =      3.14286 = (22 / 7)
area     =      78.5714 = (pi * (r ^ 2))
r        =            5 = 5
```

The effect of changing **r** on **area** can be observed because of the way **area** is defined.

```
CALC: r = 2
2         = 2
CALC: area
area      = 12.5714
```

A special variable named **$** is always equal to the most recent result printed.

Setting Constant Values

Of course, there are times when you want to set a variable to a value and not have it depend on the values of variables at a later time. To do this, you precede an expression with the number operator **#**. For example,

```
CALC: area2 = # area
12.5716      = 12.5716
CALC: ^V
pi       =      3.14286 = (22 / 7)
area     =      12.5716 = (pi * (r ^ 2))
r        =            2 = 2
area2    =      12.5716 = 12.5716
```

area2 does not depend on the variable to which it was set because the number operator **#** only lets numbers through it rather than expressions. If **area2** was set without the **#** operator, it would be subject to any changes in **area** or to any changes in variables on which **area** depends.

```
CALC: area2 = area
area      = 12.5716
CALC: ^V
```

```
pi       =      3.14286 = (22 / 7)
area     =      12.5716 = (pi * (r ^ 2))
r        =            2 = 2
area2    =      12.5716 = area
```

Testing Conditions

Variables can be set based on a tested condition. For example, you may want a variable **max** to always be the maximum of **a** and **b**.

```
CALC: max = if a > b then a else b
(if (a > b) then a else b)    = UNDEFINED
```

max is undefined because **a** and **b** have not been set.

```
CALC: a = 21
21     = 21
CALC: b = 3^3
(3 ^ 3)    = 27
CALC: max
max      = 27
CALC: a = 50
50   = 50
CALC: max
max      = 50
```

The if-then-else expression allows variables to be set based on conditions. This condition can be made up with relational and logical operators. The relational operators available with **calc** are:

==	test equality
!=	test inequality
>=	greater than or equal
<=	less than or equal
>	greater than
<	less than

while the logical operators are:

&	and
|	or
!	not

A more complicated expression involving these is:

```
if a > b & b > c then b
```

The **else** part of the conditional is optional, and if not present and the condition is false, the conditional is undefined.

Undefined Variables

Variables are undefined if they have not been set, if they depend on variables that are undefined, or if they are set to an expression involving an illegal operation.

```
CALC: 1/0
(1 / 0)     = UNDEFINED
```

You can be confident that no operations will result in **calc** blowing up. Thus you could write the equation for the roots of a quadratic formula with the following definitions and always get reasonable answers.

```
x = 0
a = b = 1
```

```
c = -1
radical = sqrt (b^2 - 4*a*c)
equation = a*x^2 + b*x + c
derivative = 2*a*x + b
root1 = (-b + radical) / (2 * a)
root2 = (-b - radical) / (2 * a)
```

Control Characters

Non-mathematical operations are accomplished with control characters. To type a control character, say CTRL-p, while you hold down the key labeled CTRL you type a **p**. This will appear as **^P**. Some control characters have special meanings, such as "stop the program" so you must be careful with them. On UNIX, you can avoid some problems with control characters by typing a **^V** before them. This character removes any special meaning associated with the character immediately following it. So to type **^P** you could be extra safe and type **^V^P**. To type a **^V**, you may have to type it twice. Unfortunately, these conventions are not universal.

The following control operations are available with **calc**.

^P	toggle the printing of expressions (UNIX only)
^Rf	read the input from file f and return to current state
^V	print the variable table
^Wf	write the variable table to file **f**
	(**^W** is a synonym for **^V**)

If you forget any of these commands, you can type a **?** to get **calc** to remind you.

Table of `calc` Operations

Operator	Precedence	Associativity	Description
`$`	const	none	numerical value of previous calculation
`#`a	1	none	numerical value of a
a`=`b	2	right	a is set to expression b
`if` a `then` b	3	left	if a != 0 then b else `UNDEFINED`
`else`	4	left	
a`|`b	5	left	true if a or b is true
a`&`b	6	left	true is a and b are true
`!`a	7	none	true is a is false
a`==`b	8	none	true if a equals b
a`!=`b	8	none	true if a is not equal b
a`<`b	8	none	true if a is less than b
a`>`b	8	none	true if a greater than b
a`>=`b	8	none	true if a > b | a == b
a`<=`b	8	none	true if a < b | a == b
a`+`b	9	left	a plus b
a`-`b	9	left	a minus b
a`*`b	10	left	a times b
a`/`b	10	left	a divided by b
a`%`b	10	left	a modulo b
a`^`b	11	right	a to the b
`-`a	12	none	change sign
`abs(`a`)`	12	none	absolute value
`exp(`a`)`	12	none	e to the power a
`log(`a`)`	12	none	natural logarithm of a
`sqrt(`a`)`	12	none	square root of a
`sin(`a`)`	12	none	sine of a in radians (`cos` & `tan`)
`asin(`a`)`	12	none	arc sine of a (`acos` & `atan`)

CHAPTER 6

Manual Entries

This chapter contains the alphabetically ordered manual entries for the programs. The format follows that used on UNIX systems, and to be honest, it takes some getting used to. One possible source of confusion for users is the format of examples in the entries. The examples are chosen to work on UNIX using my preferred command shell, **ksh**, so some translation is needed for UNIX **csh** users, and for MSDOS users. See Chapter 3 on conventions used in the entries. Besides the manual entries, there is online help with most programs with the **-O** option. Information about limits, previously part of the entries, is now only available with the **-L** option.

Learning About the Programs. After learning how to use a few programs, it would be a good idea to skim the manual entries to see all the programs and their options. Besides the data manipulation and analysis programs, there are manual entries for special programs included in the |STAT distribution. **cat** is provided for MSDOS versions that do not have the corresponding UNIX program. The MSDOS **type** utility does not handle multiple files nor wildcards; **cat** does both. **ff** is a versatile text formatting filter that allows control of text filling to any width, right justification, line spacing, pagination, line numbering, tab expansion, and so on. **fpack** creates a plain text archive of a series of files. **fpack** can save space by reducing space wasted by many small files, and it can save time in file transfers by sending several files in one package.

Reading Manual Entries Online. The **manstat** program lets you read the manual entries online, assuming that they have been installed. To read the entry on a program, say **desc**, you just type:

```
manstat desc
```