© 1986 Gary Perlman

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
```

© 1986 Gary Perlman

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
```

© 1986 Gary Perlman

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:

© 1986 Gary Perlman

```
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**.

© 1986 Gary Perlman

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
```

© 1986 Gary Perlman

```
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.

© 1986 Gary Perlman

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
```

© 1986 Gary Perlman

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
```

© 1986 Gary Perlman

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).

© 1986 Gary Perlman

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"
```

© 1986 Gary Perlman

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.

© 1986 Gary Perlman

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.

© 1986 Gary Perlman

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**.

© 1986 Gary Perlman

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.

© 1986 Gary Perlman

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, **0**, is left associative if **x0x0x** is parsed as **(x0x)0x**, while one that is right associative is parsed as **x0(x0x)**.

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
```

© 1986 Gary Perlman

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
```

© 1986 Gary Perlman

CHAPTER 4

Data Manipulation

4.1 **Data Generation/Augmentation** 4-1
repeat: repeat a string or file
series: generate a linear series
probdist: generate random numbers
abut: number lines, recycle files
dm: number lines
4.2 **Data Transformation** 4-3
dm: conditional algebraic combinations of columns
probdist: probability/statistic conversion
ranksort: convert data to ranks
4.3 **Data Formatting** 4-5
maketrix: form a matrix format file
perm: permute lines
dsort: sort data lines by multiple keys
transpose: transpose matrix format file
reverse: reverse lines, columns, characters
colex: reorder columns, reformat columns
dm: reorder columns
abut: paste corresponding lines from files
4.4 **Data Extraction** 4-8
dm: conditional data extraction
colex: quick column extraction
linex: line extraction
4.5 **Data Validation** 4-9
validata: data validation
dm: conditional data validation
4.6 **DM: Tutorial and Manual** 4-10
Data Types
User Interface
Operations
Expression Syntax
Some Examples

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.

© 1986 Gary Perlman