Sassafras Manual

George Weigt

May 9, 2017

Contents

1 Introduction

Sassafras is a shell mode program for statistical analysis. The source tarball can be downloaded from `http://rococo.website/sassafras.tgz`. Then do the following to build and run the program.

```
make
./sassafras infile
```

Infile is a text file that tells the program what to do. The syntax is compatible with SAS-Language. There are "data steps" and "procedure steps." Data steps get data into the program and procedure steps compute the results. A data step begins with the keyword *data* and a procedure step begins with the keyword *proc*.

Example

A die, which may be loaded, is tossed six times. The observed point values are one to six. Compute a 95% confidence interval for the true mean μ given the observed data.

```
data
input y
datalines
1
2
3
4
5
6
;
proc means clm
```

The following result is displayed.

```
                    Variable     95% CLM MIN     95% CLM MAX
                    Y                  1.537           5.463
```

Here is the same result using R.

```
> y = c(1,2,3,4,5,6)
> t.test(y)

        One Sample t-test

data:  y
t = 4.5826, df = 5, p-value = 0.005934
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 1.536686 5.463314
```

2 Data Step

A data step is used to get data into the program.

`data` *name*

`infile "`*filename*`" dlm="`*delims*`" firstobs=`*n*

`input` *list*

var `=` *expression*

`datalines`

Notes

1. *name* is optional.
2. The `dlm` and `firstobs` settings are optional.
3. *delims* is a sequence of delimiter characters. The default is tab, comma, and space.
4. *n* is the starting input record number. Use `firstobs=2` to skip a header in the data file.
5. *list* is a list of variable names separated by spaces. For each categorical variable place a `$` after the variable name.
6. Optional *var* = *expression* statements create new vectors in the data set.
7. The `datalines` statement is followed by observational data. At the end of the data a blank line or a semicolon terminates the statement.

Example 1

The following example is a minimalist data step with in-line data.

```
data
input y
datalines
1
2
3
4
5
6
```

Example 2

Use `@@` at the end of an input statement to allow multiple values on an input line.

```
data
input y @@
datalines
```

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

Example 3

A dollar sign after an input variable indicates that the variable is categorical instead of numerical.

```
data
input trt $ y @@
datalines
A 6    A 0    A 2    A 8   A 11
A 4    A 13   A 1    A 8   A 0
B 0    B 2    B 3    B 1   B 18
B 4    B 14   B 9    B 1   B 9
C 13   C 10   C 18   C 5   C 23
C 12   C 5    C 16   C 1   C 20
```

Example 4

An infile statement is used to read data from a file.

```
data
input color $ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 y
infile "wine.txt"
```

Example 5

Expressions in a data step create new data vectors. The following example creates Y2 which is the input vector Y squared element-wise.

```
data
input color $ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 y
infile "wine.txt"
y2 = y ** 2
```

3 Anova Procedure

The anova procedure fits a classification model to data using ordinary least squares. The response variable must be numeric and the explanatory variables must be categorical.

`proc anova data=`*name*

`model` *y* `=` *list*

`means` *list*

`means` *list* `/ lsd ttest alpha=`*value*

Notes

1. `data=`*name* is optional. The default is data from the most recent data step.
2. *y* is the response variable which must be numeric.
3. *list* is one or more explanatory variables separated by spaces. The explanatory variables must be categorical. Interaction terms are specified using the syntax `A*B`.
4. The means statement can include one or more of the following options.

`lsd`	Compare treatment means using least significance difference
`ttest`	Compare treatment means using two sample t-test
`alpha`	Set the level of significance. Default is 0.05.

Example

```
data
input trt $ y @@
datalines
A 6    A 0    A 2    A 8   A 11
A 4    A 13   A 1    A 8   A 0
B 0    B 2    B 3    B 1   B 18
B 4    B 14   B 9    B 1   B 9
C 13   C 10   C 18   C 5   C 23
C 12   C 5    C 16   C 1   C 20

proc anova
model y = trt
means trt / lsd ttest
```

The following result is displayed.

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	2	293.60000000	146.80000000	3.98	0.0305
Error	27	995.10000000	36.85555556		
Total	29	1288.70000000			

R-Square	Coeff Var	Root MSE	Y Mean
0.227826	76.846553	6.070878	7.900000

Source	DF	Anova SS	Mean Square	F Value	Pr > F
TRT	2	293.60000000	146.80000000	3.98	0.0305

Mean Response

TRT	N	Mean Y	95% CI MIN	95% CI MAX
A	10	5.300000	1.360938	9.239062
B	10	6.100000	2.160938	10.039062
C	10	12.300000	8.360938	16.239062

Least Significant Difference Test

TRT	TRT	Delta Y	95% CI MIN	95% CI MAX	t Value	Pr > |t|	
A	B	-0.800000	-6.370676	4.770676	-0.29	0.7705	
A	C	-7.000000	-12.570676	-1.429324	-2.58	0.0157	*
B	A	0.800000	-4.770676	6.370676	0.29	0.7705	
B	C	-6.200000	-11.770676	-0.629324	-2.28	0.0305	*
C	A	7.000000	1.429324	12.570676	2.58	0.0157	*
C	B	6.200000	0.629324	11.770676	2.28	0.0305	*

Two Sample t-Test

TRT	TRT	Delta Y	95% CI MIN	95% CI MAX	t Value	Pr > |t|	
A	B	-0.800000	-5.922306	4.322306	-0.33	0.7466	
A	C	-7.000000	-12.664270	-1.335730	-2.60	0.0182	*
B	A	0.800000	-4.322306	5.922306	0.33	0.7466	
B	C	-6.200000	-12.467653	0.067653	-2.08	0.0523	
C	A	7.000000	1.335730	12.664270	2.60	0.0182	*
C	B	6.200000	-0.067653	12.467653	2.08	0.0523	

Mean response table

The confidence interval for a treatment mean is computed as follows.

$$\bar{y_i} \pm t(1-\alpha/2, dfe) \cdot \sqrt{\frac{MSE}{n_i}}$$

Recall that MSE is an estimate of model variance. From the anova table

Error	27	995.10000000	36.85555556

we obtain

$$MSE = 36.85555556$$
$$dfe = 27$$

Using R, the confidence interval for the mean of treatment A can be checked as follows.

```
> MSE = 36.8556
> dfe = 27
> t = qt(0.975,dfe)
> 5.3 - t * sqrt(MSE/10)
[1] 1.360934
> 5.3 + t * sqrt(MSE/10)
[1] 9.239066
```

Least significant difference test

The least significant difference of two means $\bar{y}_i$ and $\bar{y}_j$ is

$$LSD_{ij} = t(1-\alpha/2, dfe) \cdot \sqrt{MSE \cdot \left(\frac{1}{n_i} + \frac{1}{n_j}\right)}$$

The corresponding confidence interval is

$$\bar{y}_i - \bar{y}_j \pm LSD_{ij}$$

Two sample t-test

The two sample t-test is computed as follows.

$$\begin{aligned} SSE &= \widehat{Var}_i \cdot (n_i - 1) + \widehat{Var}_j \cdot (n_j - 1) \\ dfe &= n_i + n_j - 2 \\ MSE &= \frac{SSE}{dfe} \\ SE &= \sqrt{MSE \cdot \left(\frac{1}{n_i} + \frac{1}{n_j}\right)} \\ t^* &= \frac{\bar{y}_i - \bar{y}_j}{SE} \end{aligned}$$

SSE is the sum of squares error recovered from variance estimates, dfe is the degrees of freedom error, MSE is mean square error, SE is the standard error, and t^* is the test statistic. The confidence interval is

$$\bar{y}_i - \bar{y}_j \pm t(1-\alpha/2, dfe) \cdot SE$$

The null hypothesis is that the two treatment means are equal.

$$H_0 : \bar{y}_i = \bar{y}_j$$

If $|t^*|$ is greater than the critical value $t(1-\alpha/2, dfe)$, or equivalently, if the confidence interval does not cross zero, then reject H_0 and conclude that the treatment means are not equal. The following R session uses the above equations to duplicate the Sassafras result for treatments A and B.

```
> YA = c(6,0,2,8,11,4,13,1,8,0)
> YB = c(0,2,3,1,18,4,14,9,1,9)
> sse = var(YA) * (length(YA) - 1) + var(YB) * (length(YB) - 1)
> dfe = length(YA) + length(YB) - 2
```

```
> mse = sse / dfe
> se = sqrt(mse * (1 / length(YA) + 1 / length(YB)))
> t = (mean(YA) - mean(YB)) / se
> mean(YA) - mean(YB) - qt(0.975,dfe) * se
[1] -5.922307
> mean(YA) - mean(YB) + qt(0.975,dfe) * se
[1] 4.322307
> 2 * (1 - pt(abs(t),dfe))
[1] 0.746606
```

The same result is obtained with the t-test function.

```
> t.test(YA,YB,var.equal=TRUE)

        Two Sample t-test

data:  YA and YB
t = -0.3281, df = 18, p-value = 0.7466
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -5.922307  4.322307
sample estimates:
mean of x mean of y
      5.3       6.1
```

4 Means Procedure

The means procedure prints statistics about a data set.

`proc means data=`*name* `alpha=`*value* `maxdec=`*n* *stats*

`var` *list*

`class` *list*

Notes

1. The settings that follow the `means` keyword are optional. The settings can appear in any order.
2. If `data` is not specified then the default is data from the most recent data step.
3. `alpha` sets the level of significance. The default is 0.05.
4. `maxdec` sets the decimal precision in the output. n ranges from 0 to 8. The default is 3.
5. *stats* is a list of statistics keywords from the following table.

`clm`	Confidence limits of the mean
`max`	Maximum value
`mean`	Mean value
`min`	Minimum value
`n`	Number of observations
`range`	max − min
`std`	Standard deviation s
`stddev`	Another keyword for s
`stderr`	Standard error $s/\sqrt{n}$
`var`	Variance s^2

   If *stats* is not specified then the default list is `n mean std min max`.
6. The optional `var` statement specifies which variables to print. The default is all variables. Variable names in *list* are separated by spaces.
7. The optional `class` statement prints statistics for each level of the categorical variables in *list*. Variable names in *list* are separated by spaces.

Example 1

The following example reads in the wine[1] data set and shows the default action of proc means.

```
data wine
input color $ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 y
infile "wine.txt"

proc means
```

The following result is displayed.

[1] P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. *Modeling wine preferences by data mining from physicochemical properties.* In Decision Support Systems, Elsevier, 47(4):547-553, 2009.

```
Variable        N        Mean     Std Dev     Minimum     Maximum
X1           6497       7.215       1.296       3.800      15.900
X2           6497       0.340       0.165       0.080       1.580
X3           6497       0.319       0.145       0.000       1.660
X4           6497       5.443       4.758       0.600      65.800
X5           6497       0.056       0.035       0.009       0.611
X6           6497      30.525      17.749       1.000     289.000
X7           6497     115.745      56.522       6.000     440.000
X8           6497       0.995       0.003       0.987       1.039
X9           6497       3.219       0.161       2.720       4.010
X10          6497       0.531       0.149       0.220       2.000
X11          6497      10.492       1.193       8.000      14.900
Y            6497       5.818       0.873       3.000       9.000
```

Example 2

The following example adds a var statement to show Y by itself. Also, the desired statistics are specified.

```
data wine
input color $ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 y
infile "wine.txt"

proc means n mean clm
var y
```

The following result is displayed.

```
Variable        N      Mean     95% CLM MIN     95% CLM MAX
Y            6497     5.818           5.797           5.840
```

Example 3

The following example adds a class statement to show statistics for each wine color.

```
data wine
input color $ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 y
infile "wine.txt"

proc means n mean clm
var y
class color
```

The following result is displayed.

```
COLOR     Variable        N      Mean     95% CLM MIN     95% CLM MAX
red       Y            1599     5.636           5.596           5.676
white     Y            4898     5.878           5.853           5.903
```

5 Print Procedure

The print procedure prints data in a data set.

```
proc print data=name
var list
```

Notes

1. `data=`*name* is optional. The default is data from the most recent data step.
2. The optional `var` statement specifies which variables to print. The default is all variables. Variable names in *list* are separated by spaces.

Example

The following example reads a data set and prints it.

```
data
input trt $ y @@
datalines
A 6    A 0    A 2    A 8   A 11
A 4    A 13   A 1    A 8   A 0
B 0    B 2    B 3    B 1   B 18
B 4    B 14   B 9    B 1   B 9

proc print
```

The following result is displayed.

Obs	TRT	Y
1	A	6
2	A	0
3	A	2
4	A	8
5	A	11
6	A	4
7	A	13
8	A	1
9	A	8
10	A	0
11	B	0
12	B	2
13	B	3
14	B	1
15	B	18
16	B	4
17	B	14
18	B	9

19	B	1
20	B	9

6 Reg Procedure

The reg procedure fits a linear model to data using ordinary least squares. The response variable must be numeric. For models with no intercept, anova results will differ from R. This is because R switches to uncorrected sums of squares for models with no intercept.

`proc reg data=`*name*

`model` *y* `=` *list*

`model` *y* `=` *list* `/ noint`

Notes

1. `data=`*name* is optional. The default is data from the most recent data step.
2. *y* is the response variable which must be numeric.
3. *list* is a list of explanatory variables separated by spaces. If functions of explanatory variables are required then they must be defined in the data step.
4. The `noint` option fits a linear model with no intercept term.

Example 1

The following example reads in the wine data set and fits a linear model with no intercept term.

```
data
input color $ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 y
infile "wine.txt"

proc reg
model y = color x1 / noint
```

The following result is displayed.

```
                          Analysis of Variance

              DF     Sum of Squares     Mean Square     F Value     Pr > F
   Model       2           72.79210        36.39605       48.42     0.0000
   Error    6494         4880.89360         0.75160
   Total    6496         4953.68570

            Root MSE            0.86695     R-Square     0.0147
            Dependent Mean      5.81838     Adj R-Sq     0.0144
            Coeff Var          14.90018

                          Parameter Estimates

                     Estimate     Std Err     t Value     Pr > |t|
       COLOR red      5.77309     0.08194       70.45       0.0000
```

COLOR white	5.99084	0.06628	90.39	0.0000
X1	-0.01647	0.00950	-1.73	0.0829

Example 2

The following exercise is from *Econometrics*[2]. Using data from a 1963 paper by Marc Nerlove, estimate parameters for the model

$$\log(\text{COST}) = \beta_0 + \beta_1 \log(\text{KWH}) + \beta_2 \log(\text{PL}) + \beta_3 \log(\text{PF}) + \beta_4 \log(\text{PK}) + \varepsilon$$

where COST is production cost, KWH is kilowatt hours, PL is price of labor, PF is price of fuel, and PK is price of capital.

```
data
infile "nerlove.txt"
input COST KWH PL PF PK
LCOST = log(COST)
LKWH = log(KWH)
LPL = log(PL)
LPF = log(PF)
LPK = log(PK)

proc reg
model LCOST = LKWH LPL LPF LPK
```

The following result is displayed.

Analysis of Variance

	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	4	269.51482	67.37870	437.69	0.0000
Error	140	21.55201	0.15394		
Total	144	291.06683			

Root MSE	0.39236	R-Square	0.9260
Dependent Mean	1.72466	Adj R-Sq	0.9238
Coeff Var	22.74969		

Parameter Estimates

	Estimate	Std Err	t Value	Pr > |t|
Intercept	-3.52650	1.77437	-1.99	0.0488
LKWH	0.72039	0.01747	41.24	0.0000
LPL	0.43634	0.29105	1.50	0.1361
LPF	0.42652	0.10037	4.25	0.0000
LPK	-0.21989	0.33943	-0.65	0.5182

The following code can be pasted into R to obtain a similar result.

[2]Hansen, Bruce E. *Econometrics.* www.ssc.wisc.edu/∼bhansen

```
d = read.table("nerlove.txt")
lcost = log(d[,1])
lkwh = log(d[,2])
lpl = log(d[,3])
lpf = log(d[,4])
lpk = log(d[,5])
m = lm(lcost ~ lkwh + lpl + lpf + lpk)
summary(m)
```

The following result is displayed in R.

```
Call:
lm(formula = lcost ~ lkwh + lpl + lpf + lpk)

Residuals:
     Min       1Q   Median       3Q      Max
-0.97784 -0.23817 -0.01372  0.16031  1.81751

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -3.52650    1.77437  -1.987   0.0488 *
lkwh         0.72039    0.01747  41.244  < 2e-16 ***
lpl          0.43634    0.29105   1.499   0.1361
lpf          0.42652    0.10037   4.249 3.89e-05 ***
lpk         -0.21989    0.33943  -0.648   0.5182
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3924 on 140 degrees of freedom
Multiple R-squared:  0.926,        Adjusted R-squared:  0.9238
F-statistic: 437.7 on 4 and 140 DF,  p-value: < 2.2e-16
```

Example 3

The following model uses the "trees" data set from R.

```
data
input Girth Height Volume
LG = log(Girth)
LH = log(Height)
LV = log(Volume)
datalines
  8.3     70   10.3
  8.6     65   10.3
  8.8     63   10.2
 10.5     72   16.4
 10.7     81   18.8
 10.8     83   19.7
 11.0     66   15.6
 11.0     75   18.2
```

```
 11.1     80   22.6
 11.2     75   19.9
 11.3     79   24.2
 11.4     76   21.0
 11.4     76   21.4
 11.7     69   21.3
 12.0     75   19.1
 12.9     74   22.2
 12.9     85   33.8
 13.3     86   27.4
 13.7     71   25.7
 13.8     64   24.9
 14.0     78   34.5
 14.2     80   31.7
 14.5     74   36.3
 16.0     72   38.3
 16.3     77   42.6
 17.3     81   55.4
 17.5     82   55.7
 17.9     80   58.3
 18.0     80   51.5
 18.0     80   51.0
 20.6     87   77.0
```

```
proc reg
model LV = LG LH
```

The following result is displayed.

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	2	1.53213547	0.76606773	613.19	0.0000
Error	28	0.03498056	0.00124931		
Total	30	1.56711603			

Root MSE	0.03535	R-Square	0.9777
Dependent Mean	1.42133	Adj R-Sq	0.9761
Coeff Var	2.48679		

Parameter Estimates

Parameter	Estimate	Std Err	t Value	Pr > |t|
(Intercept)	-2.88007	0.34734	-8.29	0.0000
log(Girth)	1.98265	0.07501	26.43	0.0000
log(Height)	1.11712	0.20444	5.46	0.0000

Let us see if the above parameters correspond to the volume of a cone given by

$$V = \frac{\pi}{12} d^2 h$$

where d is the diameter (girth) and h is the height of the cone. The model from the regression is

$$\log V = -2.88 + 1.98 \log d + 1.12 \log h$$

Take the antilog of both sides and obtain

$$V = 0.00132 \times d^{1.98} \times h^{1.12}$$

The exponents resemble the volume formula but the overall coefficient 0.00132 is two orders of magnitude smaller than $\pi/12 \approx 0.262$. It turns out the discrepancy is due to the units of measure. Girth is measured in inches while height and volume are measured in feet. To convert girth from inches to feet requires a factor of 1/12. Hence the leading coefficient should be

$$\frac{\pi}{12} \times \frac{1}{144} \approx 0.00182$$

which is in the ballpark of 0.00132 from the regression model.

Let us compare the Reg results to R. The following block of code can be pasted directly into the R shell prompt.

```
d=log10(trees[,1])
h=log10(trees[,2])
V=log10(trees[,3])
m=lm(V~d+h)
summary(m)
```

This is the R result, which matches Reg.

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -2.88007    0.34734  -8.292 5.06e-09 ***
d            1.98265    0.07501  26.432  < 2e-16 ***
h            1.11712    0.20444   5.464 7.81e-06 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.03535 on 28 degrees of freedom
Multiple R-squared:  0.9777,        Adjusted R-squared:  0.9761
F-statistic: 613.2 on 2 and 28 DF,  p-value: < 2.2e-16
```

7 Review

Analysis of Variance

The components of an analysis of variance table are computed as follows.

	DF	SS	Mean Square	F-value	p-value
Model	$p-1$	SSR	$MSR = SSR/(p-1)$	$F^* = MSR/MSE$	$1 - F(F^*, p-1, n-p)$
Error	$n-p$	SSE	$MSE = SSE/(n-p)$		
Total	$n-1$	SST			

In the table, n is the number of observations and p is the number of model parameters including the intercept term if there is one. The sums of squares are computed as follows.

$$SSR = \sum (\hat{y}_i - \bar{y})^2$$
$$SSE = \sum (y_i - \hat{y}_i)^2$$
$$SST = \sum (y_i - \bar{y})^2$$

Recall that MSE is an estimate of model variance.

$$MSE = \hat{\sigma}^2$$

A simple way to model the response variable is to use the average $\bar{y}$. The p-value above indicates whether or not the regression model is better than $\bar{y}$. The null hypothesis is that the regression model is no better than the average, that is

$$H_0 : SST = SSE$$

The test for H_0 is known as an omnibus test because an equivalent hypothesis is

$$H_0 : \beta_1 = \beta_2 = \cdots = \beta_{p-1} = 0$$

Under H_0 we have $SSR = 0$ hence another equivalent hypothesis is

$$H_0 : F^* = 0$$

The test statistic F^* is used because it has a well-known distribution. Recall that the p-value is (loosely) the probability that H_0 is true. Hence for small p-values, reject H_0 and conclude that the regression model is better than $\bar{y}$.

Confidence interval of the mean

The confidence interval of the mean is

$$\bar{x} \pm t_{1-\alpha/2, n-1} \frac{s}{\sqrt{n}}$$

where $\bar{x}$ is the observed mean, s is the observed standard deviation, n is the number of observations, and $t_{1-\alpha/2,n-1}$ is the quantile function. In R, the confidence interval of the mean of `1:10` can be computed as follows.

```
> x = 1:10
> n = length(x)
> alpha = 0.05
> mean(x) - qt(1-alpha/2,n-1) * sd(x)/sqrt(n)
[1] 3.334149
> mean(x) + qt(1-alpha/2,n-1) * sd(x)/sqrt(n)
[1] 7.665851
```

Alternatively, the `t.test` function can be used.

```
> t.test(1:10)

        One Sample t-test

data:  1:10
t = 5.7446, df = 9, p-value = 0.0002782
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 3.334149 7.665851
sample estimates:
mean of x
      5.5
```

Recall that the quantile function is the inverse of the cumulative distribution function. Let F be the cumulative distribution function. Then

$$F(t_{1-\alpha/2,n-1}) = 1 - \alpha/2$$

For example, in R we have

```
> t = qt(0.975,8)
> t
[1] 2.306004
> pt(t,8)
[1] 0.975
```

8 Example infiles

8.1 alfalfa-demo.txt

```
* Compare yields of four types of alfalfa.
* This is a randomized complete block design.

* References
* 1. "Statistical Design" by G. Casella, p. 97.
* 2. http://www.stat.ufl.edu/~casella/StatDesign/WebDataSets/Alfalfa.txt

data
input Variety $ Block $ Rep $ Yield
datalines
Ladak    1   1   3.1
Ladak    2   1   4.06
Ladak    3   1   4.73
Ladak    4   1   3.1
Ladak    1   2   3.25
Ladak    2   2   4.26
Ladak    3   2   4.71
Ladak    4   2   4.21
Ladak    1   3   3.86
Ladak    2   3   4.53
Ladak    3   3   5.26
Ladak    4   3   3.84
Narrag   1   1   4.65
Narrag   2   1   5.64
Narrag   3   1   4.94
Narrag   4   1   5.38
Narrag   1   2   5.46
Narrag   2   2   5.48
Narrag   3   2   5.26
Narrag   4   2   5.68
Narrag   1   3   4.21
Narrag   2   3   5.09
Narrag   3   3   5.8
Narrag   4   3   5.82
DuPuits  1   1   5.47
DuPuits  2   1   5.62
DuPuits  3   1   6.71
DuPuits  4   1   6.87
DuPuits  1   2   6.41
DuPuits  2   2   6.3
DuPuits  3   2   6.96
DuPuits  4   2   6.28
DuPuits  1   3   5.57
DuPuits  2   3   6.46
DuPuits  3   3   5.92
DuPuits  4   3   6.46
Flamand  1   1   6.85
Flamand  2   1   6.33
Flamand  3   1   6.88
Flamand  4   1   6.23
Flamand  1   2   6.34
Flamand  2   2   5.83
Flamand  3   2   6.59
Flamand  4   2   6.52
Flamand  1   3   5.45
```

```
Flamand 2    3    4.33
Flamand 3    3    6.06
Flamand 4    3    6.81

proc anova
model Yield = Block Variety Block*Variety
means Variety / lsd
```

8.2 corrosion-demo.txt

```
* Reference
* "Split-Plot Designs: What, Why, and How" by Bradley Jones, Christopher L. Nachtsheim
* jmp.info/software/pdf/30612.pdf

* W - Whole plot (furnace)
* T - Temperature
* C - Coating
* Y - Corrosion resistance of steel bars

data
input W $ T $ C $ Y
datalines
1 360 C2 73
1 360 C3 83
1 360 C1 67
1 360 C4 89
2 370 C1 65
2 370 C3 87
2 370 C4 86
2 370 C2 91
3 380 C3 147
3 380 C1 155
3 380 C2 127
3 380 C4 212
4 380 C4 153
4 380 C3 90
4 380 C2 100
4 380 C1 108
5 370 C4 150
5 370 C1 140
5 370 C3 121
5 370 C2 142
6 360 C1 33
6 360 C4 54
6 360 C2 8
6 360 C3 46

proc anova
model Y = T C T*C W
means C / lsd

* Proc anova computes F(T) = MS(T)/MSE = 106.47
* Correct value is F(T) = MS(T)/MS(W) = 2.75 and
* corresponding p value is 0.209
```

8.3 diet-demo.txt

```
* This is a split plot design.

* References
* 1. "Statistical Design" by G. Casella, p. 171.
* 2. http://www.stat.ufl.edu/~casella/StatDesign/WebDataSets/Diet.txt

data
input Diet $ Subject $ Time $ BP
datalines
1    1    AM    123
1    1    PM    135
1    2    AM    120
1    2    PM    136
1    3    AM    122
1    3    PM    129
2    4    AM    117
2    4    PM    139
2    5    AM    125
2    5    PM    136
2    6    AM    122
2    6    PM    142
3    7    AM    114
3    7    PM    123
3    8    AM    109
3    8    PM    132
3    9    AM    115
3    9    PM    132
4    10   AM    140
4    10   PM    150
4    11   AM    141
4    11   PM    147
4    12   AM    138
4    12   PM    154

proc anova
model BP = Diet Subject*Diet Time Time*Diet
means Diet / lsd

* Subject is nested within Diet.

* Replacing Subject*Diet with just Subject yields
* the same result because each subject is uniquely
* identified.

* Anova in the book includes Time*Subject*Diet.
* It turns out that
* MS(Time*Subject*Diet) = MSE = 16.17

* Proc anova computes F(*) = MS(*)/MSE
* Correct F(Diet) = MS(Diet)/MS(Subject*Diet)
* Hence F(Diet) = 624.49/7.33 = 85.20
```

8.4 dry-weight-demo.txt

```
* Compare dry weights of geraniums grown with three types of fertilizer.

* References
* 1. "Statistical Design" by G. Casella, p. 2.
* 2. http://www.stat.ufl.edu/~casella/StatDesign/WebDataSets/DryWeight.txt

data
input Fert $ DryWeight
datalines
A       1.02
A       0.79
A       1
A       0.59
A       0.97
B       1
B       1.21
B       1.22
B       0.96
B       0.79
C       0.99
C       1.36
C       1.22
C       1.12
C       1.17

proc anova
model DryWeight = Fert
means Fert / lsd
```

8.5 fish-tank-demo.txt

```
* Compare fish weight gains for three types of diets.
* Tanks are nested within diets.

* References
* 1. "Statistical Design" by G. Casella, p. 6.
* 2. http://www.stat.ufl.edu/~casella/StatDesign/WebDataSets/FishTank.txt

data
input Diet $ Tank $ WtGain
datalines
1       1       9.759
1       1       7.399
1       1       -0.209
1       1       2.204
1       1       0.267
1       1       3.002
1       2       8.031
1       2       6.25
1       2       0.134
1       2       4.594
1       2       4.414
1       2       9.816
1       3       -2.23
1       3       9.69
1       3       -2.239
1       3       9.499
1       3       6.927
1       3       3.449
1       4       4.274
1       4       7.52
1       4       12.141
1       4       4.828
1       4       3.391
1       4       3.621
2       5       21.819
2       5       6.503
2       5       31.596
2       5       24.633
2       5       15.73
2       5       22.231
2       6       11.672
2       6       26.479
2       6       19.784
2       6       20.884
2       6       21.811
2       6       26.344
2       7       22.161
2       7       16.429
2       7       23.311
2       7       21.983
2       7       12.181
2       7       18.252
2       8       30.865
2       8       17.875
2       8       24.562
2       8       20.442
2       8       20.791
2       8       19.44
```

```
3       9       47.588
3       9       38.219
3       9       44.445
3       9       47.115
3       9       30.83
3       9       58.708
3       10      62.477
3       10      29.834
3       10      49.902
3       10      59.946
3       10      46.587
3       10      38.567
3       11      54.545
3       11      38.846
3       11      52.163
3       11      43.141
3       11      27.822
3       11      58.025
3       12      57.749
3       12      47.737
3       12      40.653
3       12      45.344
3       12      46.702
3       12      50.091

proc anova
model WtGain = Diet Tank*Diet
means Diet / lsd
```

8.6 ivd-demo.txt

```
* Analyze in vitro digestibility (IVD) of dried alfalfa grown at different temperatures.
* This is a one-way completely randomized design.

* References:
* 1. "Statistical Design" by G. Casella, p. 44.
* 2. http://www.stat.ufl.edu/~casella/StatDesign/WebDataSets/IVD.txt

data
input Temp $ IVD
datalines
17    94.2
17    94.5
17    95
17    94.7
22    94.5
22    94
22    94.6
22    94
27    95.1
27    95.7
27    95.5
27    96.1
32    95.2
32    96
32    96.1
32    95.3

proc anova
model IVD = Temp
means Temp / lsd
```

8.7 lmc-demo.txt

```
* Distance to the Large Magellanic Cloud

* Dataset from http://astrostatistics.psu.edu/datasets/LMC_distance.dat

* TYPE is population type
* DM is distance modulum
* PM is +/-

data
input TYPE $ DM PM
datalines
I 18.70      0.16    Feast & Catchpole 1997
I 18.55      0.06    Laney & Stobie 1994
I 18.55      0.10    Gieren et al. 1998, Di Benedetto 1997
I 18.575     0.2     Groenewegen 2000
I 18.4       0.1     Ribas et al. 2002
I 18.42      0.07    This paper see \S 6.2
I 18.45      0.07    This paper see \S 6.2
I 18.59      0.09    Romaniello et al. 2000
I 18.471     0.12    Pietrzynsky \& Gieren 2002
I 18.54      0.10    Sarajedini et al. 2002
I 18.54      0.18    Van Leeuwen 1997
I 18.64      0.14    Feast 2000
I 18.58      0.05    Panagia 1998
II 18.45      0.09   Gratton et al. 2002 and this paper
II 18.45      0.13   Carretta et al. 2000b and this paper
II 18.44      0.13   Carretta et al. 2000b and this paper
II 18.30      0.14   This paper see \S 7
II 18.38      0.16   This paper see \S 7
II 18.50      0.16   Cacciari et al. 2000 and this paper
II 18.55      0.19   A97 and A00
II 18.52      0.18   Kovacs 2000 and this paper
II 18.40      0.15   Gratton et al. 2002 and this paper
II 18.45      0.16   Benedict et al. 2002 and this paper
II 18.55      0.09   Cioni et al. 2000
II 18.69      0.26   Romaniello et al. 2000.

* Does DM depend on TYPE? (Answer: No)
proc anova
model DM = TYPE
means TYPE / lsd

* Does PM depend on TYPE? (Answer: Yes)
proc anova
model PM = TYPE
means TYPE / lsd
```

8.8 peanut-demo.txt

```
* Compare peanut yields for four treatments.
* This is a latin square design.

* References
* 1. "Statistical Design" by G. Casella, p. 118.
* 2. http://www.stat.ufl.edu/~casella/StatDesign/WebDataSets/Peanut.txt

data
input Row $ Column $ Treatment $ Yield
datalines
1   1   3   26.7
2   1   1   23.1
3   1   2   28.3
4   1   4   25.1
1   2   1   19.7
2   2   2   20.7
3   2   4   20.1
4   2   3   17.4
1   3   2   28
2   3   4   24.9
3   3   3   29
4   3   1   28.7
1   4   4   29.8
2   4   3   29
3   4   1   27.3
4   4   2   34.1

proc anova
model Yield = Row Column Treatment
means Treatment / lsd
```

8.9 prof-hunter-demo.txt

```
* Comparision of two experimental designs.

* Reference
* "Design of Experiments" learning series by Professor J. Stuart Hunter, Lesson 12.
* http://www.youtube.com/watch?v=k3n9iSB6Cns
* http://www.youtube.com/watch?v=3fwoU16MHJM
*
* A list of all videos in the series:
* http://www.youtube.com/playlist?list=PLzaIpDs2EIQgXzVAPS9rviC9J7Cpyet5y
*
* Below are the anova tables seen in the video.
*
* CSS - corrected sum of squares
* TSS - treatment sum of squares
* BSS - block sum of squares
* RSS - residual sum of squares
*
* 1st anova table
*                    DF      MS
* CSS      112.0055  19
* TSS        0.8405   1
* RSS      111.1650  18  6.1758
*
* 2nd anova table
*                    DF      MS
* CSS      112.0055  19
* TSS        0.8405   1
* BSS      110.4905   9
* RSS        0.6745   9  0.0750
*
* 2nd design reduces RSS.
*
*  I  Gasoline with additive
* II  Gasoline without additive
*  y  Miles per 5 gallons

title "Completely Randomized Design"

data
input trt $ y
datalines
I 74.0
I 68.8
I 71.2
I 74.2
I 71.8
I 66.4
I 69.8
I 71.3
I 69.3
I 73.6
II 73.2
II 68.2
II 70.9
II 74.3
II 70.7
II 66.6
II 69.5
```

```
II 70.8
II 68.8
II 73.3

proc anova
model y = trt
means trt / lsd

title "Randomized Block Design"

data
input car $ trt $ y
datalines
A I 74.0
B I 68.8
C I 71.2
D I 74.2
E I 71.8
F I 66.4
G I 69.8
H I 71.3
I I 69.3
J I 73.6
A II 73.2
B II 68.2
C II 70.9
D II 74.3
E II 70.7
F II 66.6
G II 69.5
H II 70.8
I II 68.8
J II 73.3

proc anova
model y = car trt
means trt / lsd
```

8.10 red-clover-demo.txt

```
* Compare red clover yields for two levels of nitrogen and four levels of sulphur.
* This is a two-way completely randomized design.

* References
* 1. "Statistical Design" by G. Casella, p. 46.
* 2. http://www.stat.ufl.edu/~casella/StatDesign/WebDataSets/RedClover.txt

data
input Nitrogen $ Sulphur $ Yield
datalines
0    0    4.48
0    0    4.52
0    0    4.63
0    3    4.7
0    3    4.65
0    3    4.57
0    6    5.21
0    6    5.23
0    6    5.28
0    9    5.88
0    9    5.98
0    9    5.88
20   0    5.76
20   0    5.72
20   0    5.78
20   3    7.01
20   3    7.11
20   3    7.02
20   6    5.88
20   6    5.82
20   6    5.73
20   9    6.26
20   9    6.26
20   9    6.37

proc anova
model Yield = Sulphur Nitrogen Sulphur*Nitrogen
means Sulphur*Nitrogen
```

8.11 seedling-demo.txt

```
* Compare heights of seedlings grown from seeds from five forests.
* This is a nested design. Source tree is nested within forest.

* Reference
* http://www.ohio.edu/plantbio/staff/mccarthy/quantmet/lectures/ANOVA-III.pdf

data
input Forest $ Tree $ Height
datalines
A T1 15.8
A T1 15.6
A T1 16.0
A T2 13.9
A T2 14.2
A T2 13.5
B T3 18.5
B T3 18.0
B T3 18.4
B T4 17.9
B T4 18.1
B T4 17.4
C T5 12.3
C T5 13.0
C T5 12.7
C T6 14.0
C T6 13.1
C T6 13.5
D T7 19.5
D T7 17.5
D T7 19.1
D T8 18.7
D T8 19.0
D T8 18.8
E T9 16.0
E T9 15.7
E T9 16.1
E T0 15.8
E T0 15.6
E T0 16.3

proc anova
model Height = Forest Tree*Forest
means Forest / lsd
```

8.12 strawberry-demo.txt

```
* Compare yields for three types of strawberries.
* This is a randomized complete block design.

* References
* 1. "Statistical Design" by G. Casella, p. 9 and 94.
* 2. http://www.stat.ufl.edu/~casella/StatDesign/WebDataSets/Strawberry.txt

data
input Block $ Treatment $ Yield
datalines
1   A   10.1
1   B   6.3
1   C   8.4
2   A   10.8
2   B   6.9
2   C   9.4
3   A   9.8
3   B   5.3
3   C   9
4   A   10.5
4   B   6.2
4   C   9.2

proc anova
model Yield = Block Treatment
means Treatment / lsd
```

8.13 trees-demo.txt

```
data
input Girth Height Volume
LG = log(Girth)
LH = log(Height)
LV = log(Volume)
datalines
   8.3      70    10.3
   8.6      65    10.3
   8.8      63    10.2
  10.5      72    16.4
  10.7      81    18.8
  10.8      83    19.7
  11.0      66    15.6
  11.0      75    18.2
  11.1      80    22.6
  11.2      75    19.9
  11.3      79    24.2
  11.4      76    21.0
  11.4      76    21.4
  11.7      69    21.3
  12.0      75    19.1
  12.9      74    22.2
  12.9      85    33.8
  13.3      86    27.4
  13.7      71    25.7
  13.8      64    24.9
  14.0      78    34.5
  14.2      80    31.7
  14.5      74    36.3
  16.0      72    38.3
  16.3      77    42.6
  17.3      81    55.4
  17.5      82    55.7
  17.9      80    58.3
  18.0      80    51.5
  18.0      80    51.0
  20.6      87    77.0

proc reg
model LV = LG LH
```

9 Source Code

9.1 defs.h

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <setjmp.h>
#define __USE_ISOC99
#include <math.h>
#ifndef NAN
#define NAN nan("0")
#endif

#define MAXVAR 100
#define MAXSTAT 12

struct dataset {
      struct dataset *next;
      char *name;
      int nvar;
      int nobs;
      int max;
      struct spec {
            char *name;
            int max_levels;
            int num_levels;
            char **ltab;
            int w; // number of decimal digits after decimal point
            double *v;
      } spec[MAXVAR];
};

// tokens

#define NAME 1001
#define NUMBER 1002
#define STRING 1003
#define ATAT 1004
#define STARSTAR 1005

// keywords

enum {
      KALPHA = 301,
      KANOVA,
      KBY,
      KCARDS,
      KCLASS,
      KCLM,
      KCLM1,
      KCLM2,
      KDATA,
      KDATALINES,
      KDELIMITER,
      KDLM,
      KFIRSTOBS,
      KINFILE,
```

```
56          KINPUT,
57          KLCLM,
58          KLSD,
59          KMAX,
60          KMAXDEC,
61          KMEAN,
62          KMEANS,
63          KMIN,
64          KMODEL,
65          KN,
66          KNOINT,
67          KPRINT,
68          KPROC,
69          KRANGE,
70          KREG,
71          KRUN,
72          KSTD,
73          KSTDDEV,
74          KSTDERR,
75          KSTDMEAN,
76          KSUM,
77          KT,
78          KTITLE,
79          KTITLE1,
80          KTITLE2,
81          KTITLE3,
82          KTTEST,
83          KUCLM,
84          KVAR,
85          KWELCH,
86  };
87
88  extern struct dataset *dataset;
89  extern char pgm[];
90  extern char *inp;
91  extern char *token_str;
92  extern int token;
93  extern char errbuf[];
94  extern char strbuf[];
95  extern double token_num;
96  extern FILE *infile;
97
98  extern double alpha;
99  extern int maxdec;
100 extern int nstat;
101 extern int stat[];
102 extern int nvar;
103 extern int var[];
104 extern int nby;
105 extern int by[];
106 extern int nclass;
107 extern int class[];
108 extern char *title;
109 extern char *title1;
110 extern char *title2;
111 extern char *title3;
112 extern char *prefix;
113
114 #include "prototypes.h"
```

9.2 main.c

```
1  #include "defs.h"
2
3  char pgm[10000];
4  char errbuf[1000];
5  char *inp;
6  char *token_str;
7  int token;
8  double token_num;
9  char *prefix = ".";
10 jmp_buf jmpbuf;
11
12 struct dataset *dataset;
13 double alpha;
14 int maxdec;
15 char *title;
16 char *title1;
17 char *title2;
18 char *title3;
19 int nby;
20 int by[MAXVAR];
21 int nclass;
22 int class[MAXVAR];
23 int nstat;
24 int stat[MAXSTAT];
25 int nvar;
26 int var[MAXVAR];
27
28 int
29 main(int argc, char **argv)
30 {
31       int n;
32       FILE *f;
33       if (argc < 2) {
34             printf("missing infile\n");
35             return 0;
36       }
37       f = fopen(argv[1], "r");
38       if (f == NULL) {
39             printf("cannot open infile\n");
40             return 0;
41       }
42       n = fread(pgm, 1, sizeof pgm, f);
43       fclose(f);
44       if (n == sizeof pgm) {
45             printf("infile exceeds max\n");
46             return 0;
47       }
48       pgm[n] = 0;
49       run();
50       return 0;
51 }
52
53 void
54 run()
55 {
56       run1();
57       if (infile) {
58             fclose(infile);
```

```
59              infile = NULL;
60      }
61      free_datasets();
62  }
63
64  void
65  run1()
66  {
67      if (setjmp(jmpbuf))
68          return;
69      inp = pgm;
70      scan();
71      for (;;) {
72          keyword();
73          switch (token) {
74          case 0:
75              return;
76          case KDATA:
77              data_step();
78              break;
79          case KPROC:
80              procedure_step();
81              break;
82          case KRUN:
83              scan();
84              if (token != ';')
85                  stop("';'_expected");
86              scan();
87              break;
88          default:
89              parse_default();
90              break;
91          }
92      }
93  }
94
95  void
96  procedure_step(void)
97  {
98      nby = 0;
99      nvar = 0;
100     nstat = 0;
101     nclass = 0;
102
103     maxdec = 3;
104     alpha = 0.05;
105
106     select_dataset(NULL);
107
108     scan(); // get token after PROC
109
110     keyword();
111
112     switch (token) {
113     case KANOVA:
114         proc_anova();
115         break;
116     case KMEANS:
117         proc_means();
118         break;
```

```
119         case KPRINT:
120             proc_print();
121             break;
122         case KREG:
123             proc_reg();
124             break;
125         default:
126             expected("procedure name");
127         }
128 }
129
130 void
131 parse_default()
132 {
133         switch (token) {
134         case ';':
135             scan();
136             break;
137         case '*':
138             parse_comment_stmt();
139             break;
140         case KTITLE:
141             parse_title_stmt();
142             break;
143         case KTITLE1:
144             parse_title1_stmt();
145             break;
146         case KTITLE2:
147             parse_title2_stmt();
148             break;
149         case KTITLE3:
150             parse_title3_stmt();
151             break;
152         default:
153             stop("Unexpected token");
154         }
155 }
156
157 void
158 parse_comment_stmt(void)
159 {
160         while (*inp) {
161
162             // semicolon or end of line terminates comment statement
163
164             if (*inp == ';' || *inp == '\n' || *inp == '\r')
165                 break;
166
167             inp++;
168         }
169
170         scan();
171 }
172
173 void
174 parse_title_stmt()
175 {
176         scan();
177         if (token != STRING)
178             expected("string");
```

```
179         if (title)
180             free(title);
181         if (strbuf[0] == 0)
182             title = NULL;
183         else
184             title = strdup(strbuf);
185         scan();
186         if (token != ';')
187             stop("';' expected");
188         scan();
189     }
190
191     void
192     parse_title1_stmt()
193     {
194         scan();
195         if (token != STRING)
196             expected("string");
197         if (title1)
198             free(title1);
199         if (strbuf[0] == 0)
200             title1 = NULL;
201         else
202             title1 = strdup(strbuf);
203         scan();
204         if (token != ';')
205             stop("';' expected");
206         scan();
207     }
208
209     void
210     parse_title2_stmt()
211     {
212         scan();
213         if (token != STRING)
214             expected("string");
215         if (title2)
216             free(title2);
217         if (strbuf[0] == 0)
218             title2 = NULL;
219         else
220             title2 = strdup(strbuf);
221         scan();
222         if (token != ';')
223             stop("';' expected");
224         scan();
225     }
226
227     void
228     parse_title3_stmt()
229     {
230         scan();
231         if (token != STRING)
232             expected("string");
233         if (title3)
234             free(title3);
235         if (strbuf[0] == 0)
236             title3 = NULL;
237         else
238             title3 = strdup(strbuf);
```

```
239         scan();
240         if (token != ';')
241             stop("';' expected");
242         scan();
243     }
244
245     void
246     parse_alpha_option()
247     {
248         scan(); // skip alpha token
249
250         if (token != '=')
251             expected("equals sign");
252
253         scan(); // skip equals sign
254
255         if (token != NUMBER)
256             expected("number");
257
258         alpha = atof(strbuf);
259
260         if (alpha < 0 || alpha > 1)
261             expected("number between 0 and 1");
262     }
263
264     void
265     parse_data_option()
266     {
267         scan();
268         if (token != '=')
269             stop("'=' expected");
270         scan();
271         if (token != NAME)
272             stop("DATA=NAME expected");
273         select_dataset(strbuf);
274     }
275
276     void
277     parse_maxdec_option()
278     {
279         scan();
280         if (token != '=')
281             stop("'=' expected");
282
283         scan();
284         if (token != NUMBER)
285             stop("Number expected");
286
287         maxdec = (int) token_num;
288
289         if (maxdec < 0)
290             maxdec = 0;
291
292         if (maxdec > 8)
293             maxdec = 8;
294     }
295
296     void
297     parse_by_stmt(void)
298     {
```

```
299         int i;
300
301         for (;;) {
302
303             scan();
304
305             if (token == ';')
306                 break;
307
308             if (token != NAME)
309                 stop("Variable name expected");
310
311             // look for match in dataset
312
313             for (i = 0; i < dataset->nvar; i++)
314                 if (strcmp(dataset->spec[i].name, strbuf) == 0)
315                     break;
316
317             if (i == dataset->nvar) {
318                 sprintf(errbuf, "The variable %s not in the dataset", strbuf);
319                 stop(errbuf);
320             }
321
322             if (dataset->spec[i].ltab == NULL) {
323                 sprintf(errbuf, "The variable %s is not a categorical variable", strbuf);
324                 stop(errbuf);
325             }
326
327             by[nby++] = i;
328         }
329
330         scan(); // eat the semicolon
331     }
332
333     void
334     parse_class_stmt(void)
335     {
336         int i;
337
338         for (;;) {
339
340             scan();
341
342             if (token == ';')
343                 break;
344
345             if (token != NAME)
346                 stop("Variable name expected");
347
348             // look for match in dataset
349
350             for (i = 0; i < dataset->nvar; i++)
351                 if (strcmp(dataset->spec[i].name, strbuf) == 0)
352                     break;
353
354             if (i == dataset->nvar) {
355                 sprintf(errbuf, "The variable %s not in the dataset", strbuf);
356                 stop(errbuf);
357             }
358
```

```
359             if (dataset->spec[i].ltab == NULL) {
360                 sprintf(errbuf, "The variable %s is not a categorical variable", strbuf);
361                 stop(errbuf);
362             }
363
364             class[nclass++] = i;
365         }
366
367         scan(); // eat the semicolon
368     }
369
370     void
371     parse_var_stmt(void)
372     {
373         int i;
374
375         for (;;) {
376
377             scan();
378
379             if (token == ';')
380                 break;
381
382             if (token != NAME)
383                 stop("Variable name expected");
384
385             // look for match in dataset
386
387             for (i = 0; i < dataset->nvar; i++)
388                 if (strcmp(dataset->spec[i].name, strbuf) == 0)
389                     break;
390
391             if (i == dataset->nvar) {
392                 sprintf(errbuf, "Variable %s?", strbuf);
393                 stop(errbuf);
394             }
395
396             var[nvar++] = i;
397         }
398
399         scan(); // eat the semicolon
400     }
401
402     #define A(i, j) (a + (i) * ncol)[j]
403
404     void
405     print_table_and_free(char **a, int nrow, int ncol, char *fmt)
406     {
407         int i, j;
408         print_table(a, nrow, ncol, fmt);
409         for (i = 0; i < nrow; i++)
410             for (j = 0; j < ncol; j++)
411                 free(A(i, j));
412         free(a);
413     }
414
415     void
416     print_table(char **a, int nrow, int ncol, char *fmt)
417     {
418         int c, i, j, k, n, nsp = 0, t, *w;
```

```
419         char *b, *buf, *s;
420
421         w = (int *) xmalloc(ncol * sizeof (int));
422
423         // measure column widths
424
425         t = 0;
426
427         for (j = 0; j < ncol; j++) {
428               w[j] = 0;
429               for (i = 0; i < nrow; i++) {
430                     s = A(i, j);
431                     if (s == NULL)
432                           continue;
433                     n = (int) strlen(s);
434                     if (n > w[j])
435                           w[j] = n;
436               }
437               t += w[j];
438         }
439
440         // spaces between columns
441
442         if (ncol > 1) {
443               nsp = (80 - t) / (ncol - 1);
444               if (nsp < 2)
445                     nsp = 2;
446               if (nsp > 5)
447                     nsp = 5;
448               t += (ncol - 1) * nsp;
449         }
450
451         // number of spaces to center the line
452
453         if (t < 80)
454               c = (80 - t) / 2;
455         else
456               c = 0;
457
458         // alloc line buffer
459
460         buf = (char *) xmalloc(c + t + 1);
461
462         // leading spaces
463
464         memset(buf, ' ', c);
465
466         // for each row
467
468         for (i = 0; i < nrow; i++) {
469
470               b = buf + c;
471
472               // for each column
473
474               for (j = 0; j < ncol; j++) {
475
476                     // space between columns
477
478                     if (j > 0) {
```

```
479                         memset(b, ' ', nsp);
480                         b += nsp;
481                 }
482
483                 s = A(i, j);
484
485                 if (s == NULL) {
486                         memset(b, ' ', w[j]);
487                         b += w[j];
488                 } else {
489
490                         n = (int) strlen(s);
491
492                         k = w[j] - n;
493
494                         switch (fmt[j]) {
495
496                         // right aligned
497
498                         case 0:
499                                 memset(b, ' ', k);
500                                 b += k;
501                                 strcpy(b, s);
502                                 b += n;
503                                 break;
504
505                         // left aligned
506
507                         case 1:
508                                 strcpy(b, s);
509                                 b += n;
510                                 memset(b, ' ', k);
511                                 b += k;
512                                 break;
513                         }
514                 }
515         }
516
517         *b = 0;
518
519         emit_line(buf);
520     }
521
522     free(buf);
523     free(w);
524
525     emit_line("");
526 }
527
528 void
529 print_title()
530 {
531     if (title)
532         emit_line_center(title);
533
534     if (title1)
535         emit_line_center(title1);
536
537     if (title2)
538         emit_line_center(title2);
```

```
539
540         if (title3)
541                 emit_line_center(title3);
542
543         if (title || title1 || title2 || title3)
544                 emit_line("");
545 }
546
547 void
548 emit_line(char *s)
549 {
550         printf("%s\n", s);
551 }
552
553 void
554 emit_line_center(char *s)
555 {
556         int i, n;
557
558         n = strlen(s);
559
560         n = (80 - n) / 2;
561
562         for (i = 0; i < n; i++)
563                 printf(" ");
564
565         printf("%s\n", s);
566 }
567
568 void *
569 xmalloc(int size)
570 {
571         void *p = malloc(size);
572         if (p == NULL)
573                 stop("Out of memory");
574         return p;
575 }
576
577 void *
578 xcalloc(int size)
579 {
580         void *p = calloc(1, size);
581         if (p == NULL)
582                 stop("Out of memory");
583         return p;
584 }
585
586 void *
587 xrealloc(void *p, int size)
588 {
589         p = realloc(p, size);
590         if (p == NULL)
591                 stop("Out of memory");
592         return p;
593 }
594
595 void
596 print_pgm()
597 {
598         char c, *s, *t;
```

```
599
600         s = pgm;
601
602         for (;;) {
603
604             while (*s && s != inp && (*s == '\n' || *s == '\r'))
605                 s++;
606
607             if (*s == 0 || s == inp)
608                 break;
609
610             t = s;
611
612             while (*s && s != inp && *s != '\n' && *s != '\r')
613                 s++;
614
615             c = *s;
616             *s = 0;
617             emit_line(t);
618             *s = c;
619         }
620 }
621
622 void
623 stop(char *s)
624 {
625         print_pgm();
626         emit_line(s);
627         longjmp(jmpbuf, 1);
628 }
629
630 void
631 expected(char *s)
632 {
633         if (token == 0)
634             sprintf(errbuf, "Expected %s before end of program", s);
635         else if (token == ';')
636             sprintf(errbuf, "Expected %s before end of statement", s);
637         else
638             sprintf(errbuf, "Expected %s instead of \"%s\"", s, strbuf);
639         stop(errbuf);
640 }
```

9.3 data_step.c

```
1  #include "defs.h"
2
3  static char lbuf[100]; // level name buffer
4
5  int nbytecode;
6  unsigned char bytecode[10000];
7
8  #define TOS 100
9  double stack[TOS];
10 int tos;
11
12 // dd is the head of the linked list of datsets
13
14 static struct dataset *dd;
15
```

```
16 #define BUFLEN 10001
17 static char *inb, buf[BUFLEN];
18 static char filename[1000], delim[100];
19 static int ctrl;
20 static int firstobs;
21 FILE *infile;
22
23 enum {
24       SCAN_NUM,
25       SCAN_STR,
26       NEG,
27       ADD,
28       SUB,
29       MUL,
30       DIV,
31       POW,
32       LOG,
33       LIT,
34       LOAD,
35       STORE,
36       GETBUF,
37       CHKBUF,
38 };
39
40 // The DATALINES statement is the last statement in the DATA step and
41 // immediately precedes the first data line.
42
43 void
44 data_step(void)
45 {
46       ctrl = 0;
47       firstobs = 1;
48       nbytecode = 0;
49       *filename = 0;
50
51       strcpy(delim, "\t ,");
52
53       parse_data_stmt();
54
55       parse_data_body();
56 }
57
58 // parse statements that follow DATA statement
59
60 void
61 parse_data_body()
62 {
63       for (;;) {
64
65             if (token == NAME) {
66
67                   // look ahead for NAME = EXPR
68
69                   while (*inp == ' ' || *inp == '\t')
70                         inp++;
71
72                   if (*inp == '=') {
73                         parse_data_expr();
74                         continue;
75                   }
```

```
76              }
77
78              keyword();
79
80              switch (token) {
81              case 0:
82              case KDATA:
83              case KPROC:
84              case KRUN:
85                      if (*filename == 0)
86                              expected("infile or datalines");
87                      read_file();
88                      return;
89              case KDATALINES:
90                      if (*filename)
91                              stop("Datalines after infile?");
92                      datalines_stmt();
93                      return;
94              case KINPUT:
95                      input_stmt();
96                      break;
97              case KINFILE:
98                      infile_stmt();
99                      break;
100             default:
101                     parse_default();
102                     break;
103             }
104     }
105 }
106
107 void
108 read_file(void)
109 {
110     infile = fopen(filename, "r");
111
112     if (infile == NULL) {
113         sprintf(errbuf, "Cannot open %s", filename);
114         stop(errbuf);
115     }
116
117     get_data();
118
119     fclose(infile);
120     infile = NULL;
121 }
122
123 void
124 input_stmt(void)
125 {
126     int n;
127
128     scan(); // get next token after INPUT
129
130     // scan data specifiers
131
132     for (;;) {
133
134         if (token != NAME)
135             break;
```

```
136
137             if (dd->nvar == MAXVAR)
138                     stop("Too many variables");
139
140             n = dd->nvar++;
141
142             dd->spec[n].name = strdup(strbuf);
143
144             scan();
145
146             if (token == '$') {
147                     dd->spec[n].max_levels = 10;
148                     dd->spec[n].num_levels = 0;
149                     dd->spec[n].ltab = (char **) xmalloc(10 * sizeof (char *));
150                     emit(SCAN_STR);
151                     emit(n);
152                     scan();
153             } else {
154                     dd->spec[n].max_levels = 0;
155                     dd->spec[n].num_levels = 0;
156                     dd->spec[n].ltab = NULL;
157                     emit(SCAN_NUM);
158                     emit(n);
159             }
160     }
161
162     if (token == ATAT) {
163             emit(CHKBUF);
164             scan();
165     } else
166             emit(GETBUF);
167
168     if (token != ';')
169             expected("end of statement");
170
171     scan();
172 }
173
174 void
175 parse_data_stmt()
176 {
177     struct dataset *d;
178
179     d = (struct dataset *) xmalloc(sizeof (struct dataset));
180
181     bzero(d, sizeof (struct dataset));
182
183     d->next = dd;
184     dd = d;
185
186     d->max = 100;
187
188     scan();
189
190     if (token == NAME) {
191             d->name = strdup(strbuf); // dataset NAME
192             scan();
193     }
194
195     if (token != ';')
```

```
196             expected("end of statment");
197
198         scan();
199     }
200
201     void
202     infile_stmt(void)
203     {
204         if (filename[0])
205             stop("Multiple INFILE statements");
206
207         get_next_token();
208
209         switch (token) {
210         case STRING:
211             if (*strbuf == '/')
212                 strcpy(filename, strbuf);
213             else {
214                 strcpy(filename, prefix);
215                 strcat(filename, "/");
216                 strcat(filename, strbuf);
217             }
218             break;
219         case KCARDS:
220         case KDATALINES:
221             break;
222         default:
223             expected("file name");
224         }
225
226         for (;;) {
227
228             scan();
229             keyword();
230
231             switch (token) {
232
233             case ';':
234                 scan();
235                 return;
236
237             case KDLM:
238             case KDELIMITER:
239                 scan();
240                 if (token != '=')
241                     expected("equals sign");
242                 scan();
243                 if (token != STRING)
244                     expected("string");
245                 if (strlen(strbuf) >= sizeof delim)
246                     stop("string too long");
247                 strcpy(delim, strbuf);
248                 break;
249
250             case KFIRSTOBS:
251                 scan();
252                 if (token != '=')
253                     expected("equals sign");
254                 scan();
255                 if (token != NUMBER)
```

```
256                     expected("number");
257                 if (sscanf(strbuf, "%d", &firstobs) != 1)
258                     stop("Error in number syntax");
259                 break;
260
261             default:
262                 expected("end of statement");
263             }
264         }
265 }
266
267 void
268 datalines_stmt(void)
269 {
270         // don't allow garbage after datalines token
271
272         while (isspace(*inp) && *inp != '\n' && *inp != '\r')
273             inp++;
274
275         if (*inp == ';') {
276             inp++;
277             while (isspace(*inp) && *inp != '\n' && *inp != '\r')
278                 inp++;
279         }
280
281         if (*inp != '\n' && *inp != '\r') {
282             scan();
283             expected("datalines data");
284         }
285
286         get_data();
287
288         scan(); // get next token (current token is still DATALINES)
289 }
290
291 void
292 get_data(void)
293 {
294         alloc_data_vectors();
295         getbuf();
296         while (inb) {
297             check_data_vectors();
298             get_data1();
299             dd->nobs++;
300         }
301 }
302
303 // get one row of observations
304
305 void
306 get_data1(void)
307 {
308         int i, j, n, w, x;
309
310         double t;
311
312         // observation number
313
314         n = dd->nobs;
315
```

```
316         // process bytecodes
317
318         tos = 0;
319
320         for (i = 0; i < nbytecode; i++) {
321             switch (bytecode[i]) {
322
323             case SCAN_NUM:
324                 x = bytecode[++i];
325                 w = get_digits();
326                 t = get_number();
327                 if (w > dd->spec[x].w)
328                     dd->spec[x].w = w;
329                 dd->spec[x].v[n] = t;
330                 break;
331
332             case SCAN_STR:
333                 x = bytecode[++i];
334                 get_string();
335                 catvar(x, n);
336                 break;
337
338             case LOAD:
339                 if (tos == TOS)
340                     stop("Stack overflow");
341                 x = bytecode[++i];
342                 stack[tos++] = dd->spec[x].v[n];
343                 break;
344
345             case STORE:
346                 if (tos == 0)
347                     stop("Stack underrun");
348                 x = bytecode[++i];
349                 t = stack[--tos];
350                 dd->spec[x].w = 4; // default precision
351                 dd->spec[x].v[n] = t;
352                 break;
353
354             case LIT:
355                 if (tos == TOS)
356                     stop("Stack overflow");
357                 for (j = 0; j < sizeof (double); j++)
358                     ((char *) &t)[j] = bytecode[++i];
359                 stack[tos++] = t;
360                 break;
361
362             case NEG:
363                 if (tos == 0)
364                     stop("Stack underrun");
365                 stack[tos - 1] = -stack[tos - 1];
366                 break;
367
368             case ADD:
369                 if (tos < 2)
370                     stop("Stack underrun");
371                 t = stack[--tos];
372                 stack[tos - 1] += t;
373                 break;
374
375             case SUB:
```

```
376                 if (tos < 2)
377                     stop("Stack underrun");
378                 t = stack[--tos];
379                 stack[tos - 1] -= t;
380                 break;
381
382             case MUL:
383                 if (tos < 2)
384                     stop("Stack underrun");
385                 t = stack[--tos];
386                 stack[tos - 1] *= t;
387                 break;
388
389             case DIV:
390                 if (tos < 2)
391                     stop("Stack underrun");
392                 t = stack[--tos];
393                 stack[tos - 1] /= t;
394                 break;
395
396             case POW:
397                 if (tos < 2)
398                     stop("Stack underrun");
399                 t = stack[--tos];
400                 stack[tos - 1] = pow(stack[tos - 1], t);
401                 break;
402
403             case LOG:
404                 if (tos < 1)
405                     stop("Stack underrun");
406                 stack[tos - 1] = log(stack[tos - 1]);
407                 break;
408
409             case GETBUF:
410                 getbuf();
411                 break;
412
413             case CHKBUF:
414                 chkbuf();
415                 break;
416
417             default:
418                 stop("Bytecode error");
419             }
420         }
421 }
422
423 void
424 alloc_data_vectors(void)
425 {
426     int i;
427     for (i = 0; i < dd->nvar; i++)
428         if (dd->spec[i].v == NULL)
429             dd->spec[i].v = (double *) xmalloc(dd->max * sizeof (double));
430 }
431
432 // increase length of data vectors if necessary
433
434 void
435 check_data_vectors(void)
```

```
436 {
437     int i;
438     if (dd->nobs < dd->max)
439         return;
440     if (dd->max < 10000)
441         dd->max *= 10;
442     else
443         dd->max += 10000;
444     for (i = 0; i < dd->nvar; i++)
445         dd->spec[i].v = xrealloc(dd->spec[i].v, dd->max * sizeof (double));
446 }
447
448 void
449 getbuf()
450 {
451     do {
452         if (infile)
453             for (;;) {
454                 inb = fgets(buf, sizeof buf, infile);
455                 if (inb == NULL || firstobs < 2)
456                     break;
457                 firstobs--;
458             }
459         else
460             inb = get_dataline(buf, sizeof buf);
461
462         if (inb == NULL)
463             return;
464
465         // skip spaces
466
467         while (isspace(*inb))
468             inb++;
469
470     } while (*inb == 0); // skip blank lines
471 }
472
473 void
474 chkbuf()
475 {
476     // check for end of input
477
478     if (inb == NULL)
479         return;
480
481     // skip spaces
482
483     while (isspace(*inb))
484         inb++;
485
486     if (*inb == 0)
487         getbuf();
488 }
489
490 // get number of decimal digits
491
492 int
493 get_digits(void)
494 {
495     int n = 0;
```

```
496         char *s = inb;
497
498         // scan to decimal point
499
500         while (strchr(delim, *s) == NULL && *s != '.')
501             s++;
502
503         if (*s != '.')
504             return 0;
505
506         while (isdigit(*++s) && n < 8)
507             n++;
508
509         return n;
510 }
511
512 // get a number from the input buffer
513
514 double
515 get_number(void)
516 {
517         double d = NAN;
518
519         // check end of input
520
521         if (inb == NULL)
522             return d;
523
524         // scan leading spaces
525
526         while (isspace(*inb))
527             inb++;
528
529         if (*inb == 0)
530             return d;
531
532         if (strchr(delim, *inb)) {
533             inb++;
534             return d;
535         }
536
537         // scan number
538
539         sscanf(inb, "%lg", &d);
540
541         // scan delimiter
542
543         while (strchr(delim, *inb) == NULL)
544             inb++;
545
546         if (*inb)
547             inb++;
548
549         return d;
550 }
551
552 // get a string from the input buffer
553
554 void
555 get_string()
```

```
556 {
557     int i, n;
558     char *s, *t;
559
560     lbuf[0] = 0;
561
562     // check end of input
563
564     if (inb == NULL)
565         return;
566
567     // scan leading spaces
568
569     while (isspace(*inb))
570         inb++;
571
572     if (*inb == 0)
573         return;
574
575     if (strchr(delim, *inb)) {
576         inb++;
577         return;
578     }
579
580     // start of string
581
582     s = inb;
583
584     // find end of string
585
586     while (strchr(delim, *inb) == NULL)
587         inb++;
588
589     // skip trailing spaces
590
591     t = inb;
592     while (isspace(t[-1]))
593         t--;
594
595     n = (int) (t - s);
596
597     if (n + 1 > sizeof lbuf)
598         stop("In the observed data, a level name is too long");
599
600     // missing data char?
601
602     if (n == 1 && *s == '.')
603         return;
604
605     // copy the string into lbuf
606
607     for (i = 0; i < n; i++)
608         lbuf[i] = s[i];
609
610     lbuf[n] = 0;
611
612     // scan delimiter
613
614     if (*inb)
615         inb++;
```

```
616 }
617
618 void
619 dump_data(void)
620 {
621     int i, j;
622     double d;
623
624     printf("dump_data: %s nvar=%d\n", dd->name, dd->nvar);
625
626     for (i = 0; i < dd->nvar; i++)
627         printf(" %s", dd->spec[i].name);
628
629     printf("\n");
630
631     for (i = 0; i < dd->nobs; i++) {
632         for (j = 0; j < dd->nvar; j++) {
633             d = dd->spec[j].v[i];
634             if (dd->spec[j].ltab)
635                 printf(" %s", dd->spec[j].ltab[(int) d]);
636             else
637                 printf(" %g", d);
638         }
639         printf("\n");
640     }
641 }
642
643 void
644 free_datasets()
645 {
646     int i;
647     struct dataset *t;
648
649     while (dd) {
650         t = dd;
651         dd = dd->next;
652         if (t->name)
653             free(t->name);
654         for (i = 0; i < MAXVAR; i++) {
655             if (t->spec[i].name)
656                 free(t->spec[i].name);
657             if (t->spec[i].v)
658                 free(t->spec[i].v);
659             if (t->spec[i].ltab)
660                 free(t->spec[i].ltab);
661         }
662         free(t);
663     }
664 }
665
666 void
667 select_dataset(char *s)
668 {
669     dataset = dd;
670     if (s == NULL)
671         return;
672     while (dataset) {
673         if (dataset->name && strcmp(dataset->name, s) == 0)
674             return;
675         dataset = dataset->next;
```

```
676         }
677         sprintf(errbuf, "Dataset %s?", s);
678         stop(errbuf);
679 }
680
681 // NAME = EXPR
682
683 void
684 parse_data_expr()
685 {
686         if (dd->nvar == MAXVAR)
687                 stop("Too many variables");
688         dd->spec[dd->nvar].name = strdup(strbuf);
689         scan(); // skip over name
690         scan(); // skip over equals sign
691         parse_expr();
692         if (token != ';')
693                 stop("';' expected");
694         scan();
695         emit(STORE);
696         emit(dd->nvar);
697         dd->nvar++;
698 }
699
700 void
701 parse_expr()
702 {
703         switch (token) {
704         case '+':
705                 scan();
706                 parse_term();
707                 break;
708         case '-':
709                 scan();
710                 parse_term();
711                 emit(NEG);
712                 break;
713         default:
714                 parse_term();
715                 break;
716         }
717
718         for (;;) {
719                 switch (token) {
720                 case '+':
721                         scan();
722                         parse_term();
723                         emit(ADD);
724                         break;
725                 case '-':
726                         scan();
727                         parse_term();
728                         emit(SUB);
729                         break;
730                 default:
731                         return;
732                 }
733         }
734 }
735
```

```
736 void
737 parse_term()
738 {
739         parse_factor();
740
741         for (;;) {
742                 switch (token) {
743                 case '*':
744                         scan();
745                         parse_factor();
746                         emit(MUL);
747                         break;
748                 case '/':
749                         scan();
750                         parse_factor();
751                         emit(DIV);
752                         break;
753                 default:
754                         return;
755                 }
756         }
757 }
758
759 void
760 parse_factor()
761 {
762         switch (token) {
763         case NAME:
764                 if (strcmp(strbuf, "LOG") == 0)
765                         parse_log();
766                 else {
767                         emit_variable();
768                         scan();
769                 }
770                 break;
771         case NUMBER:
772                 emit_number();
773                 scan();
774                 break;
775         case '(':
776                 scan();
777                 parse_expr();
778                 if (token != ')')
779                         stop("Syntax error");
780                 scan();
781                 break;
782         default:
783                 stop("Syntax error");
784         }
785
786         if (token == STARSTAR) {
787                 scan();
788                 switch (token) {
789                 case '+':
790                         scan();
791                         parse_factor();
792                         break;
793                 case '-':
794                         scan();
795                         parse_factor();
```

```
796                     emit(NEG);
797                     break;
798             default:
799                     parse_factor();
800                     break;
801             }
802             emit(POW);
803     }
804 }
805
806 void
807 parse_log()
808 {
809     // look ahead
810
811     while (isspace(*inp) && *inp != '\n' && *inp != '\r')
812             inp++;
813
814     if (*inp != '(') {
815             emit_variable();
816             scan();
817             return;
818     }
819
820     scan();
821     scan();
822
823     parse_expr();
824
825     if (token != ')')
826             expected(")");
827
828     scan();
829
830     emit(LOG);
831 }
832
833 void
834 emit(int c)
835 {
836     if (nbytecode == sizeof bytecode)
837             stop("Bytecode buffer overrun");
838     bytecode[nbytecode++] = c;
839 }
840
841 // Variable name is in strbuf
842
843 void
844 emit_variable()
845 {
846     int i;
847     for (i = 0; i < dd->nvar; i++)
848             if (strcmp(dd->spec[i].name, strbuf) == 0)
849                     break;
850     if (i == dd->nvar)
851             stop("Variable not found");
852     if (dd->spec[i].ltab)
853             stop("An arithemetic expression cannot use a category variable");
854     emit(LOAD);
855     emit(i);
```

```
856 }
857
858 void
859 emit_number()
860 {
861       int i;
862       char *p = (char *) &token_num;
863       emit(LIT);
864       for (i = 0; i < sizeof (double); i++)
865             emit(p[i]);
866 }
867
868 // To here with category level from input data stream
869 //
870 //    x     Variable index
871 //
872 //    obs   Observation number
873
874 void
875 catvar(int x, int obs)
876 {
877       int i, m, n;
878
879       // missing data?
880
881       if (lbuf[0] == 0) {
882             dd->spec[x].v[obs] = NAN;
883             return;
884       }
885
886       m = dd->spec[x].max_levels;
887       n = dd->spec[x].num_levels;
888
889       // search for the level name
890
891       for (i = 0; i < n; i++)
892             if (strcmp(dd->spec[x].ltab[i], lbuf) == 0)
893                   break;
894
895       // check for new level name
896
897       if (i == n) {
898
899             // get space for more levels if necessary
900
901             if (m == n) {
902                   m += 100;
903                   dd->spec[x].max_levels = m;
904                   dd->spec[x].ltab = xrealloc(dd->spec[x].ltab, m * sizeof (char *));
905             }
906
907             dd->spec[x].ltab[i] = strdup(lbuf);
908             dd->spec[x].num_levels++;
909       }
910
911       // save the level number
912
913       dd->spec[x].v[obs] = i;
914 }
```

9.4 proc_anova.c

```
1  #include "defs.h"
2
3  //   B     Regression coefficients
4  //
5  //   css   Corrected sum of squares
6  //
7  //   df    Degrees of freedom
8  //
9  //   dfe   Degrees of freedom error
10 //
11 //   G     Inverse of X'X
12 //
13 //   gstate     Current state of G
14 //
15 //   miss Missing vector
16 //
17 //   ncol Number of columns in design matrix
18 //
19 //   nobs Number of rows in design matrix
20 //
21 //   npar Number of fit parameters
22 //
23 //   nx    Number of explanatory variables
24 //
25 //   ss    Sequential sum of squares regression
26 //
27 //   ssr  Sum of squares regression
28 //
29 //   sse  Sum of squares error
30 //
31 //   T     Temporary matrix
32 //
33 //   X     Design matrix
34 //
35 //   xtab List of explanatory variables
36 //
37 //   Y     Response vector
38 //
39 //   Yhat Predicted response X * B
40 //
41 //   ybar Mean of response variable Y
42
43 static double *B;
44 static int df[MAXVAR];
45 static double *GG;
46 static int kk[MAXVAR];
47 int *miss;
48 static int ncol;
49 static int nobs;
50 static int npar;
51 static int nx;
52 static double *TT;
53 static double *XX;
54 static int xx[MAXVAR];
55 static int xtab[MAXVAR];
56 static double *Y;
57 static double ybar;
58 static int yvar;
```

```
59  static int gstate;
60  static double *Yhat;
61  static double css;
62  static double ssr;
63  static double sse;
64  static double ss[MAXVAR];
65  static double msr;
66  static double mse;
67  static double rootmse;
68  static double fval;
69  static double pval;
70  static double rsquare;
71  static double adjrsq;
72  static double cv;
73  static char buf[1000];
74  static int nmiss;
75  #define MAXLVL 100
76  static double mean[MAXLVL];
77  static double variance[MAXLVL];
78  static int count[MAXLVL];
79  #define MAXITEM 100
80  static int item[MAXITEM];
81  static double dfe;
82
83  #define G(i, j) (GG + (i) * ncol)[j]
84  #define T(i, j) (TT + (i) * ncol)[j]
85  #define X(i, j) (XX + (i) * ncol)[j]
86
87  static void parse_proc_anova_stmt();
88  static void parse_proc_anova_body();
89  static void parse_model_stmt();
90  static void parse_model_options();
91  static void parse_explanatory_variable();
92  static void regression();
93  static int get_var_index();
94  static void add_interaction(int);
95  static void create_interaction_level_names(int);
96  static void add_factorial(int);
97  static void add_nested(int);
98  static void prelim();
99  static int a_compute_G();
100 static void fit();
101 static void fit1(int, int);
102 static void compute_B();
103 static void compute_Yhat();
104 static void compute_ss();
105 static void model();
106 static void compute_means();
107 static void print_means();
108 static void parse_means_stmt();
109 static void parse_means_item();
110 static void print_lsd(int);
111 static void print_ttest(int);
112
113 void
114 proc_anova()
115 {
116     nx = 0;
117
118     parse_proc_anova_stmt();
```

```
119
120         if (dataset == NULL)
121             stop("No data set");
122
123         parse_proc_anova_body();
124     }
125
126     static void
127     parse_proc_anova_stmt()
128     {
129         for (;;) {
130             scan();
131             keyword();
132             switch (token) {
133             case ';':
134                 scan(); // eat the semicolon
135                 return;
136             case KDATA:
137                 parse_data_option();
138                 break;
139             default:
140                 expected("end of statement or data=name");
141             }
142         }
143     }
144
145     static void
146     parse_proc_anova_body()
147     {
148         for (;;) {
149             keyword();
150             switch (token) {
151             case 0:
152             case KDATA:
153             case KPROC:
154             case KRUN:
155                 return;
156             case KCLASS:
157                 parse_class_stmt();
158                 break;
159             case KMEANS:
160                 parse_means_stmt();
161                 break;
162             case KMODEL:
163                 parse_model_stmt();
164                 model();
165                 break;
166             default:
167                 parse_default();
168                 break;
169             }
170         }
171     }
172
173     static void
174     parse_means_stmt()
175     {
176         int i, lsd, n, ttest, x;
177         char *s;
178
```

```
179         alpha = 0.05;
180         ttest = 0;
181         lsd = 0;
182         n = 0;
183
184         scan();
185
186         for (;;) {
187
188               if (token == ';' || token == '/')
189                     break;
190
191               parse_means_item();
192
193               for (i = 0; i < dataset->nvar; i++) {
194                     s = dataset->spec[i].name;
195                     if (strcmp(s, buf) == 0)
196                           break;
197               }
198
199               if (i == dataset->nvar)
200                     stop("Not in data set");
201
202               if (dataset->spec[i].ltab == NULL)
203                     stop("Not categorical");
204
205               if (n == MAXITEM)
206                     stop("Buffer overrun");
207
208               item[n++] = i;
209         }
210
211         // parse options
212
213         if (token == '/') {
214               scan();
215               while (token != ';') {
216                     keyword();
217                     switch (token) {
218                     case KALPHA:
219                           parse_alpha_option();
220                           break;
221                     case KT:
222                     case KLSD:
223                           lsd = 1;
224                           break;
225                     case KTTEST:
226                           ttest = 1;
227                           break;
228                     default:
229                           expected("statement option");
230                           break;
231                     }
232                     scan();
233               }
234         }
235
236         scan(); // eat end of line
237
238         for (i = 0; i < n; i++) {
```

```
239             x = item[i];
240             compute_means(x);
241             print_means(x);
242             if (lsd)
243                     print_lsd(x);
244             if (ttest)
245                     print_ttest(x);
246     }
247 }
248
249 // An item is a variable name or an interaction
250
251 // Returns token in buf[]
252
253 static void
254 parse_means_item()
255 {
256     if (token != NAME)
257         expected("variable name");
258
259     if (strlen(strbuf) + 1 > sizeof buf)
260         stop("Buffer overrun");
261
262     strcpy(buf, strbuf);
263
264     scan();
265
266     while (token == '*') {
267         scan();
268         if (token != NAME)
269             expected("variable name");
270         if (strlen(buf) + strlen(strbuf) + 2 > sizeof buf)
271             stop("Buffer overrun");
272         strcat(buf, "*");
273         strcat(buf, strbuf);
274         scan();
275     }
276 }
277
278 static void
279 parse_model_stmt()
280 {
281     int i;
282
283     nx = 0;
284
285     // parse dependent variable
286
287     scan();
288
289     if (token != NAME)
290         expected("variable name");
291
292     for (i = 0; i < dataset->nvar; i++)
293         if (strcmp(dataset->spec[i].name, strbuf) == 0)
294             break;
295
296     if (i == dataset->nvar)
297         expected("variable in data set");
298
```

```
299         if (dataset->spec[i].ltab)
300             expected("numeric variable");
301 
302         yvar = i;
303 
304         scan();
305 
306         if (token != '=')
307             expected("equals sign");
308 
309         scan();
310 
311         // parse explanatory variables
312 
313         for (;;) {
314 
315             if (token == ';' || token == '/')
316                 break;
317 
318             if (token != NAME)
319                 expected("variable name");
320 
321             parse_explanatory_variable();
322         }
323 
324         if (token == '/')
325             parse_model_options();
326 
327         scan(); // eat the semicolon
328 }
329 
330 static void
331 parse_model_options()
332 {
333         for (;;) {
334 
335             scan();
336 
337             keyword();
338 
339             switch (token) {
340 
341             case ';':
342                 return;
343 
344             default:
345                 stop("';' or MODEL option expected");
346             }
347         }
348 }
349 
350 // Explanatory variable forms
351 //
352 // 1.       name
353 //
354 // 2.       name * name ...
355 //
356 // 3.       name | name ...
357 //
358 // 4.       name(name)
```

```
359 //
360 // 5.      name(name name ...)
361
362 static void
363 parse_explanatory_variable()
364 {
365     int n = 1;
366
367     xx[0] = get_var_index();
368
369     scan();
370
371     switch (token) {
372
373     case '*': // name * name ...
374
375         do {
376             scan();
377             if (token != NAME)
378                 expected("variable name");
379             if (n == MAXVAR)
380                 stop("Too many interaction terms");
381             xx[n++] = get_var_index();
382             scan();
383         } while (token == '*');
384         add_interaction(n);
385         break;
386
387     case '|': // name | name ...
388
389         do {
390             scan();
391             if (token != NAME)
392                 expected("variable name");
393             if (n == MAXVAR)
394                 stop("Too many interaction terms");
395             xx[n++] = get_var_index();
396             scan();
397         } while (token == '|');
398         add_factorial(n);
399         break;
400
401     case '(': // name(name name ...)
402
403         scan();
404         while (token == NAME) {
405             if (n == MAXVAR)
406                 stop("Too many interaction terms");
407             xx[n++] = get_var_index();
408             scan();
409         };
410         if (token != ')')
411             expected("right parenthesis ')'");
412         scan();
413         add_nested(n);
414         break;
415
416     default:
417         if (nx == MAXVAR)
418             stop("Too many explanatory variables");
```

```
419             xtab[nx++] = xx[0];
420             break;
421         }
422 }
423
424 static int
425 get_var_index()
426 {
427         int i, n;
428         n = dataset->nvar;
429         for (i = 0; i < n; i++)
430             if (strcmp(dataset->spec[i].name, strbuf) == 0)
431                 break;
432         if (i == n) {
433             sprintf(buf, "The variable name \"%s\" is not in the data set", strbuf);
434             stop(buf);
435         }
436         if (dataset->spec[i].ltab == NULL) {
437             sprintf(buf, "The variable \"%s\" is numeric, please change to categorical in the data ste
438             stop(buf);
439         }
440         return i;
441 }
442
443 // There are n interaction names in xx[]
444
445 static void
446 add_interaction(int n)
447 {
448         int i, j, k, len, m, x;
449         char *name, *s;
450         double d;
451
452         len = n; // accommodate n - 1 asterisks and 1 end of line
453
454         for (i = 0; i < n; i++) {
455             x = xx[i];
456             s = dataset->spec[x].name;
457             len += strlen(s);
458         }
459
460         name = (char *) xmalloc(len);
461
462         name[0] = 0;
463
464         for (i = 0; i < n; i++) {
465             if (i > 0)
466                 strcat(name, "*");
467             x = xx[i];
468             s = dataset->spec[x].name;
469             strcat(name, s);
470         }
471
472         // is it already in the data set?
473
474         for (m = 0; m < dataset->nvar; m++)
475             if (strcmp(name, dataset->spec[m].name) == 0)
476                 break;
477
478         if (m < dataset->nvar)
```

```
479             free(name);
480         else {
481             if (dataset->nvar == MAXVAR)
482                 stop("Too many variable names");
483 
484             m = dataset->nvar++;
485 
486             dataset->spec[m].name = name;
487 
488             // number of levels
489 
490             k = 1;
491             for (i = 0; i < n; i++) {
492                 x = xx[i];
493                 k *= dataset->spec[x].num_levels;
494             }
495 
496             dataset->spec[m].ltab = (char **) xmalloc(k * sizeof (char *));
497             dataset->spec[m].max_levels = k;
498             dataset->spec[m].num_levels = k;
499             dataset->spec[m].v = (double *) xmalloc(dataset->max * sizeof (double));
500 
501             create_interaction_level_names(n);
502 
503             // data values
504 
505             for (i = 0; i < dataset->nobs; i++) {
506                 d = 0;
507                 for (j = 0; j < n; j++) {
508                     x = xx[j];
509                     d = d * dataset->spec[x].num_levels + dataset->spec[x].v[i];
510                 }
511                 dataset->spec[m].v[i] = d;
512             }
513         }
514 
515         // is it already an explanatory variable?
516 
517         for (i = 0; i < nx; i++)
518             if (xtab[i] == m)
519                 return;
520 
521         if (nx == MAXVAR)
522             stop("Too many explanatory variables");
523 
524         xtab[nx++] = m;
525     }
526 
527     // Create interaction level names for product of n terms
528 
529     static void
530     create_interaction_level_names(int n)
531     {
532         int i, j, k, len, m, x;
533         char *name, *s;
534 
535         m = dataset->nvar - 1; // index of new interaction column
536 
537         for (i = 0; i < n; i++)
538             kk[i] = 0;
```

```
539
540         for (i = 0; i < dataset->spec[m].num_levels; i++) {
541
542             len = n;
543
544             for (j = 0; j < n; j++) {
545                 x = xx[j];
546                 k = kk[j];
547                 s = dataset->spec[x].ltab[k];
548                 len += strlen(s);
549             }
550
551             name = (char *) xmalloc(len);
552
553             name[0] = 0;
554
555             for (j = 0; j < n; j++) {
556                 if (j > 0)
557                     strcat(name, "*");
558                 x = xx[j];
559                 k = kk[j];
560                 s = dataset->spec[x].ltab[k];
561                 strcat(name, s);
562             }
563
564             dataset->spec[m].ltab[i] = name;
565
566             // increment indices
567
568             for (j = n - 1; j >= 0; j--) {
569                 x = xx[j];
570                 if (++kk[j] < dataset->spec[x].num_levels)
571                     break;
572                 kk[j] = 0;
573             }
574         }
575     }
576
577     static void
578     add_factorial(int n)
579     {
580     }
581
582     static void
583     add_nested(int n)
584     {
585     }
586
587     static void
588     model()
589     {
590         regression();
591
592         print_title();
593
594         if (nmiss) {
595             sprintf(buf, "%d observations deleted due to missing data", nmiss);
596             emit_line(buf);
597         }
598
```

```
599         emit_line_center("Analysis of Variance");
600         emit_line("");
601
602         print_anova_table_part1();
603
604         print_anova_table_part2();
605
606         print_anova_table_part3();
607     }
608
609     static void
610     regression()
611     {
612         prelim();
613
614         fit();
615
616         dfe = nobs - npar;
617
618         mse = sse / dfe;
619
620         rootmse = sqrt(mse);
621
622         msr = ssr / (npar - 1);
623
624         fval = msr / mse;
625
626         pval = 1 - fdist(fval, npar - 1, nobs - npar);
627
628         rsquare = 1 - sse / css;
629
630         adjrsq = 1 - (double) (nobs - 1) / (nobs - npar) * sse / css;
631
632         cv = 100 * rootmse / ybar;
633     }
634
635     static void
636     prelim()
637     {
638         int i, j, k, x;
639
640         if (miss)
641             free(miss);
642
643         miss = (int *) xmalloc(dataset->nobs * sizeof (int));
644
645         nmiss = 0;
646
647         for (i = 0; i < dataset->nobs; i++) {
648             miss[i] = 0;
649             if (isnan(dataset->spec[yvar].v[i])) {
650                 miss[i] = 1;
651                 nmiss++;
652                 continue;
653             }
654             for (j = 0; j < nx; j++) {
655                 x = xtab[j];
656                 if (isnan(dataset->spec[x].v[i])) {
657                     miss[i] = 1;
658                     nmiss++;
```

```
659                         break;
660                     }
661                 }
662         }
663
664         nobs = dataset->nobs - nmiss;
665
666         // add up all levels
667
668         ncol = 1; // for intercept
669
670         for (i = 0; i < nx; i++) {
671                 x = xtab[i];
672                 ncol += dataset->spec[x].num_levels;
673         }
674
675         if (B) {
676                 free(B);
677                 free(GG);
678                 free(TT);
679                 free(XX);
680                 free(Y);
681                 free(Yhat);
682         }
683
684         B = (double *) xmalloc(ncol * sizeof (double));
685         GG = (double *) xmalloc(ncol * ncol * sizeof (double));
686         TT = (double *) xmalloc(ncol * ncol * sizeof (double));
687         XX = (double *) xmalloc(nobs * ncol * sizeof (double));
688         Y = (double *) xmalloc(nobs * sizeof (double));
689         Yhat = (double *) xmalloc(nobs * sizeof (double));
690
691         // fill Y with non-missing values
692
693         k = 0;
694
695         for (i = 0; i < dataset->nobs; i++) {
696                 if (miss[i])
697                         continue;
698                 Y[k++] = dataset->spec[yvar].v[i];
699         }
700
701         ybar = 0;
702
703         for (i = 0; i < nobs; i++)
704                 ybar += Y[i];
705
706         ybar /= nobs;
707
708         // corrected sum of squares
709
710         css = 0;
711
712         for (i = 0; i < nobs; i++)
713                 css += (Y[i] - ybar) * (Y[i] - ybar);
714 }
715
716 static void
717 fit()
718 {
```

```
719         int i, j, n, x;
720
721         // put in intercept
722
723         for (i = 0; i < nobs; i++)
724             X(i, 0) = 1;
725
726         npar = 1;
727
728         // fit explanatory variables
729
730         for (i = 0; i < nx; i++) {
731             x = xtab[i];
732             n = dataset->spec[x].num_levels;
733             for (j = 0; j < n; j++)
734                 fit1(x, j); // fit next column
735             df[i] = npar;
736             if (gstate == -1)
737                 a_compute_G();
738             compute_B();
739             compute_Yhat();
740             compute_ss();
741             ss[i] = ssr;
742         }
743
744         // differentials
745
746         for (i = nx - 1; i > 0; i--) {
747             df[i] -= df[i - 1];
748             ss[i] -= ss[i - 1];
749         }
750
751         df[0]--; // subtract 1 for intercept
752     }
753
754     static void
755     fit1(int x, int level)
756     {
757         int i, k = 0;
758
759         for (i = 0; i < dataset->nobs; i++) {
760             if (miss[i])
761                 continue;
762             X(k++, npar) = (dataset->spec[x].v[i] == level) ? 1 : 0;
763         }
764
765         npar++;
766
767         gstate = a_compute_G();
768
769         if (gstate == -1)
770             npar--; // X'X is singular hence remove column
771     }
772
773     static int
774     a_compute_G()
775     {
776         int d, i, j, k;
777         double m, max, min, t;
778
```

```
779         min = 0;
780         max = 0;
781
782         // G = I
783
784         for (i = 0; i < npar; i++)
785               for (j = 0; j < npar; j++)
786                     if (i == j)
787                           G(i, j) = 1;
788                     else
789                           G(i, j) = 0;
790
791         // T = X'X
792
793         for (i = 0; i < npar; i++)
794               for (j = 0; j < npar; j++) {
795                     t = 0;
796                     for (k = 0; k < nobs; k++)
797                           t += X(k, i) * X(k, j);
798                     T(i, j) = t;
799               }
800
801         // G = inv T
802
803         for (d = 0; d < npar; d++) {
804
805               // find the best pivot row
806
807               k = d;
808               for (i = d + 1; i < npar; i++)
809                     if (fabs(T(i, d)) > fabs(T(k, d)))
810                           k = i;
811
812               // exchange rows if necessary
813
814               if (k != d) {
815                     for (j = d; j < npar; j++) { // skip zeroes, start at d
816                           t = T(d, j);
817                           T(d, j) = T(k, j);
818                           T(k, j) = t;
819                     }
820                     for (j = 0; j < npar; j++) {
821                           t = G(d, j);
822                           G(d, j) = G(k, j);
823                           G(k, j) = t;
824                     }
825               }
826
827               // multiply the pivot row by 1 / pivot
828
829               m = T(d, d);
830
831               if (m == 0)
832                     return -1;
833
834               if (fabs(m) > max)
835                     max = fabs(m);
836
837               m = 1 / m;
838
```

```
839             if (fabs(m) > min)
840                  min = fabs(m);
841
842             for (j = d; j < npar; j++) // skip zeroes, start at d
843                  T(d, j) *= m;
844             for (j = 0; j < npar; j++)
845                  G(d, j) *= m;
846
847             // clear out column below d
848
849             for (i = d + 1; i < npar; i++) {
850                  m = -T(i, d);
851                  for (j = d; j < npar; j++) // skip zeroes, start at d
852                       T(i, j) += m * T(d, j);
853                  for (j = 0; j < npar; j++)
854                       G(i, j) += m * G(d, j);
855             }
856        }
857
858        // clear out columns above diagonal
859
860        for (d = npar - 1; d > 0; d--)
861             for (i = 0; i < d; i++) {
862                  m = -T(i, d);
863                  for (j = 0; j < npar; j++)
864                       G(i, j) += m * G(d, j);
865             }
866
867        // check ratio of biggest divisor to smallest divisor
868
869        // domain is [1, inf)
870
871        // printf("cond = %g\n", max * min);
872
873        if (max * min < 1e10)
874             return 0;
875        else
876             return -1;
877   }
878
879   // B = G * X^T * Y
880
881   static void
882   compute_B()
883   {
884        int i, j;
885        double t;
886
887        for (i = 0; i < npar; i++) {
888             t = 0;
889             for (j = 0; j < nobs; j++)
890                  t += X(j, i) * Y[j];
891             T(0, i) = t;
892        }
893
894        for (i = 0; i < npar; i++) {
895             t = 0;
896             for (j = 0; j < npar; j++)
897                  t += G(i, j) * T(0, j);
898             B[i] = t;
```

```
899         }
900 }
901
902 // Yhat = X * B
903
904 static void
905 compute_Yhat()
906 {
907         int i, j;
908         double t;
909
910         for (i = 0; i < nobs; i++) {
911                 t = 0;
912                 for (j = 0; j < npar; j++)
913                         t += X(i, j) * B[j];
914                 Yhat[i] = t;
915         }
916 }
917
918 static void
919 compute_ss()
920 {
921         int i;
922
923         ssr = 0;
924         sse = 0;
925
926         for (i = 0; i < nobs; i++) {
927                 ssr += (Yhat[i] - ybar) * (Yhat[i] - ybar);
928                 sse += (Y[i] - Yhat[i]) * (Y[i] - Yhat[i]);
929         }
930 }
931
932 #if 0
933
934 static void
935 print_T()
936 {
937         int i, j;
938         printf("T =\n");
939         for (i = 0; i < npar; i++) {
940                 for (j = 0; j < npar; j++)
941                         printf(" %g", X(i, j));
942                 printf("\n");
943         }
944 }
945
946 static void
947 print_X()
948 {
949         int i, j;
950         printf("X =\n");
951         for (i = 0; i < nobs; i++) {
952                 for (j = 0; j < npar; j++)
953                         printf(" %g", X(i, j));
954                 printf("\n");
955         }
956 }
957
958 #endif
```

```
959
960 #undef A
961 #define A(i, j) (a + 6 * (i))[j]
962
963 void
964 print_anova_table_part1()
965 {
966       char **a;
967
968       a = (char **) xmalloc(4 * 6 * sizeof (char *));
969
970       // 1st row
971
972       A(0, 0) = strdup("Source");
973       A(0, 1) = strdup("DF");
974       A(0, 2) = strdup("Sum of Squares");
975       A(0, 3) = strdup("Mean Square");
976       A(0, 4) = strdup("F Value");
977       A(0, 5) = strdup("Pr > F");
978
979       // 2nd row
980
981       A(1, 0) = strdup("Model");
982
983       sprintf(buf, "%d", npar - 1);
984       A(1, 1) = strdup(buf);
985
986       sprintf(buf, "%0.8f", ssr);
987       A(1, 2) = strdup(buf);
988
989       sprintf(buf, "%0.8f", msr);
990       A(1, 3) = strdup(buf);
991
992       sprintf(buf, "%0.2f", fval);
993       A(1, 4) = strdup(buf);
994
995       sprintf(buf, "%0.4f", pval);
996       A(1, 5) = strdup(buf);
997
998       // 3rd row
999
1000      A(2, 0) = strdup("Error");
1001
1002      sprintf(buf, "%d", nobs - npar);
1003      A(2, 1) = strdup(buf);
1004
1005      sprintf(buf, "%0.8f", sse);
1006      A(2, 2) = strdup(buf);
1007
1008      sprintf(buf, "%0.8f", mse);
1009      A(2, 3) = strdup(buf);
1010
1011      A(2, 4) = strdup("");
1012      A(2, 5) = strdup("");
1013
1014      // 4th row
1015
1016      A(3, 0) = strdup("Total");
1017
1018      sprintf(buf, "%d", nobs - 1);
```

```
1019        A(3, 1) = strdup(buf);
1020
1021        sprintf(buf, "%0.8f", css);
1022        A(3, 2) = strdup(buf);
1023
1024        A(3, 3) = strdup("");
1025        A(3, 4) = strdup("");
1026        A(3, 5) = strdup("");
1027
1028        buf[0] = 1; // left justify
1029        buf[1] = 0;
1030        buf[2] = 0;
1031        buf[3] = 0;
1032        buf[4] = 0;
1033        buf[5] = 0;
1034
1035        print_table_and_free(a, 4, 6, buf);
1036  }
1037
1038  #undef A
1039  #define A(i, j) (a + 4 * (i))[j]
1040
1041  void
1042  print_anova_table_part2()
1043  {
1044        char **a;
1045
1046        a = (char **) xmalloc(8 * sizeof (char *));
1047
1048        A(0, 0) = strdup("R-Square");
1049        A(0, 1) = strdup("Coeff Var");
1050        A(0, 2) = strdup("Root MSE");
1051
1052        sprintf(buf, "%s Mean", dataset->spec[yvar].name);
1053        A(0, 3) = strdup(buf);
1054
1055        sprintf(buf, "%0.6f", rsquare);
1056        A(1, 0) = strdup(buf);
1057
1058        sprintf(buf, "%0.6f", cv);
1059        A(1, 1) = strdup(buf);
1060
1061        sprintf(buf, "%0.6f", rootmse);
1062        A(1, 2) = strdup(buf);
1063
1064        sprintf(buf, "%0.6f", ybar);
1065        A(1, 3) = strdup(buf);
1066
1067        buf[0] = 0; // right justified
1068        buf[1] = 0;
1069        buf[2] = 0;
1070        buf[3] = 0;
1071
1072        print_table_and_free(a, 2, 4, buf);
1073  }
1074
1075  #undef A
1076  #define A(i, j) (a + 6 * (i))[j]
1077
1078  void
```

```
1079 print_anova_table_part3()
1080 {
1081     int i, x;
1082     double msq, fval, pval;
1083     char **a;
1084
1085     a = (char **) xmalloc(6 * (nx + 1) * sizeof (char *));
1086
1087     A(0, 0) = strdup("Source");
1088     A(0, 1) = strdup("DF");
1089     A(0, 2) = strdup("Anova SS");
1090     A(0, 3) = strdup("Mean Square");
1091     A(0, 4) = strdup("F Value");
1092     A(0, 5) = strdup("Pr > F");
1093
1094     for (i = 0; i < nx; i++) {
1095
1096         x = xtab[i];
1097
1098         // Source
1099
1100         A(i + 1, 0) = strdup(dataset->spec[x].name);
1101
1102         // DF
1103
1104         sprintf(buf, "%d", df[i]);
1105         A(i + 1, 1) = strdup(buf);
1106
1107         if (df[i] == 0) {
1108             A(i + 1, 2) = strdup(".");
1109             A(i + 1, 3) = strdup(".");
1110             A(i + 1, 4) = strdup(".");
1111             A(i + 1, 5) = strdup(".");
1112             continue;
1113         }
1114
1115         // Anova SS
1116
1117         sprintf(buf, "%0.8f", ss[i]);
1118         A(i + 1, 2) = strdup(buf);
1119
1120         // Mean Square
1121
1122         msq = ss[i] / df[i];
1123
1124         sprintf(buf, "%0.8f", msq);
1125         A(i + 1, 3) = strdup(buf);
1126
1127         // F Value
1128
1129         fval = msq / mse;
1130
1131         sprintf(buf, "%0.2f", fval);
1132         A(i + 1, 4) = strdup(buf);
1133
1134         // Pr > F
1135
1136         pval = 1 - fdist(fval, df[i], nobs - npar);
1137
1138         sprintf(buf, "%0.4f", pval);
```

```
1139            A(i + 1, 5) = strdup(buf);
1140        }
1141
1142        buf[0] = 1; // left justify
1143        buf[1] = 0;
1144        buf[2] = 0;
1145        buf[3] = 0;
1146        buf[4] = 0;
1147        buf[5] = 0;
1148
1149        print_table_and_free(a, nx + 1, 6, buf);
1150    }
1151
1152    static void
1153    compute_means(int x)
1154    {
1155        int i, level, n;
1156        double t, y;
1157
1158        n = dataset->spec[x].num_levels;
1159
1160        if (n > MAXLVL)
1161            stop("Too many levels");
1162
1163        for (i = 0; i < n; i++) {
1164            mean[i] = 0;
1165            variance[i] = 0;
1166            count[i] = 0;
1167        }
1168
1169        for (i = 0; i < dataset->nobs; i++) {
1170            if (miss[i])
1171                continue;
1172            y = dataset->spec[yvar].v[i];
1173            level = dataset->spec[x].v[i];
1174            count[level]++;
1175            t = mean[level];
1176            mean[level] += (y - t) / count[level];
1177            variance[level] += (y - t) * (y - mean[level]);
1178        }
1179
1180        for (i = 0; i < n; i++)
1181            variance[i] /= (count[i] - 1);
1182    }
1183
1184    #undef A
1185    #define A(i, j) (a + (5 + m) * (i))[j]
1186
1187    static void
1188    print_means(int x)
1189    {
1190        int i, j, m, n;
1191        double q, t;
1192        char **a, *b, *s;
1193
1194        q = qt(1 - alpha / 2, dfe);
1195
1196        emit_line_center("Mean Response");
1197        emit_line("");
1198
```

```
1199        // count the number of interactions
1200
1201        m = 0;
1202        s = dataset->spec[x].name;
1203        while (*s)
1204             if (*s++ == '*')
1205                  m++;
1206
1207        n = dataset->spec[x].num_levels;
1208
1209        a = (char **) xmalloc((5 + m) * (n + 1) * sizeof (char *));
1210
1211        // split up interaction string
1212
1213        s = dataset->spec[x].name;
1214        for (i = 0; i < m + 1; i++) {
1215             b = buf;
1216             while (*s && *s != '*')
1217                  *b++ = *s++;
1218             if (*s)
1219                  s++;
1220             *b = 0;
1221             A(0, i) = strdup(buf);
1222        }
1223
1224        A(0, m + 1) = strdup("N");
1225
1226        s = dataset->spec[yvar].name;
1227        sprintf(buf, "Mean %s", s);
1228        A(0, m + 2) = strdup(buf);
1229
1230        sprintf(buf, "%g%% CI MIN", 100 * (1 - alpha));
1231        A(0, m + 3) = strdup(buf);
1232
1233        sprintf(buf, "%g%% CI MAX", 100 * (1 - alpha));
1234        A(0, m + 4) = strdup(buf);
1235
1236        for (i = 0; i < n; i++) {
1237
1238             // Level (decompose interactions)
1239
1240             s = dataset->spec[x].ltab[i];
1241             for (j = 0; j < m + 1; j++) {
1242                  b = buf;
1243                  while (*s && *s != '*')
1244                       *b++ = *s++;
1245                  if (*s)
1246                       s++;
1247                  *b = 0;
1248                  A(i + 1, j) = strdup(buf);
1249             }
1250
1251             // N
1252
1253             sprintf(buf, "%d", count[i]);
1254             A(i + 1, m + 1) = strdup(buf);
1255
1256             if (count[i] < 1) {
1257                  A(i + 1, m + 2) = strdup(".");
1258                  A(i + 1, m + 3) = strdup(".");
```

```
1259                A(i + 1, m + 4) = strdup(".");
1260                continue;
1261            }
1262
1263            // Mean
1264
1265            sprintf(buf, "%0.6f", mean[i]);
1266            A(i + 1, m + 2) = strdup(buf);
1267
1268            // Confidence Interval
1269
1270            t = q * sqrt(mse / count[i]);
1271
1272            sprintf(buf, "%0.6f", mean[i] - t);
1273            A(i + 1, m + 3) = strdup(buf);
1274
1275            sprintf(buf, "%0.6f", mean[i] + t);
1276            A(i + 1, m + 4) = strdup(buf);
1277        }
1278
1279        for (i = 0; i < m + 1; i++)
1280            buf[i] = 1;
1281        buf[m + 1] = 0;
1282        buf[m + 2] = 0;
1283        buf[m + 3] = 0;
1284        buf[m + 4] = 0;
1285
1286        print_table_and_free(a, n + 1, 5 + m, buf);
1287    }
1288
1289    #undef A
1290    #define A(i, j) (a + ncol * (i))[j]
1291
1292    static void
1293    print_lsd(int x)
1294    {
1295        int i, j, k, n, ncol, nrow;
1296        char **a, *s;
1297        double d, lsd, pval, q, se, tval;
1298
1299        emit_line_center("Least Significant Difference Test");
1300        emit_line("");
1301
1302        q = qt(1 - alpha / 2, dfe); // degrees of freedom error
1303
1304        n = dataset->spec[x].num_levels;
1305
1306        nrow = 1 + n * (n - 1);
1307
1308        ncol = 7;
1309
1310        a = (char **) xmalloc(nrow * ncol * sizeof (char *));
1311
1312        s = dataset->spec[x].name;
1313        A(0, 0) = strdup(s);
1314        A(0, 1) = strdup(s);
1315
1316        s = dataset->spec[yvar].name;
1317        sprintf(buf, "Delta %s", s);
1318        A(0, 2) = strdup(buf);
```

```
1319
1320        sprintf(buf, "%g%% CI MIN", 100 * (1 - alpha));
1321        A(0, 3) = strdup(buf);
1322
1323        sprintf(buf, "%g%% CI MAX", 100 * (1 - alpha));
1324        A(0, 4) = strdup(buf);
1325
1326        A(0, 5) = strdup("t Value");
1327        A(0, 6) = strdup("Pr > |t|  ");
1328
1329        k = 0;
1330
1331        for (i = 0; i < n; i++) {
1332             for (j = 0; j < n; j++) {
1333
1334                  if (i == j)
1335                       continue;
1336
1337                  k++;
1338
1339                  // Level
1340
1341                  s = dataset->spec[x].ltab[i];
1342                  A(k, 0) = strdup(s);
1343
1344                  // Level
1345
1346                  s = dataset->spec[x].ltab[j];
1347                  A(k, 1) = strdup(s);
1348
1349                  // Sanity check
1350
1351                  if (count[i] == 0 || count[j] == 0) {
1352                       A(k, 2) = strdup(".");
1353                       A(k, 3) = strdup(".");
1354                       A(k, 4) = strdup(".");
1355                       A(k, 5) = strdup(".");
1356                       A(k, 6) = strdup(".  ");
1357                       continue;
1358                  }
1359
1360                  // Difference
1361
1362                  d = mean[i] - mean[j];
1363                  sprintf(buf, "%0.6f", d);
1364                  A(k, 2) = strdup(buf);
1365
1366                  // confidence interval
1367
1368                  se = sqrt(mse * (1.0 / count[i] + 1.0 / count[j]));
1369
1370                  lsd = q * se;
1371
1372                  tval = d / se;
1373
1374                  pval = 2 * (1 - tdist(fabs(tval), dfe));
1375
1376                  sprintf(buf, "%0.6f", d - lsd);
1377                  A(k, 3) = strdup(buf);
1378
```

```
1379                 sprintf(buf, "%0.6f", d + lsd);
1380                 A(k, 4) = strdup(buf);
1381
1382                 // t Value
1383
1384                 sprintf(buf, "%0.2f", tval);
1385                 A(k, 5) = strdup(buf);
1386
1387                 // Pr > |t|
1388
1389                 if (pval > alpha)
1390                      sprintf(buf, "%0.4f  ", pval);
1391                 else
1392                      sprintf(buf, "%0.4f *", pval);
1393                 A(k, 6) = strdup(buf);
1394           }
1395      }
1396
1397      buf[0] = 1; // left justify
1398      buf[1] = 1;
1399      buf[2] = 0;
1400      buf[3] = 0;
1401      buf[4] = 0;
1402      buf[5] = 0;
1403      buf[6] = 0;
1404
1405      print_table_and_free(a, nrow, ncol, buf);
1406 }
1407
1408 static void
1409 print_ttest(int x)
1410 {
1411      int dfe, i, j, k, n, ncol, nrow;
1412      char **a, *s;
1413      double d, mse, pval, se, sse, t, tval;
1414
1415      emit_line_center("Two Sample t-Test");
1416      emit_line("");
1417
1418      n = dataset->spec[x].num_levels;
1419
1420      nrow = 1 + n * (n - 1);
1421
1422      ncol = 7;
1423
1424      a = (char **) xmalloc(nrow * ncol * sizeof (char *));
1425
1426      s = dataset->spec[x].name;
1427      A(0, 0) = strdup(s);
1428      A(0, 1) = strdup(s);
1429
1430      s = dataset->spec[yvar].name;
1431      sprintf(buf, "Delta %s", s);
1432      A(0, 2) = strdup(buf);
1433
1434      sprintf(buf, "%g%% CI MIN", 100 * (1 - alpha));
1435      A(0, 3) = strdup(buf);
1436
1437      sprintf(buf, "%g%% CI MAX", 100 * (1 - alpha));
1438      A(0, 4) = strdup(buf);
```

```
1439
1440        A(0, 5) = strdup("t Value");
1441        A(0, 6) = strdup("Pr> |t|  ");
1442
1443        k = 0;
1444
1445        for (i = 0; i < n; i++) {
1446             for (j = 0; j < n; j++) {
1447
1448                  if (i == j)
1449                       continue;
1450
1451                  k++;
1452
1453                  // Level
1454
1455                  s = dataset->spec[x].ltab[i];
1456                  A(k, 0) = strdup(s);
1457
1458                  // Level
1459
1460                  s = dataset->spec[x].ltab[j];
1461                  A(k, 1) = strdup(s);
1462
1463                  // Sanity check
1464
1465                  if (count[i] == 0 || count[j] == 0) {
1466                       A(k, 2) = strdup(".");
1467                       A(k, 3) = strdup(".");
1468                       A(k, 4) = strdup(".");
1469                       A(k, 5) = strdup(".");
1470                       A(k, 6) = strdup(".  ");
1471                       continue;
1472                  }
1473
1474                  // Difference
1475
1476                  d = mean[i] - mean[j];
1477                  sprintf(buf, "%0.6f", d);
1478                  A(k, 2) = strdup(buf);
1479
1480                  // confidence interval
1481
1482                  if (count[i] + count[j] < 3) {
1483                       A(k, 3) = strdup(".");
1484                       A(k, 4) = strdup(".");
1485                       A(k, 5) = strdup(".");
1486                       A(k, 6) = strdup(".  ");
1487                       continue;
1488                  }
1489
1490                  sse = variance[i] * (count[i] - 1) + variance[j] * (count[j] - 1);
1491
1492                  dfe = count[i] + count[j] - 2;
1493
1494                  mse = sse / dfe;
1495
1496                  se = sqrt(mse * (1.0 / count[i] + 1.0 / count[j]));
1497
1498                  tval = d / se;
```

```
1499
1500                pval = 2 * (1 - tdist(fabs(tval), dfe));
1501
1502                t = qt(1 - alpha / 2, dfe) * se;
1503
1504                sprintf(buf, "%0.6f", d - t);
1505                A(k, 3) = strdup(buf);
1506
1507                sprintf(buf, "%0.6f", d + t);
1508                A(k, 4) = strdup(buf);
1509
1510                sprintf(buf, "%0.2f", tval);
1511                A(k, 5) = strdup(buf);
1512
1513                if (pval > alpha)
1514                     sprintf(buf, "%0.4f  ", pval);
1515                else
1516                     sprintf(buf, "%0.4f *", pval);
1517                A(k, 6) = strdup(buf);
1518           }
1519      }
1520
1521      buf[0] = 1; // left justify
1522      buf[1] = 1;
1523      buf[2] = 0;
1524      buf[3] = 0;
1525      buf[4] = 0;
1526      buf[5] = 0;
1527      buf[6] = 0;
1528
1529      print_table_and_free(a, nrow, ncol, buf);
1530 }
```

9.5 proc_means.c

```
1  #include "defs.h"
2
3  static char **a, s[MAXVAR + MAXSTAT + 1];
4  static int row, nrow, ncol;
5  static int filter[MAXVAR];
6
7  void
8  proc_means()
9  {
10     parse_proc_means_stmt();
11
12     if (dataset == NULL)
13          stop("No data set");
14
15     parse_proc_means_body();
16
17     print_title();
18
19     run_proc_means();
20 }
21
22 void
23 parse_proc_means_stmt()
24 {
25     for (;;) {
```

```
26              scan();
27              keyword();
28              switch (token) {
29              case ';':
30                      scan(); // eat the semicolon
31                      return;
32              case KALPHA:
33                      parse_alpha_option();
34                      break;
35              case KDATA:
36                      parse_data_option();
37                      break;
38              case KMAXDEC:
39                      parse_maxdec_option();
40                      break;
41              case KCLM:
42                      if (nstat + 2 > MAXSTAT)
43                              stop("Too many statistics keywords");
44                      stat[nstat++] = KCLM1;
45                      stat[nstat++] = KCLM2;
46                      break;
47              case KLCLM:
48              case KMAX:
49              case KMEAN:
50              case KMIN:
51              case KN:
52              case KSUM:
53              case KSTD:
54              case KSTDDEV:
55              case KSTDERR:
56              case KSTDMEAN:
57              case KUCLM:
58              case KVAR:
59                      if (nstat == MAXSTAT)
60                              stop("Too many statistics keywords");
61                      stat[nstat++] = token;
62                      break;
63              default:
64                      expected("proc means option");
65              }
66      }
67  }
68
69  void
70  parse_proc_means_body()
71  {
72      for (;;) {
73              keyword();
74              switch (token) {
75              case 0:
76              case KDATA:
77              case KPROC:
78              case KRUN:
79                      return;
80              case KCLASS:
81                      parse_class_stmt();
82                      break;
83              case KVAR:
84                      parse_var_stmt();
85                      break;
```

```
86              default:
87                      parse_default();
88                      break;
89              }
90      }
91  }
92
93  #define A(i, j) (a + (i) * ncol)[j]
94
95  void
96  run_proc_means()
97  {
98      int i, j, k, n;
99      char *t;
100
101     // print all numeric variables if not specified with VAR
102
103     if (nvar == 0) {
104         k = 0;
105         for (i = 0; i < dataset->nvar; i++)
106             if (dataset->spec[i].ltab == NULL)
107                 var[k++] = i;
108         nvar = k;
109     }
110
111     // default statistics
112
113     if (nstat == 0) {
114         nstat = 5;
115         stat[0] = KN;
116         stat[1] = KMEAN;
117         stat[2] = KSTD;
118         stat[3] = KMIN;
119         stat[4] = KMAX;
120     }
121
122     ncol = nclass + nstat + 1;
123
124     nrow = nvar;
125
126     for (i = 0; i < nclass; i++) {
127         k = class[i];
128         n = dataset->spec[k].num_levels;
129         nrow *= n;
130     }
131
132     nrow++; // for header row
133
134     a = (char **) xmalloc(nrow * ncol * sizeof (char *));
135
136     // table header line
137
138     // categorical variable names, if any
139
140     for (i = 0; i < nclass; i++) {
141         k = class[i];
142         t = dataset->spec[k].name;
143         A(0, i) = strdup(t);
144     }
145
```

```
146         // variable name column
147
148         A(0, nclass) = strdup("Variable");
149
150         // statistic names
151
152         for (i = 0; i < nstat; i++) {
153               switch (stat[i]) {
154               case KCLM1:
155                     sprintf(s, "%g%% CLM MIN", 100 * (1 - alpha));
156                     t = s;
157                     break;
158               case KCLM2:
159                     sprintf(s, "%g%% CLM MAX", 100 * (1 - alpha));
160                     t = s;
161                     break;
162               case KLCLM:
163                     sprintf(s, "%g%% LCLM", 100 * (1 - alpha));
164                     t = s;
165                     break;
166               case KMAX:
167                     t = "Maximum";
168                     break;
169               case KMEAN:
170                     t = "Mean";
171                     break;
172               case KMIN:
173                     t = "Minimum";
174                     break;
175               case KN:
176                     t = "N";
177                     break;
178               case KRANGE:
179                     t = "Range";
180                     break;
181               case KSUM:
182                     t = "Sum";
183                     break;
184               case KSTD:
185               case KSTDDEV:
186                     t = "Std Dev";
187                     break;
188               case KSTDERR:
189               case KSTDMEAN:
190                     t = "Std Err";
191                     break;
192               case KUCLM:
193                     sprintf(s, "%g%% UCLM", 100 * (1 - alpha));
194                     t = s;
195                     break;
196               case KVAR:
197                     t = "Variance";
198                     break;
199               default:
200                     t = ".";
201                     break;
202               }
203               A(0, nclass + 1 + i) = strdup(t);
204         }
205
```

```
206         row = 1;
207
208         f(0);
209
210         // right justify variable names
211
212         for (i = 0; i < ncol; i++) {
213             if (i < nclass + 1)
214                 s[i] = 1;
215             else
216                 s[i] = 0;
217         }
218
219         print_table(a, nrow, ncol, s);
220
221         for (i = 0; i < nrow; i++)
222             for (j = 0; j < ncol; j++)
223                 free(A(i, j));
224
225         free(a);
226 }
227
228 // k is the index into class[]
229
230 void
231 f(int k)
232 {
233         int i, n, varnum;
234
235         if (k == nclass) {
236             g();
237             return;
238         }
239
240         varnum = class[k];
241
242         // number of levels
243
244         n = dataset->spec[varnum].num_levels;
245
246         // for each level...
247
248         for (i = 0; i < n; i++) {
249             filter[k] = i;
250             f(k + 1);
251         }
252 }
253
254 // to here after category filter is set up
255
256 void
257 g()
258 {
259         int i, j, level, varnum;
260
261         // for each numeric variable...
262
263         for (i = 0; i < nvar; i++) {
264
265             // level names
```

```
266
267             if (i == 0)
268                  for (j = 0; j < nclass; j++) {
269                       varnum = class[j];
270                       level = filter[j];
271                       A(row, j) = strdup(dataset->spec[varnum].ltab[level]);
272                  }
273             else
274                  for (j = 0; j < nclass; j++)
275                       A(row, j) = strdup("");
276
277             varnum = var[i];
278
279             // name of the variable
280
281             A(row, nclass) = strdup(dataset->spec[varnum].name);
282
283             h(varnum);
284
285             row++;
286        }
287   }
288
289   static char *fmt[9] = {
290        "%0.0f",
291        "%0.1f",
292        "%0.2f",
293        "%0.3f",
294        "%0.4f",
295        "%0.5f",
296        "%0.6f",
297        "%0.7f",
298        "%0.8f",
299   };
300
301   // one row of statistics for varnum
302
303   void
304   h(int varnum)
305   {
306        int i, j, k, w;
307        double m, t1, t2, x;
308        static char s[100];
309
310        int n = 0;
311
312        double mean = NAN;
313        double variance = NAN;
314
315        double min = NAN;
316        double max = NAN;
317        double sum = NAN;
318
319        double stddev = NAN;
320        double stderr = NAN;
321
322        double range = NAN;
323
324        for (i = 0; i < dataset->nobs; i++) {
325
```

```
326             // filter
327
328             for (j = 0; j < nclass; j++) {
329                 k = class[j];
330                 if (dataset->spec[k].v[i] != filter[j])
331                     break;
332             }
333
334             if (j < nclass)
335                 continue;
336
337             // if categorical variable then only N makes sense
338
339             if (dataset->spec[varnum].ltab) {
340                 if (dataset->spec[varnum].v[i])
341                     n++;
342                 continue;
343             }
344
345             x = dataset->spec[varnum].v[i];
346
347             if (isnan(x))
348                 continue;
349
350             n++;
351
352             if (n == 1) {
353                 mean = x;
354                 variance = 0.0;
355                 min = x;
356                 max = x;
357                 sum = 0.0;
358                 stddev = 0.0;
359                 stderr = 0.0;
360             }
361
362             m = mean;
363
364             mean += (x - m) / n;
365
366             variance += (x - m) * (x - mean);
367
368             if (x < min)
369                 min = x;
370
371             if (x > max)
372                 max = x;
373
374             sum += x;
375         }
376
377         range = max - min;
378
379         if (n > 1) {
380             variance /= (n - 1);
381             stddev = sqrt(variance);
382             stderr = stddev / sqrt(n);
383         }
384
385         t1 = qt(1 - alpha, n - 1.0);
```

```
386         t2 = qt(1 - alpha / 2, n - 1.0);
387
388         for (i = 0; i < nstat; i++) {
389             w = maxdec;
390             switch (stat[i]) {
391             case KCLM1:
392                 x = mean - t2 * stderr;
393                 break;
394             case KCLM2:
395                 x = mean + t2 * stderr;
396                 break;
397             case KLCLM:
398                 x = mean - t1 * stderr;
399                 break;
400             case KMAX:
401                 x = max;
402                 break;
403             case KMEAN:
404                 x = mean;
405                 break;
406             case KMIN:
407                 x = min;
408                 break;
409             case KN:
410                 x = n;
411                 w = 0;
412                 break;
413             case KRANGE:
414                 x = range;
415                 break;
416             case KSUM:
417                 x = sum;
418                 break;
419             case KSTD:
420             case KSTDDEV:
421                 x = stddev;
422                 break;
423             case KSTDERR:
424             case KSTDMEAN:
425                 x = stderr;
426                 break;
427             case KUCLM:
428                 x = mean + t1 * stderr;
429                 break;
430             case KVAR:
431                 x = variance;
432                 break;
433             default:
434                 x = NAN;
435                 break;
436             }
437             if (isnan(x))
438                 A(row, nclass + 1 + i) = strdup(".");
439             else {
440                 sprintf(s, fmt[w], x);
441                 A(row, nclass + 1 + i) = strdup(s);
442             }
443         }
444 }
```

9.6 proc_print.c

```
1  #include "defs.h"
2
3  static char *fmt[9] = {
4        "%0.0f",
5        "%0.1f",
6        "%0.2f",
7        "%0.3f",
8        "%0.4f",
9        "%0.5f",
10       "%0.6f",
11       "%0.7f",
12       "%0.8f",
13 };
14
15 void
16 proc_print()
17 {
18       parse_proc_print_stmt();
19
20       if (dataset == NULL)
21             stop("No data set");
22
23       parse_proc_print_body();
24
25       run_proc_print();
26 }
27
28 void
29 parse_proc_print_stmt(void)
30 {
31       for (;;) {
32             get_next_token();
33             switch (token) {
34             case ';':
35                   scan();
36                   return;
37             case KDATA:
38                   parse_data_option();
39                   break;
40             default:
41                   stop("';' expected");
42             }
43       }
44 }
45
46 // parse statements that follow PROC PRINT
47
48 void
49 parse_proc_print_body(void)
50 {
51       for (;;) {
52             keyword();
53             switch (token) {
54             case 0:
55             case KDATA:
56             case KPROC:
57             case KRUN:
58                   return;
```

```
59              case KVAR:
60                      parse_var_stmt();
61                      break;
62              default:
63                      parse_default();
64                      break;
65              }
66      }
67  }
68
69  #define A(i, j) (a + (i) * (nvar + 1))[j]
70
71  static char buf[100];
72
73  void
74  run_proc_print(void)
75  {
76      int i, j, nobs, w, x;
77      double d;
78      char **a, *s;
79
80      print_title();
81
82      nobs = dataset->nobs;
83
84      // print all variables if not specified with VAR
85
86      if (nvar == 0) {
87              nvar = dataset->nvar;
88              for (i = 0; i < nvar; i++)
89                      var[i] = i;
90      }
91
92      // create array
93
94      a = (char **) xmalloc((nobs + 1) * (nvar + 1) * sizeof (char *));
95
96      // column names
97
98      A(0, 0) = strdup("Obs");
99
100     for (j = 0; j < nvar; j++) {
101             x = var[j];
102             A(0, j + 1) = strdup(dataset->spec[x].name);
103     }
104
105     // for each row
106
107     for (i = 0; i < nobs; i++) {
108
109             // observation number
110
111             sprintf(buf, "%d", i + 1);
112
113             A(i + 1, 0) = strdup(buf);
114
115             // for each variable
116
117             for (j = 0; j < nvar; j++) {
118
```

```
119                 x = var[j];
120
121                 d = dataset->spec[x].v[i];
122
123                 if (isnan(d))
124                     s = ".";
125                 else {
126                     if (dataset->spec[x].ltab == NULL) {
127                         w = dataset->spec[x].w;
128                         sprintf(buf, fmt[w], d);
129                         s = buf;
130                     } else
131                         s = dataset->spec[x].ltab[(int) d];
132                 }
133
134                 A(i + 1, j + 1) = strdup(s);
135             }
136         }
137
138         s = (char *) xmalloc(nvar + 1);
139
140         s[0] = 0; // observation numbers are right justified
141
142         for (j = 0; j < nvar; j++) {
143             x = var[j];
144             if (dataset->spec[x].ltab == NULL)
145                 s[j + 1] = 0;
146             else
147                 s[j + 1] = 1;
148         }
149
150         print_table(a, nobs + 1, nvar + 1, s);
151
152         free(s);
153
154         for (i = 0; i < nobs + 1; i++)
155             for (j = 0; j < nvar + 1; j++)
156                 free(A(i, j));
157
158         free(a);
159 }
```

9.7 proc_reg.c

```
1  #include "defs.h"
2
3  static int noint;
4
5  // explanatory variables from the MODEL statement
6  static int num_x;
7  static int xtab[MAXVAR];
8
9  // response variables from the MODEL statement
10 static int num_y;
11 static int ytab[MAXVAR];
12
13 static void parse_proc_reg_stmt();
14 static void parse_proc_reg_body();
15 static void parse_model_stmt();
16 static void parse_model_options();
```

```
17
18  void
19  proc_reg()
20  {
21      num_x = 0;
22      num_y = 0;
23
24      parse_proc_reg_stmt();
25
26      if (dataset == NULL)
27          stop("No data set");
28
29      parse_proc_reg_body();
30
31      regress();
32
33      print_title();
34
35      emit_line_center("Analysis of Variance");
36      emit_line("");
37
38      print_anova_table();
39      print_diag_table();
40
41      emit_line_center("Parameter Estimates");
42      emit_line("");
43
44      print_parameter_estimates();
45  }
46
47  static void
48  parse_proc_reg_stmt()
49  {
50      for (;;) {
51          scan();
52          keyword();
53          switch (token) {
54          case ';':
55              scan(); // eat the semicolon
56              return;
57          case KALPHA:
58              parse_alpha_option();
59              break;
60          case KDATA:
61              parse_data_option();
62              break;
63          default:
64              expected("proc reg option");
65          }
66      }
67  }
68
69  static void
70  parse_proc_reg_body()
71  {
72      for (;;) {
73          keyword();
74          switch (token) {
75          case 0:
76          case KDATA:
```

```
77              case KPROC:
78              case KRUN:
79                      return;
80              case KMODEL:
81                      parse_model_stmt();
82                      break;
83              default:
84                      parse_default();
85                      break;
86              }
87      }
88  }
89
90  static void
91  parse_model_stmt(void)
92  {
93      int i;
94
95      noint = 0;
96
97      num_x = 0;
98      num_y = 0;
99
100     scan();
101
102     if (token != NAME)
103         expected("variable name");
104
105     // look for match in dataset
106
107     for (i = 0; i < dataset->nvar; i++)
108         if (strcmp(dataset->spec[i].name, strbuf) == 0)
109             break;
110
111     if (i == dataset->nvar) {
112         sprintf(errbuf, "Variable %s not in dataset", strbuf);
113         stop(errbuf);
114     }
115
116     ytab[num_y++] = i;
117
118     scan();
119
120     if (token != '=')
121         expected("equals sign");
122
123     for (;;) {
124
125         scan();
126
127         if ((token == ';' || token == '/') && num_x)
128             break;
129
130         if (token != NAME)
131             expected("variable name");
132
133         // look for match in dataset
134
135         for (i = 0; i < dataset->nvar; i++)
136             if (strcmp(dataset->spec[i].name, strbuf) == 0)
```

```
137                     break;
138
139             if (i == dataset->nvar) {
140                     sprintf(errbuf, "Variable %s not in dataset", strbuf);
141                     stop(errbuf);
142             }
143
144             xtab[num_x++] = i;
145     }
146
147     if (token == '/')
148             parse_model_options();
149
150     scan(); // eat the semicolon
151 }
152
153 static void
154 parse_model_options()
155 {
156     for (;;) {
157
158             scan();
159
160             keyword();
161
162             switch (token) {
163
164             case ';':
165                     return;
166
167             case KNOINT:
168                     noint = 1;
169                     break;
170
171             default:
172                     stop("';' or MODEL option expected");
173             }
174     }
175 }
176
177 static int nrow;
178 static int ncol;
179
180 static int npar;
181
182 static double ybar;
183
184 static double ssr;
185 static double sse;
186 static double sst;
187
188 static double msr;
189 static double mse;
190
191 static double fval;
192 static double pval;
193
194 static double rsquare; // aka coefficient of determination
195 static double adjrsq; // see KNNL p. 226
196 static double rootmse;
```

```
197 static double cv;
198 static int dfm; // degrees of freedom model
199 static int dfe; // degrees of freedom error
200 static int dft; // degrees of freedom total
201
202 static int *Z;
203
204 static double *B;
205 static double *Y;
206 static double *SE;
207 static double *TVAL;
208 static double *PVAL;
209
210 static double *_C_;
211 static double *_G_;
212 static double *_T_;
213 static double *_X_;
214
215 #define C(i, j) (_C_ + (i) * ncol)[j]
216 #define G(i, j) (_G_ + (i) * ncol)[j]
217 #define T(i, j) (_T_ + (i) * ncol)[j]
218 #define X(i, j) (_X_ + (i) * ncol)[j]
219
220 // X is the design matrix
221
222 void
223 compute_X()
224 {
225     int i, j, k, l, m, n, x, y;
226     double v;
227
228     l = 0;
229
230     for (i = 0; i < nrow; i++) {
231
232         // check for missing data
233
234         y = ytab[0];
235
236         v = dataset->spec[y].v[i];
237
238         if (isnan(v))
239             continue;
240
241         Y[l] = v;
242
243         if (noint)
244             m = 0;
245         else {
246             m = 1;
247             X(l, 0) = 1;
248         }
249
250         for (j = 0; j < num_x; j++) {
251
252             x = xtab[j];
253
254             v = dataset->spec[x].v[i];
255
256             if (isnan(v))
```

```
257                         break;
258
259                 if (dataset->spec[x].ltab == NULL)
260                         X(l, m++) = v;
261                 else {
262
263                         // categorical variable
264
265                         n = dataset->spec[x].num_levels;
266
267                         for (k = 0; k < n; k++)
268                                 if (k == v)
269                                         X(l, m + k) = 1;
270                                 else
271                                         X(l, m + k) = 0;
272
273                         m += n;
274                 }
275         }
276
277         if (j == num_x)
278                 l++; // full row, no nans
279     }
280
281     // missing data may reduce nrow
282
283     nrow = l;
284 }
285
286 // T = X^T * X
287
288 void
289 compute_T()
290 {
291     int i, j, k, l, m;
292     double t;
293     l = 0;
294     for (i = 0; i < ncol; i++) {
295         if (Z[i])
296             continue;
297         m = 0;
298         for (j = 0; j < ncol; j++) {
299             if (Z[j])
300                 continue;
301             t = 0;
302             for (k = 0; k < nrow; k++)
303                 t += X(k, i) * X(k, j);
304             T(l, m) = t;
305             m++;
306         }
307         l++;
308     }
309 }
310
311 // G = T ^ (-1)
312
313 // T is clobbered
314
315 int
316 compute_G()
```

```
317 {
318     int d, i, j, k;
319     double m, max, min, t;
320
321     min = 0;
322     max = 0;
323
324     // G = I
325
326     for (i = 0; i < npar; i++)
327         for (j = 0; j < npar; j++)
328             if (i == j)
329                 G(i, j) = 1;
330             else
331                 G(i, j) = 0;
332
333     for (d = 0; d < npar; d++) {
334
335         // find the best pivot row
336
337         k = d;
338         for (i = d + 1; i < npar; i++)
339             if (fabs(T(i, d)) > fabs(T(k, d)))
340                 k = i;
341
342         // exchange rows if necessary
343
344         if (k != d) {
345             for (j = d; j < npar; j++) { // skip zeroes, start at d
346                 t = T(d, j);
347                 T(d, j) = T(k, j);
348                 T(k, j) = t;
349             }
350             for (j = 0; j < npar; j++) {
351                 t = G(d, j);
352                 G(d, j) = G(k, j);
353                 G(k, j) = t;
354             }
355         }
356
357         // multiply the pivot row by 1 / pivot
358
359         m = T(d, d);
360
361         if (m == 0)
362             return -1;
363
364         if (fabs(m) > max)
365             max = fabs(m);
366
367         m = 1 / m;
368
369         if (fabs(m) > min)
370             min = fabs(m);
371
372         for (j = d; j < npar; j++) // skip zeroes, start at d
373             T(d, j) *= m;
374         for (j = 0; j < npar; j++)
375             G(d, j) *= m;
376
```

```
377             // clear out column below d
378
379             for (i = d + 1; i < npar; i++) {
380                 m = -T(i, d);
381                 for (j = d; j < npar; j++) // skip zeroes, start at d
382                     T(i, j) += m * T(d, j);
383                 for (j = 0; j < npar; j++)
384                     G(i, j) += m * G(d, j);
385             }
386         }
387
388         // clear out columns above diagonal
389
390         for (d = npar - 1; d > 0; d--)
391             for (i = 0; i < d; i++) {
392                 m = -T(i, d);
393                 for (j = 0; j < npar; j++)
394                     G(i, j) += m * G(d, j);
395             }
396
397         // check ratio of biggest divisor to smallest divisor
398
399         // domain is [1, inf)
400
401         // printf("cond = %g\n", max * min);
402
403         if (max * min < 1e10)
404             return 0;
405         else
406             return -1;
407 }
408
409 // B = G * X^T * Y
410
411 static void
412 compute_B()
413 {
414         int i, j, l;
415         double t;
416
417         l = 0;
418
419         for (i = 0; i < ncol; i++) {
420             if (Z[i])
421                 continue;
422             t = 0;
423             for (j = 0; j < nrow; j++)
424                 t += X(j, i) * Y[j];
425             T(0, l++) = t;
426         }
427
428         for (i = 0; i < npar; i++) {
429             t = 0;
430             for (j = 0; j < npar; j++)
431                 t += G(i, j) * T(0, j);
432             B[i] = t;
433         }
434 }
435
436 // E = Y - X * B
```

```
437
438 // sse = E^T * E
439
440 // mse = sse / (nrow - npar)
441
442 void
443 compute_mse()
444 {
445     int i, j, k;
446     double yhat;
447
448     dfm = npar - 1;
449     dfe = nrow - npar;
450     dft = nrow - 1;
451
452     ybar = 0;
453
454     for (i = 0; i < nrow; i++)
455         ybar += (Y[i] - ybar) / (i + 1);
456
457     ssr = 0;
458     sse = 0;
459     sst = 0;
460
461     for (i = 0; i < nrow; i++) {
462         k = 0;
463         yhat = 0;
464         for (j = 0; j < ncol; j++)
465             if (Z[j] == 0)
466                 yhat += X(i, j) * B[k++];
467         ssr += (yhat - ybar) * (yhat - ybar);
468         sse += (Y[i] - yhat) * (Y[i] - yhat);
469         sst += (Y[i] - ybar) * (Y[i] - ybar);
470     }
471
472     mse = sse / dfe;
473
474     rootmse = sqrt(mse);
475
476     msr = ssr / dfm;
477
478     fval = msr / mse;
479
480     pval = 1 - fdist(fval, dfm, dfe);
481
482     rsquare = 1 - sse / sst;
483
484     adjrsq = 1 - (double) dft / dfe * sse / sst;
485
486     cv = 100 * rootmse / ybar;
487 }
488
489 // C = mse * G
490
491 void
492 compute_C()
493 {
494     int i, j;
495     for (i = 0; i < npar; i++)
496         for (j = 0; j < npar; j++)
```

```
497                 C(i, j) = mse * G(i, j);
498 }
499
500 // SE[i] = sqrt(C[i][i])
501
502 void
503 compute_SE()
504 {
505         int i;
506         for (i = 0; i < npar; i++)
507                 SE[i] = sqrt(C(i, i));
508 }
509
510 void
511 compute_TVAL()
512 {
513         int i;
514         for (i = 0; i < npar; i++)
515                 TVAL[i] = B[i] / SE[i];
516 }
517
518 void
519 compute_PVAL()
520 {
521         int i, n;
522         n = nrow - npar;
523         for (i = 0; i < npar; i++)
524                 PVAL[i] = 2 * (1 - tdist(fabs(TVAL[i]), n));
525 }
526
527 void
528 print_B()
529 {
530         int i;
531         printf("B =\n");
532         for (i = 0; i < npar; i++)
533                 printf("%20g\n", B[i]);
534 }
535
536 void
537 print_X()
538 {
539         int i, j;
540         printf("X =\n");
541         for (i = 0; i < nrow; i++) {
542                 for (j = 0; j < ncol; j++)
543                         printf("%20g", X(i, j));
544                 printf("\n");
545         }
546 }
547
548 void
549 print_T()
550 {
551         int i, j;
552         printf("T =\n");
553         for (i = 0; i < npar; i++) {
554                 for (j = 0; j < npar; j++)
555                         printf("%20g", T(i, j));
556                 printf("\n");
```

```
557         }
558 }
559
560 void
561 print_G()
562 {
563         int i, j;
564         printf("G =\n");
565         for (i = 0; i < npar; i++) {
566                 for (j = 0; j < npar; j++)
567                         printf("%20g", G(i, j));
568                 printf("\n");
569         }
570 }
571
572 void
573 print_Z()
574 {
575         int i;
576         printf("Z =\n");
577         for (i = 0; i < ncol; i++)
578                 printf("%20d\n", Z[i]);
579 }
580
581 void
582 regress()
583 {
584         int i, x;
585
586         nrow = dataset->nobs;
587
588         // determine the number of design matrix columns
589
590         if (noint)
591                 ncol = 0;
592         else
593                 ncol = 1;
594
595         for (i = 0; i < num_x; i++) {
596                 x = xtab[i];
597                 if (dataset->spec[x].ltab == NULL)
598                         ncol++;
599                 else
600                         ncol += dataset->spec[x].num_levels;
601         }
602
603         if (Z) {
604                 free(Z);
605                 free(Y);
606                 free(B);
607                 free(SE);
608                 free(TVAL);
609                 free(PVAL);
610                 free(_C_);
611                 free(_G_);
612                 free(_T_);
613                 free(_X_);
614         }
615
616         Z = xmalloc(ncol * sizeof (int));
```

```
617
618         Y = xmalloc(nrow * sizeof (double));
619         B = xmalloc(ncol * sizeof (double));
620         SE = xmalloc(ncol * sizeof (double));
621         TVAL = xmalloc(ncol * sizeof (double));
622         PVAL = xmalloc(ncol * sizeof (double));
623
624         _C_ = xmalloc(ncol * ncol * sizeof (double));
625         _G_ = xmalloc(ncol * ncol * sizeof (double));
626         _T_ = xmalloc(ncol * ncol * sizeof (double));
627         _X_ = xmalloc(nrow * ncol * sizeof (double));
628
629         compute_X();
630
631         npar = ncol;
632
633         for (i = 0; i < ncol; i++)
634               Z[i] = 0;
635
636         compute_T();
637
638         // if singular then put in columns one by one
639
640         if (compute_G() == -1) {
641               npar = 0;
642               for (i = 0; i < ncol; i++)
643                     Z[i] = 1;
644               for (i = 0; i < ncol; i++) {
645                     npar++;
646                     Z[i] = 0;
647                     compute_T();
648                     if (compute_G() == -1) {
649                           npar--;
650                           Z[i] = 1;
651                     }
652               }
653
654               // did last column get zapped?
655
656               if (Z[ncol - 1]) {
657                     compute_T();
658                     compute_G();
659               }
660         }
661
662         // sanity check
663
664         if (npar < 1 || npar >= nrow) {
665               sprintf(errbuf, "Regression model kaput, p = %d, n = %d, must have 0 < p < n", npar, nrow
666               stop(errbuf);
667         }
668
669         compute_B();
670
671         compute_mse();
672
673         compute_C();
674
675         compute_SE();
676
```

```
677         compute_TVAL();
678
679         compute_PVAL();
680     }
681
682     #define A(i, j) (a + 5 * (i))[j]
683
684     void
685     print_parameter_estimates()
686     {
687         int i, j, k, m, n, x;
688         static char s[100];
689
690         m = ncol + 1;
691
692         char **a = xmalloc(m * 5 * sizeof (char *));
693
694         A(0, 0) = strdup("");
695         A(0, 1) = strdup("Estimate");
696         A(0, 2) = strdup("Std Err");
697         A(0, 3) = strdup("t Value");
698         A(0, 4) = strdup("Pr > |t|");
699
700         // variable names
701
702         k = 1;
703
704         if (noint == 0)
705             A(k++, 0) = strdup("Intercept");
706
707         for (i = 0; i < num_x; i++) {
708             x = xtab[i];
709             if (dataset->spec[x].ltab == NULL)
710                 A(k++, 0) = strdup(dataset->spec[x].name);
711             else {
712                 n = dataset->spec[x].num_levels;
713                 for (j = 0; j < n; j++) {
714                     sprintf(s, "%s %s", dataset->spec[x].name, dataset->spec[x].ltab[j]); // FIXME
715                     A(k++, 0) = strdup(s);
716                 }
717             }
718         }
719
720         k = 0;
721
722         for (i = 1; i < m; i++) {
723
724             if (Z[i - 1]) {
725                 A(i, 1) = strdup(".");
726                 A(i, 2) = strdup(".");
727                 A(i, 3) = strdup(".");
728                 A(i, 4) = strdup(".");
729                 continue;
730             }
731
732             sprintf(s, "%0.5f", B[k]);
733             A(i, 1) = strdup(s);
734
735             sprintf(s, "%0.5f", SE[k]);
736             A(i, 2) = strdup(s);
```

```
737
738             sprintf(s, "%0.2f", TVAL[k]);
739             A(i, 3) = strdup(s);
740
741             sprintf(s, "%0.4f", PVAL[k]);
742             A(i, 4) = strdup(s);
743
744             k++;
745         }
746
747         s[0] = 1; // right justify
748         s[1] = 0;
749         s[2] = 0;
750         s[3] = 0;
751         s[4] = 0;
752
753         print_table(a, m, 5, s);
754
755         for (i = 0; i < m; i++)
756             for (j = 0; j < 5; j++)
757                 free(A(i, j));
758
759         free(a);
760 }
761
762 #undef A
763 #define A(i, j) (a + 6 * (i))[j]
764
765 void
766 print_anova_table()
767 {
768         int i, j;
769         char **a, s[100];
770
771         a = (char **) xmalloc(4 * 6 * sizeof (char *));
772
773         // 1st row
774
775         A(0, 0) = strdup("");
776         A(0, 1) = strdup("DF");
777         A(0, 2) = strdup("Sum of Squares");
778         A(0, 3) = strdup("Mean Square");
779         A(0, 4) = strdup("F Value");
780         A(0, 5) = strdup("Pr > F");
781
782         // 2nd row
783
784         A(1, 0) = strdup("Model");
785
786         sprintf(s, "%d", dfm);
787         A(1, 1) = strdup(s);
788
789         sprintf(s, "%0.5f", ssr);
790         A(1, 2) = strdup(s);
791
792         sprintf(s, "%0.5f", msr);
793         A(1, 3) = strdup(s);
794
795         sprintf(s, "%0.2f", fval);
796         A(1, 4) = strdup(s);
```

```
797
798         sprintf(s, "%0.4f", pval);
799         A(1, 5) = strdup(s);
800
801         // 3rd row
802
803         A(2, 0) = strdup("Error");
804
805         sprintf(s, "%d", dfe);
806         A(2, 1) = strdup(s);
807
808         sprintf(s, "%0.5f", sse);
809         A(2, 2) = strdup(s);
810
811         sprintf(s, "%0.5f", mse);
812         A(2, 3) = strdup(s);
813
814         A(2, 4) = strdup("");
815         A(2, 5) = strdup("");
816
817         // 4th row
818
819         A(3, 0) = strdup("Total");
820
821         sprintf(s, "%d", dft);
822         A(3, 1) = strdup(s);
823
824         sprintf(s, "%0.5f", sst);
825         A(3, 2) = strdup(s);
826
827         A(3, 3) = strdup("");
828         A(3, 4) = strdup("");
829         A(3, 5) = strdup("");
830
831         s[0] = 1; // left justify
832         s[1] = 0;
833         s[2] = 0;
834         s[3] = 0;
835         s[4] = 0;
836         s[5] = 0;
837
838         print_table(a, 4, 6, s);
839
840         for (i = 0; i < 4; i++)
841             for (j = 0; j < 6; j++)
842                 free(A(i, j));
843
844         free(a);
845 }
846
847 #if 0
848
849 // diag table style for proc glm
850
851 void
852 print_diag_table()
853 {
854         char *a[2][4], s[100];
855
856         // 1st row
```

```
857
858         a[0][0] = "R-Square";
859         a[0][1] = "Coeff Var";
860         a[0][2] = "Root MSE";
861         a[0][3] = "Y Mean";
862
863         // 2nd row
864
865         sprintf(s, "%0.6f", rsquare);
866         a[1][0] = strdup(s);
867
868         sprintf(s, "%0.6f", cv);
869         a[1][1] = strdup(s);
870
871         sprintf(s, "%0.6f", rootmse);
872         a[1][2] = strdup(s);
873
874         sprintf(s, "%0.6f", ybar);
875         a[1][3] = strdup(s);
876
877         s[0] = 0;
878         s[1] = 0;
879         s[2] = 0;
880         s[3] = 0;
881
882         print_table(&a[0][0], 2, 4, s);
883
884         free(a[1][0]);
885         free(a[1][1]);
886         free(a[1][2]);
887         free(a[1][3]);
888 }
889
890 #else
891
892 void
893 print_diag_table()
894 {
895         char *a[3][4], s[100];
896
897         a[0][0] = "Root MSE";
898         a[1][0] = "Dependent Mean";
899         a[2][0] = "Coeff Var";
900
901         a[0][2] = "R-Square";
902         a[1][2] = "Adj R-Sq";
903         a[2][2] = "";
904
905         sprintf(s, "%0.5f", rootmse);
906         a[0][1] = strdup(s);
907
908         sprintf(s, "%0.5f", ybar);
909         a[1][1] = strdup(s);
910
911         sprintf(s, "%0.5f", cv);
912         a[2][1] = strdup(s);
913
914         sprintf(s, "%0.4f", rsquare);
915         a[0][3] = strdup(s);
916
```

```
917         sprintf(s, "%0.4f", adjrsq);
918         a[1][3] = strdup(s);
919
920         a[2][3] = "";
921
922         s[0] = 1; // left justify
923         s[1] = 0;
924         s[2] = 1;
925         s[3] = 0;
926
927         print_table(&a[0][0], 3, 4, s);
928
929         free(a[0][1]);
930         free(a[1][1]);
931         free(a[2][1]);
932
933         free(a[0][3]);
934         free(a[1][3]);
935 }
936
937 #endif
```

9.8 scan.c

```
1  #include "defs.h"
2
3  char strbuf[100];
4
5  void
6  scan(void)
7  {
8       static int t;
9
10      if (inp == pgm)
11           t = 0; // initial state
12      else
13           t = token;
14
15      token = scan1();
16
17      // ensure last token is semicolon
18
19      if (token == 0 && t && t != ';') {
20           strbuf[0] = ';';
21           strbuf[1] = 0;
22           token = ';';
23      }
24 #if 0
25      if (token < 128)
26           printf("token = %c\n", token);
27      else
28           printf("token = %d\n", token);
29 #endif
30 }
31
32 int
33 scan1(void)
34 {
35      int i, n;
36
```

```
37        while (*inp == '\t' || *inp == ' ')
38             inp++;
39
40        token_str = inp;
41
42        // end?
43
44        if (*inp == '\0') {
45             strbuf[0] = 0;
46             return 0;
47        }
48
49        // end of line?
50
51        if (*inp == '\n' || *inp == '\r') {
52             while (isspace(*inp))
53                  inp++;
54             strbuf[0] = ';';
55             strbuf[1] = 0;
56             return ';';
57        }
58
59        // name?
60
61        if (isalpha(*inp) || *inp == '_') {
62             for (i = 0; i < sizeof strbuf; i++) {
63                  if (isalnum(*inp) || *inp == '_')
64                       strbuf[i] = toupper(*inp++);
65                  else
66                       break;
67             }
68             if (i == sizeof strbuf)
69                  stop("Name too long");
70             strbuf[i] = 0;
71             return NAME;
72        }
73
74        // number?
75
76        if (isdigit(*inp)
77        || (inp[0] == '.' && isdigit(inp[1]))
78        || ((*inp == '+' || *inp == '-')
79        && (isdigit(inp[1]) || (inp[1] == '.' && isdigit(inp[2]))))) {
80             if (*inp == '+' || *inp == '-')
81                  inp++;
82             while (isdigit(*inp))
83                  inp++;
84             if (*inp == '.') {
85                  inp++;
86                  while (isdigit(*inp))
87                       inp++;
88             }
89             if (*inp == 'E' || *inp == 'e') {
90                  inp++;
91                  if (*inp == '+' || *inp == '-')
92                       inp++;
93                  while (isdigit(*inp))
94                       inp++;
95             }
96             n = (int) (inp - token_str);
```

```
97              if (n >= sizeof strbuf)
98                      stop("Number format?");
99              for (i = 0; i < n; i++)
100                     strbuf[i] = token_str[i];
101             strbuf[i] = 0;
102             n = sscanf(strbuf, "%lg", &token_num);
103             if (n < 1)
104                     stop("Number format?");
105             return NUMBER;
106     }
107
108     // string?
109
110     if (*inp == '\'') {
111             inp++;
112             for (i = 0; i < sizeof strbuf; i++) {
113                     if (*inp == 0)
114                             break;
115                     if (*inp == '\r' || *inp == '\n')
116                             stop("Runaway string");
117                     if (inp[0] == '\'' && inp[1] == '\'') {
118                             strbuf[i] = *inp;
119                             inp += 2;
120                             continue;
121                     }
122                     if (*inp == '\'') {
123                             inp++;
124                             break;
125                     }
126                     strbuf[i] = *inp++;
127             }
128             if (i == sizeof strbuf)
129                     stop("String too long");
130             strbuf[i] = 0;
131             return STRING;
132     }
133
134     if (*inp == '"') {
135             inp++;
136             for (i = 0; i < sizeof strbuf; i++) {
137                     if (*inp == 0)
138                             break;
139                     if (*inp == '\r' || *inp == '\n')
140                             stop("Runaway string");
141                     if (inp[0] == '"' && inp[1] == '"') {
142                             strbuf[i] = *inp;
143                             inp += 2;
144                             continue;
145                     }
146                     if (*inp == '"') {
147                             inp++;
148                             break;
149                     }
150                     strbuf[i] = *inp++;
151             }
152             if (i == sizeof strbuf)
153                     stop("String too long");
154             strbuf[i] = 0;
155             return STRING;
156     }
```

```
157
158         // double at sign?
159
160         if (inp[0] == '@' && inp[1] == '@') {
161             strcpy(strbuf, "@@");
162             inp += 2;
163             return ATAT;
164         }
165
166         // double asterisk?
167
168         if (inp[0] == '*' && inp[1] == '*') {
169             strcpy(strbuf, "**");
170             inp += 2;
171             return STARSTAR;
172         }
173
174         // anything else is a single character token
175
176         strbuf[0] = *inp;
177         strbuf[1] = 0;
178
179         return *inp++;
180     }
181
182     struct key {
183         char *s;
184         int k;
185     } keytab[] = {
186         {"ALPHA", KALPHA},
187         {"ANOVA", KANOVA},
188         {"BY",          KBY},
189         {"CARDS", KCARDS},
190         {"CLASS", KCLASS},
191         {"CLM",         KCLM},
192         {"DATA",  KDATA},
193         {"DATALINES",  KDATALINES},
194         {"DELIMITER",  KDELIMITER},
195         {"DLM",         KDLM},
196         {"FIRSTOBS",    KFIRSTOBS},
197         {"INFILE",      KINFILE},
198         {"INPUT", KINPUT},
199         {"LCLM",  KLCLM},
200         {"LSD",         KLSD},
201         {"MAX",         KMAX},
202         {"MAXDEC",      KMAXDEC},
203         {"MEAN",  KMEAN},
204         {"MEANS", KMEANS},
205         {"MIN",         KMIN},
206         {"MODEL", KMODEL},
207         {"N",           KN},
208         {"NOINT", KNOINT},
209         {"PRINT", KPRINT},
210         {"PROC",  KPROC},
211         {"RANGE", KRANGE},
212         {"REG",         KREG},
213         {"RUN",         KRUN},
214         {"STD",         KSTD},
215         {"STDDEV",      KSTDDEV},
216         {"STDERR",      KSTDERR},
```

```
217         {"STDMEAN",     KSTDMEAN},
218         {"SUM",         KSUM},
219         {"T",           KT},
220         {"TITLE",   KTITLE},
221         {"TITLE1",      KTITLE1},
222         {"TITLE2",      KTITLE2},
223         {"TITLE3",      KTITLE3},
224         {"TTEST",   KTTEST},
225         {"UCLM",    KUCLM},
226         {"VAR",         KVAR},
227         {"WELCH",   KWELCH},
228 };
229
230 void
231 keyword(void)
232 {
233         int i, n = sizeof keytab / sizeof (struct key);
234         if (token != NAME)
235                 return;
236         for (i = 0; i < n; i++)
237                 if (strcmp(strbuf, keytab[i].s) == 0) {
238                         token = keytab[i].k;
239                         break;
240                 }
241 }
242
243 // get data lines following DATALINES
244
245 char *
246 get_dataline(char *buf, int len)
247 {
248         int i;
249
250         // scan to end of current line
251
252         while (*inp && *inp != ';' && *inp != '\n' && *inp != '\r')
253                 inp++;
254
255         // end of input?
256
257         if (*inp == 0 || *inp == ';')
258                 return NULL;
259
260         // skip end of line
261
262         if (inp[0] == '\r' && inp[1] == '\n')
263                 inp += 2;
264         else
265                 inp += 1;
266
267         while (isspace(*inp) && *inp != '\n' && *inp != '\r')
268                 inp++;
269
270         if (*inp == 0 || *inp == ';' || *inp == '\n' || *inp == '\r')
271                 return NULL;
272
273         // copy data line into buf
274
275         for (i = 0; i < len - 1; i++) {
276                 buf[i] = *inp++;
```

```
277             if (*inp == 0 || *inp == ';' || *inp == '\n' || *inp == '\r')
278                     break;
279     }
280
281     if (i == len - 1)
282             stop("Buffer overrun");
283
284     buf[i + 1] = 0;
285
286     return buf;
287 }
288
289 void
290 get_next_token(void)
291 {
292     scan();
293     if (token == NAME)
294             keyword();
295 }
```

9.9 tdist.c

```
1  #include "defs.h"
2
3  // t-distribution quantile function, like qt() in R
4
5  double
6  qt(double p, double df)
7  {
8      int i;
9      double a, t, t1, t2;
10     if (isnan(p) || isnan(df) || df < 1.0)
11             return NAN;
12     t1 = -1000.0;
13     t2 = 1000.0;
14     for (i = 0; i < 100; i++) {
15             t = 0.5 * (t1 + t2);
16             a = tdist(t, df);
17             if (fabs(a - p) < 1e-10)
18                     break;
19             if (a < p)
20                     t1 = t;
21             else
22                     t2 = t;
23     }
24     return t;
25 }
26
27 // t-distribution cdf, like pt() in R
28
29 double
30 tdist(double t, double df)
31 {
32     double a;
33     if (isnan(t) || isnan(df) || df < 1.0)
34             return NAN;
35     a = 0.5 * betai(0.5 * df, 0.5, df / (df + t * t));
36     if (t > 0.0)
37             a = 1.0 - a;
38     return a;
```

```
}

// F-distribution quantile function, like qf() in R

double
qf(double p, double df1, double df2)
{
	int i;
	double a, t, t1, t2;
	if (isnan(p) || isnan(df1) || isnan(df2) || df1 < 1.0 || df2 < 1.0)
		return NAN;
	t1 = 0.0;
	t2 = 1000.0;
	for (i = 0; i < 100; i++) {
		t = 0.5 * (t1 + t2);
		a = fdist(t, df1, df2);
		if (fabs(a - p) < 1e-10)
			break;
		if (a < p)
			t1 = t;
		else
			t2 = t;
	}
	return t;
}

// F-distribution cdf, like pf() in R

double
fdist(double t, double df1, double df2)
{
	if (isnan(t) || isnan(df1) || isnan(df2) || df1 < 1.0 || df2 < 1.0)
		return NAN;
	if (t < 0.0)
		return 0.0;
	else
		return betai(0.5 * df1, 0.5 * df2, t / (t + df2 / df1));
}

// From Numerical Recipes in C, p. 214

double
gammln(double xx)
{
	double x,y,tmp,ser;
	static double cof[6]={76.18009172947146,-86.50532032941677,
		24.01409824083091,-1.231739572450155,
		0.1208650973866179e-2,-0.5395239384953e-5};
	int j;
	y=x=xx;
	tmp=x+5.5;
	tmp -= (x+0.5)*log(tmp);
	ser=1.000000000190015;
	for (j=0;j<=5;j++) ser += cof[j]/++y;
	return -tmp+log(2.5066282746310005*ser/x);
}

// From Numerical Recipes in C, p. 227

double
```

```
99  betai(double a, double b, double x)
100 {
101       double bt;
102       if (x < 0.0 || x > 1.0) return NAN;
103       if (x == 0.0 || x == 1.0) bt=0.0;
104       else
105             bt=exp(gammln(a+b)-gammln(a)-gammln(b)+a*log(x)+b*log(1.0-x));
106       if (x < (a+1.0)/(a+b+2.0))
107             return bt*betacf(a,b,x)/a;
108       else
109             return 1.0-bt*betacf(b,a,1.0-x)/b;
110 }
111
112 // From Numerical Recipes in C, p. 227
113
114 #define FPMIN 1.0e-30
115
116 double
117 betacf(double a, double b, double x)
118 {
119       int m,m2;
120       double aa,c,d,del,h,qab,qam,qap;
121       qab=a+b;
122       qap=a+1.0;
123       qam=a-1.0;
124       c=1.0;
125       d=1.0-qab*x/qap;
126       if (fabs(d) < FPMIN) d=FPMIN;
127       d=1.0/d;
128       h=d;
129       for (m=1;m<=100;m++) {
130             m2=2*m;
131             aa=m*(b-m)*x/((qam+m2)*(a+m2));
132             d=1.0+aa*d;
133             if (fabs(d) < FPMIN) d=FPMIN;
134             c=1.0+aa/c;
135             if (fabs(c) < FPMIN) c=FPMIN;
136             d=1.0/d;
137             h *= d*c;
138             aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
139             d=1.0+aa*d;
140             if (fabs(d) < FPMIN) d=FPMIN;
141             c=1.0+aa/c;
142             if (fabs(c) < FPMIN) c=FPMIN;
143             d=1.0/d;
144             del=d*c;
145             h *= del;
146             if (fabs(del-1.0) < 1e-10) break;
147       }
148       return h;
149 }
```