Introduction to Lisp-Stat and its applications.

Angelo J. Canty

Department of Mathematics
Swiss Federal Institute of Technology
1015 Lausanne
Switzerland

June 2, 1998

Chapter 1: Basics

- Developed in 1989 by Luke Tierney
- "`Lisp-Stat`: An Object-oriented Environment for Statistical Computing and Dynamic Graphics" by L. Tierney (1990, Wiley).
- Code and some documentation available from

  `http://www.stat.umn.edu/~luke/xls/xlsinfo/xlsinfo.html`
- Unix, Microsoft Windows, Apple Macintosh.
- Based on the `Xlisp` implementation of `Lisp`.

- `Lisp` is based on the idea of lists.
- A list is a collection of items within parentheses.
- Items in list can be compound (other lists) or simple.
- Simple items are
  - integers
  - floating-point numbers
  - strings
  - logical (`t` or `nil`)
  - symbols
- `Xlispstat` is not case sensitive.

- All lists are executed unless preceded by a single quote (').
- When a list is executed the first element is assumed to evaluate to a function, all other elements are passed to this function as arguments.

```
> (+ 1 2 3)
6
> '(+ 1 2 3)
(+ 1 2 3)
```

Data Input

- `(def` *<name> <data list>*`)`
  - *<name>* should be a symbol giving a new variable name.
  - *<data list>* is what should be stored in the variable.
  - `(def temp '(1 2 3))`
- `(read-data-file` *<filename>*`)`
  - *<filename>* is a string giving the name of a file.
  - Returns a list containing all of the data in the file.

- `(read-data-columns` *<filename> <numcols>*`)`
  - *<numcols>* is an integer specifying the number of columns
  - Returns a list of list representing a row-wise read of the data into the required number of columns.
- `(undef` *<variable name>*`)`
  - Remove a variable and free its space.

Generating Data

- Sequences of numbers : `iseq` and `rseq`
- `repeat` is used to generate patterned data.
- Pseudo-random numbers.
  - Four random number generators.
  - Seed is initialized to the system clock.
  - `make-random-state` returns a copy or changes the state.
  - `(sample` *<data> <num>*`)`returns a sample.

- Random numbers can be generated from 10 distributions.
  - `(binomial-rand k n p)`
  - `(poisson-rand n mu)`
  - `(beta-rand n a b)`
  - `(cauchy-rand n)`
  - `(chisq-rand n df)`
  - `(f-rand n df1 df2)`
  - `(gamma-rand n a)`
  - `(normal-rand n)`
  - `(t-rand n df)`
  - `(uniform-rand n)`

Miscellaneous Details

- All indices in `Lisp` are base 0.
- `(dribble` *<filename>*`)`sends a copy of commands and output to a file.
- `(savevar` *<vars>* *<fileroot>*`)`saves variable definitions in a file terminating in `.lsp`.
- `(load` *<filename>*`)`loads in saved data.
- `help` and `help*` print out brief help information.
- `apropos` lists symbols containing a given pattern.

Basic Statistical Functions

- `(mean` *<data>*`)`, `(median` *<data>*`)`
- `(standard-deviation` *<data>*`)`
- `(min` *<data>*`)`, `(max` *<data>*`)`
- `pmin` and `pmax`
- `(quantile` *<data>* *<alpha>*`)`
- `(interquartile-range` *<data>*`)`
- Standard mathematical functions such as `log`, `exp`, `sqrt`, trigonometric and hyperbolic functions and their inverses etc.
- Most functions are vectorized.
- Multiple arguments must be the same length.

Probability Distributions

- Univariate continuous distributions

Standard normal	`normal`
Standard Cauchy	`cauchy`
Beta(α, β)	`beta`
t(df)	`t`
Chi-squared(df)	`chisq`
F($df1, df2$)	`f`
Gamma($\alpha, 1$)	`gamma`

  – `-dens` for pdf, `-cdf` for cdf and `-quant` for quantiles.

- Univariate discrete

Binomial(n, p)	`binomial`
Poisson(λ)	`poisson`

  – `-pmf` for probability mass function, `-cdf` for cdf and `-quant` for quantiles.

- `(bivnorm-cdf` *<x>* *<y>* *<r>*`)`
  Standard bivariate normal cdf with correlation r.

Univariate Plots

- `(histogram` *<data>*`)`
- `(boxplot` *<data-list>*`)`
  <data-list> can be a single dataset or a list of datasets for parallel boxplots.
- `probability-plot` and `quantile-plot`.
  - Use `normal-cdf` or `normal-quant` by default.
  - Can specify other compatible functions to change distribution.

Bivariate Plots

- `(plot-points` <*x-data*> <*y-data*>`)`
- `(plot-lines` <*x-data*> <*y-data*>`)`
- `(plot-function` <*function*> <*lower*> <*upper*>`)`

Higher Dimensions

- `(spin-plot` <*data-list*>`)`
- `(scatterplot-matrix` <*data-list*>`)`
- `(spin-function` <*fn*> <*xl*> <*xu*> <*yl*> <*yu*>`)`

Interacting with Plots

- Each plot has a menu.
- Mouse can be used to select points.
- Two or more plots can be linked.
- Plots are implemented as objects.
- Can send `messages` to plot objects.
- Requires that plot is assigned to a symbol.
- `(send`<*plot*>`:help)` lists all possible messages.

- Some useful messages for a 2D plot are :

```
:abline
:add-lines
:clear-lines
:point-label
:variable-label
:point-symbol
:point-color
:use-color
:redraw
```

- Messages are sent using the `send` function.
- The message name is always preceded by a colon (`:`)

Chapter 2: Statistics with Lisp-Stat

Linear Regression

- `regression-model` returns an object.
- Prints out basic summary information.
- Many messages to get other details.
- `:plot-residuals` and `:plot-bayes-residuals`.
- Other plots must be constructed explicitly.
- `:leverages` and `:cooks-distance` can be used for diagnostics.
- Linking of diagnostic plots is often useful.

Stack-loss Data Example

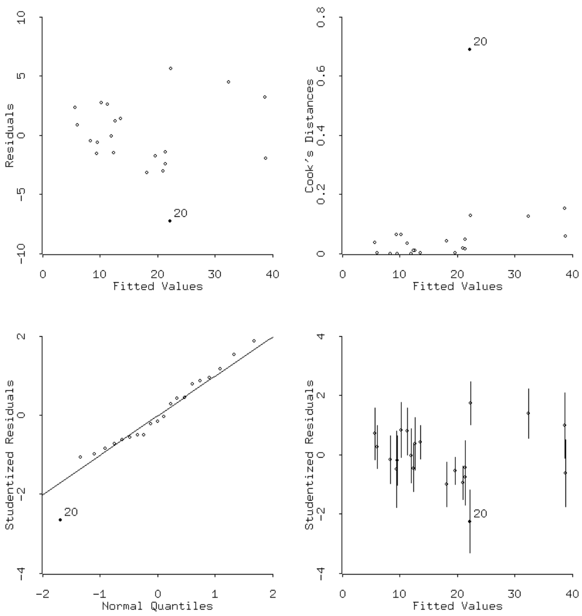
```
> (def model1 (regression-model (list air temp conc) loss
        :response-name "Ammonia Loss"
        :predictor-names
          (list "Air Flow" "Water Temp." "Acid Conc.")))

Least Squares Estimates:

Constant                   -39.9197      (11.8960)
Air Flow                   0.715640      (0.134858)
Water Temp.                 1.29529      (0.368024)
Acid Conc.                -0.152123      (0.156294)

R Squared:                 0.913577
Sigma hat:                  3.24336
Number of cases:                 21
Degrees of freedom:              17

> (send model1 :help)
REGRESSION-MODEL-PROTO
Normal Linear Regression Model
Help is available on the following:

ADD-METHOD ADD-SLOT BASIS CASE-LABELS COEF-ESTIMATES
COEF-STANDARD-ERRORS COMPUTE COOKS-DISTANCES
DELETE-DOCUMENTATION DELETE-METHOD DELETE-SLOT DF DISPLAY
DOC-TOPICS DOCUMENTATION EXTERNALLY-STUDENTIZED-RESIDUALS
FIT-VALUES GET-METHOD HAS-METHOD HAS-SLOT HELP INCLUDED
INTERCEPT INTERNAL-DOC ISNEW LEVERAGES METHOD-SELECTORS
NEW NUM-CASES NUM-COEFS NUM-INCLUDED OWN-METHODS OWN-SLOTS
PARENTS PLOT-BAYES-RESIDUALS PLOT-RESIDUALS
PRECEDENCE-LIST PREDICTOR-NAMES PRINT PROTO R-SQUARED
RAW-RESIDUALS REPARENT RESIDUAL-SUM-OF-SQUARES RESIDUALS
RESPONSE-NAME RETYPE SAVE SHOW SIGMA-HAT SLOT-NAMES
SLOT-VALUE STUDENTIZED-RESIDUALS SUM-OF-SQUARES SWEEP-MATRIX
TOTAL-SUM-OF-SQUARES WEIGHTS X X-MATRIX XTXINV Y
```


t-statistics for the coefficients.

```
> (/ (send model1 :coef-estimates)
     (send model1 :coef-standard-errors))
(-3.3557 5.3066 3.5196 -0.9733)
> (- 1 (t-cdf (abs *) (send model1 :df))
(0.0019 2.9E-5 0.0013 0.1720)
```

Diagnostic plots

```
> (def plot1 (send model1 :plot-residuals))
> (send plot1 :variable-label '(0 1)
                '("Fitted Values" "Residuals"))
> (def plot2 (plot-points (send model1 :fit-values)
                          (send model1 :cooks-distances)))
> (send plot2 :variable-label '(0 1)
                '("Fitted Values" "Cook's Distances"))
> (def plot3 (quantile-plot
              (send model1 :studentized-residuals)))
> (send plot3 :variable-label '(0 1)
                '("Normal Quantiles" "Studentized Residuals"))
> (send plot3 :abline 0 1)
> (def plot4 (send model1 :plot-bayes-residuals))
> (send plot4 :variable-label '(0 1)
                '("Fitted Values" "Studentized Residuals"))
```

Regression diagnostic plots for the stack loss data. The highlighted point is a point of high influence.

Generalized Linear Models

- `poissonreg-model`, `binomialreg-model` and `gammareg-model`.
- Objects similar to those produced for linear regressions.
- Default is the canonical link function.
- Link function can be specified using `:link` keyword argument.
- Many of the same messages as for linear regression.

Binomial Regression Example

- Cancer cell activity measure and remission indicator for 27 patients.

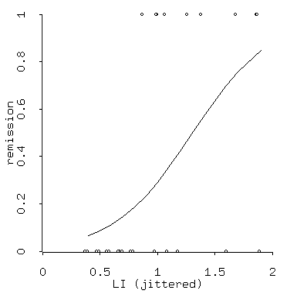
LI	0.4	0.4	0.5	0.5	0.6	0.6	0.6	0.7	0.7
r	0	0	0	0	0	0	0	0	0
LI	0.7	0.8	0.8	0.8	0.9	1.0	1.0	1.0	1.1
r	0	0	0	0	1	0	1	1	0
LI	1.1	1.2	1.3	1.4	1.6	1.7	1.9	1.9	1.9
r	1	0	1	1	0	1	0	1	1

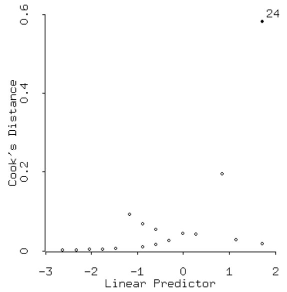
- Use logistic link (canonical)


```
> (def LI (repeat (rseq 0.4 1.9 16)
                  '(2 2 3 3 3 1 3 2 1 1 1 0 1 1 0 3)))
> (def remission
       '(0 0 0 0 0 0 0 0 0 0 0 0 0 1
         0 1 1 0 1 0 1 1 0 1 0 1 1))
> (def cancer-mod (binomialreg-model
                   LI remission (repeat 1 27)
                   :predictor-names '("LI")))

Iteration 1: deviance = 26.5268
Iteration 2: deviance = 26.0798
Iteration 3: deviance = 26.0730

Weighted Least Squares Estimates:

Constant                  -3.77714      (1.37812)
LI                         2.89726      (1.18642)

Scale taken as:            1.00000
Deviance:                  26.0730
Number of cases:                27
Degrees of freedom:             25
```


Some Plots

```
> (def LI.jitter (+ LI (/ (- (uniform-rand 27)) 25)))
> (def plot1 (plot-points LI.jitter remission
                          :variable-labels
                            '("LI (jittered)" "remission")))
> (send plot1 :add-lines LI
              (send cancer-mod :fit-probabilities))
> (def plot2 (plot-points
              (send cancer-mod :fit-values)
              (send cancer-mod :cooks-distances)
              :variable-labels
                '("Linear Predictor" "Cook's Distance")))
```

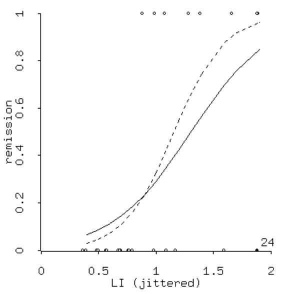




Point 24 is identified as an influential point.

Refit model excluding point 24

```
> (def incl (combine (repeat t 24) nil (repeat t 2)))
> (def cancer-mod1 (binomialreg-model
                    LI remission (repeat 1 27)
                    :predictor-names '("LI")
                    :included incl :print nil
                    :verbose nil))
> (send cancer-mod1 :coef-estimates)
(-5.2151 4.5030)
> (send cancer-mod1 :coef-standard-errors)
(1.8752 1.7629)
> (send plot1 :add-lines LI
              (send cancer-mod1 :fit-probabilities)
              :type 'dashed)
```



Poisson Model Example

- Coronary heart disease and smoking
- Model is that number of deaths $\sim$ Poisson($n\lambda$).
  - n is number of person-years,
  - λ is the death rate.
- Data taken from Breslow (1985)

	Person-years		Deaths	
Age	NS	S	NS	S
35–44	18790	52407	2	32
45–54	10673	43248	12	104
55–64	5710	28612	28	206
65–74	2585	12663	28	186
75–84	1462	5317	31	102

- Two possible models

$$\lambda = \exp(\alpha_{age} + \beta x) \quad (1)$$
$$\lambda = \exp(\alpha + \beta_1 age + \beta_2 x) \quad (2)$$


```
> (def age (repeat (* 10 (iseq 4 8)) 2))
> (def smoke (repeat '(0 1) '(5 5)))
> (def person-years
       '(18790 10673 5710 2585 1462
         52407 43248 28612 12663 5317))
> (def deaths '(2 12 28 28 31 32 104 206 186 102))
```

First fit the larger model $\lambda = \exp(\alpha_{age} + \beta x)$

```
> (def breslow-mod1 (poissonreg-model
                      (append (indicators age)
                              (list smoke))
                      deaths
                      :offset (log person-years)
                      :predictor-names
                         (append
                           (level-names age :prefix "Age")
                           '("Smoke"))))

Weighted Least Squares Estimates:

Constant                  -7.91932      (0.191412)
Age(50)                    1.48400      (0.194793)
Age(60)                    2.62750      (0.183401)
Age(70)                    3.35048      (0.184475)
Age(80)                    3.70009      (0.191904)
Smoke                     0.354534      (0.107297)

Scale taken as:            1.00000
Deviance:                  12.1324
Number of cases:                10
Degrees of freedom:              4
```


Now the reduced model $\lambda = \exp(\alpha + \beta_1 age + \beta_2 x)$

```
> (def breslow-mod2 (poissonreg-model
                      (list age smoke)
                      deaths
                      :offset (log person-years)
                      :predictor-names
                        (list "Age" "Smoke")))

Weighted Least Squares Estimates:

Constant                  -10.6258        (0.209720)
Age                         8.358337E-2   (2.904497E-3)
Smoke                       0.406370      (0.107195)

Scale taken as:             1.00000
Deviance:                   69.1821
Number of cases:                 10
Degrees of freedom:               7
```

As these models are nested can compare their deviances.

```
> (def dev-diff (- (send breslow-mod2 :deviance)
                   (send breslow-mod1 :deviance)))
> (list dev-diff
        (- 1 (chisq-cdf dev-diff
                        (- (send breslow-mod2 :df)
                           (send breslow-mod1 :df)))))
(57.04971, 2.5079E-12)
> (chisq-quant 0.95 (- (send breslow-mod2 :df)
                      (send breslow-mod1 :df)))
7.8147
```


One-way ANOVA

- `oneway-model`
- Data is a list of lists.
- Returned object very similar to regression object.

```
> (def blood '((62 60 63 59)
               (63 67 71 64 65 66)
               (68 66 71 67 68 68)
               (56 62 60 61 63 64 63 59)))
> (def blood-model (oneway-model blood))
Least Squares Estimates:

Group 0                    61.0000      (1.18322)
Group 1                    66.0000      (0.966092)
Group 2                    68.0000      (0.966092)
Group 3                    61.0000      (0.836660)

R Squared:                0.670588
Sigma hat:                 2.36643
Number of cases:                24
Degrees of freedom:             20

Group Mean Square:         76.0000      (3.00000)
Error Mean Square:         5.60000      (20.0000)
```

Nonlinear Regression

- `nreg-model`
- Normal error structure only.
- Gauss-Newton algorithm with numerical derivatives.

Approximate Bayesian Computations

- `bayes-model`
- 1st and 2nd order moment approximations.
- Laplace approximations to unimodal marginal posteriors.

Maximization

- `newtonmax` and `nelmeadmax`

Chapter 3: Lisp-Stat Programming

Data Handling

- `combine` and `append` merge lists.
- `(cons` *<item>* *<list>*`)`
- `(select` *<list>* *<indices>*`)`selects elements of a list.
- `first`, `second`, ... , `tenth`.
- `which` can be used to get indices satisfying a condition.
- `setf` can be used to change specific items of a list.
- `(def y x)` does **NOT** make a copy of `x`. Use `(def y (copy-list x))`


```
> (def x '(1 (2 3) (4 5)))
> (def y (select x 1))
> (def z (copy-list y))
> (setf (select y 0) -5)
> x
(1 (-5 3) (4 5))
> y
(-5 3)
> z
(2 3)
```


Defining Functions

`(defun` *<name>* *<par-list>* *<body>*`)`

- *<name>* can be any Lisp-Stat symbol.
- Functions and variables are kept separate.
- *<par-list>* is a list of the parameters for the function.
  - Required arguments must be first
  - `&optional` for unnamed optional arguments.
  - `&rest` for a list of all remaining unnamed arguments.
  - `&key` for keyword (named) arguments.
  - Defaults can be specified for all optional arguments.
- *<body>* is one or more valid Lisp-Stat expressions.

Conditional Evaluation

- (if *<test>* *<cons>* *<alt>*).
  - *<cons>* and *<alt>* are expressions.
  - *<alt>* is optional (defaults to `nil`).
- (cond (*<test 1>* *<expr 1>*)
  (*<test 2>* *<expr 2>*)
  ....
  (*<test n>* *<expr n>*))
  - Tests are conducted in sequence.
  - At most one expression is evaluated.
  - Common to make last test `t`.
- `case` is a simpler form of `cond` for tests of exact equality of discrete variables.

Looping

```
(do ((<name 1> <initial 1> <update 1>)
      .......
    (<name n> <initial n> <update n>))
   (<test> <result>))
```

- All bindings are local.
- Bindings done in parallel (`do*` for sequential).
- (dotimes (*<var>* *<count>* *<result>*)*<body>*).
- (dolist (*<var>* *<list>* *<result>*)*<body>*).
- `mapcar` is generally better than `dolist`.

Local Variables

```
(let ((<name 1> <expr 1>)
       .......
     (<name n> <expr n>))
    <body>)
```

- Statements in *<body>* executed with local bindings.
- Bindings in parallel.
  (use `let*` for sequential bindings).
- Free variables bound in parent environment.
- Must use `setf` to change local variables.

Local Functions

```
(flet ((<name 1> <parlist 1> <body 1>)
        .......
      (<name n> <parlist n> <body n>))
     <main body>)
```

- Local functions are visible only in *<main body>*.
- `labels` allows local functions to call previous local functions.
- Parameter lists identical to those for `defun`.
- No single environment for both local functions and variables. Must use nesting of environments.

More About Functions

- Functions can be passed to and from functions in the same way as data.
- `funcall` must be used to call a function which has been passed as data.
- `(function` <*name*>`)`or `#'`<*name*> return the function closure.
- Anonymous functions `lambda`.
- Free variables are bound in the environment in which the function was originally defined.

```
(defun michaelis (x)
     #'(lambda (theta) (/ (* (first theta) x)
                          (+ (second theta) x))))
(defun mich1 (x theta)
     (funcall (michaelis x) theta))
```


An Example — Simpson's Method

$$\int_a^{a+2h} f(x)dx \approx \frac{h}{3}[f(a) + 4f(a+h) + f(a+2h)].$$

```
(defun simpsons (f lower upper
                 &optional (n 100) (epsilon 0.01))
  (let ((h (/ (- upper lower) (* 2 n))))
    (flet ((simp (a)
              (let ((f0 (funcall f a))
                    (f1 (funcall f (+ a h)))
                    (f2 (funcall f (+ a h h))))
                 (* (/ h 3) (+ f0 (* 4 f1) f2)))))
      (do ((a lower (+ a h h))
           (int 0 (+ int (simp a))))
          ((>= a (- upper (* epsilon h))) int)))))
```


2-dimensional Simpson's method

```
(defun simpsons2D (f lower.x upper.x lower.y upper.y
                   &optional (n 100) (epsilon 0.01))
   (let ((h (/ (- upper.x lower.x) (* 2 n))))
      (flet ((simp (a)
               (let ((f0 (simpsons
                          #'(lambda (y)
                              (funcall f a y))
                          (if (numberp lower.y) lower.y
                              (funcall lower.y a))
                          (if (numberp upper.y) upper.y
                              (funcall upper.y a))
                          n epsilon))
                     (f1 (simpsons
                          #'(lambda (y)
                              (funcall f (+ a h) y))
                          (if (numberp lower.y) lower.y
                              (funcall lower.y (+ a h)))
                          (if (numberp upper.y) upper.y
                              (funcall upper.y (+ a h)))
                          n epsilon))
                     (f2 (simpsons
                          #'(lambda (y)
                              (funcall f (+ a h h) y))
                          (if (numberp lower.y) lower.y
                              (funcall lower.y (+ a h h)))
                          (if (numberp upper.y) upper.y
                              (funcall upper.y (+ a h h)))
                          n epsilon)))
                  (* (/ h 3) (+ f0 (* 4 f1) f2)))))
        (do ((a lower.x (+ a h h))
             (int 0 (+ int (simp a))))
          ((>= a (- upper.x (* epsilon h))) int)))))
```


Miscellaneous

- `dyn-load` can be used to load in `C` or `Fortran` object files.
- `compile` and `compile-file` compile `Lisp-Stat` code into a *byte code*.
  - Improves speed by a factor of at least 2.
  - Not as fast as writing in `C` or `Fortran`
  - Machine Independent.
- `error` is used to signal a fatal error.
- Comments: `;` or `#| ... |#`.
- `break` and `trace` are useful for debugging code.

Object-Oriented Programming

- *Objects* are a special type of compound data structure.
- Each object has a set of named *slots*.
- Respond to *messages* by executing a *method*.
- Hierarchical structure of inheritance.
- *Prototypes* give templates for constructing *instances*.

`(send` *<object>* *<message>* *<args>*`)`.

- Sends a message to an object.
- Causes associated method to be executed with the supplied arguments.
- Messages **always** start with a colon.

`(defmeth` *<object>* *<name>* *<arglist>* *<body>*`)`

- Structure is identical to `defun`.
- *<object>* can be a prototype or an instance.
- Documentation string can be included before *<body>*.

`(defproto` *<name>* *<instance-slots>* *<shared-slots>*
<parents> *<documentation>*`)`

- Defines a new prototype.
- If no parents supplied, inherits from `*object*`.
- *<instance-slots>* and *<shared-slots>* are lists of symbols naming the slots.
- *<shared-slots>* occur only in the prototype.
- *<instance-slots>* occur in every instance.

Data-set Objects Example

```
(defproto data-set-proto
          '(data title documentation)
          () ())
```

- Inherits from `*object*`.
- Each instance will have 3 slots plus those inherited by `data-set-proto`.
- No shared slots.
- Will respond to default methods only.

```
(send data-set-proto :help)
DATA-SET-PROTO
The root object.
Help is available on the following:

ADD-METHOD ADD-SLOT DELETE-DOCUMENTATION DELETE-METHOD
DELETE-SLOT DOC-TOPICS DOCUMENTATION GET-METHOD HAS-METHOD
HAS-SLOT HELP INTERNAL-DOC ISNEW METHOD-SELECTORS NEW
OWN-METHODS OWN-SLOTS PARENTS PRECEDENCE-LIST PRINT PROTO
REPARENT RETYPE SHOW SLOT-NAMES SLOT-VALUE
```

```
(defmeth data-set-proto :display ()
  (if (slot-value 'title)
      (princ (slot-value 'title)))
  (print (slot-value 'data))
  (terpri))

(defmeth data-set-proto :datadoc ()
   (princ (slot-value 'documentation))
   (terpri))

(defmeth data-set-proto :boxplot ()
   (boxplot (slot-value 'data)))
```


Want to distinguish between types of data-sets.

```
(defproto one-sample-proto () () data-set-proto)
(defproto two-sample-proto () () data-set-proto)
```

- Each inherits from `data-set-proto`.
- No new slots so will have the same slots as `data-set-proto`.
- Will respond to all the methods for `data-set-proto`.
- Can define new methods or overwrite existing ones for these prototypes.

Display the data in a data-set in a more informative way.

```
(defmeth one-sample-proto :display ()
  (call-next-method)
  (format t "The sample size is ~d~%"
          (length (slot-value 'data))))

(defmeth two-sample-proto :display ()
  (if (slot-value 'title)
      (princ (slot-value 'title)))
  (let ((data (slot-value 'data)))
    (print (first data))
    (terpri)
    (format t "The first sample size is ~d~%"
            (length (first data)))
    (print (second data))
    (terpri)
    (format t "The second sample size is ~d~%"
            (length (second data)))))
```

`call-next-method` calls the next `:display` method in the inheritance list.

New methods to produce a summary.

```
(defmeth one-sample-proto :summary ()
  (format t "Summary for ~a~%"
          (if (slot-value 'title)
              (slot-value 'title)
              "a univariate sample"))
  (format t "The sample size is ~d~%"
          (length (slot-value 'data)))
  (format t "The mean is ~g~%"
          (mean (slot-value 'data)))
  (format t "The standard deviation is ~g~%"
          (standard-deviation (slot-value 'data))))

(defmeth two-sample-proto :summary ()
  (format t "Summary for ~a~%"
          (if (slot-value 'title)
              (slot-value 'title)
              "a two-sample dataset"))
  (format t "The sample sizes are ~d and ~d~%"
          (length (first (slot-value 'data)))
          (length (second (slot-value 'data))))
  (format t "The means are ~g and ~g~%"
          (mean (first (slot-value 'data)))
          (mean (second (slot-value 'data))))
  (format t "The std. deviations are ~g and ~g~%"
          (standard-deviation
             (first (slot-value 'data)))
          (standard-deviation
             (second (slot-value 'data)))))
```

- To create instance send the `:new` message to the prototype.
- Executes the `:isnew` method.
- Default method has no required arguments.
- Keyword arguments specify values in slots.

```
(defmeth one-sample-proto :isnew
        (data &key title doc)
  (if (one-sample-p data)
      (call-next-method :data data
                        :title title
                        :documentation doc)
      (print "Data is not a univariate sample.")
      (print "  Object not created")))

(defmeth two-sample-proto :isnew
        (data &key title doc)
  (if (two-sample-p data)
      (call-next-method :data data
                        :title title
                        :documentation doc)
      (print "Data is not a two-sample data set.")
      (print "  Object not created")))
```



```
(defun one-sample-p (data)
  (and data
       (listp data)
       (not (reduce #'(lambda (x y) (or x y))
                    (mapcar #'listp data)))))

(defun two-sample-p (data)
  (and (= (length data) 2)
       (one-sample-p (first data))
       (one-sample-p (second data))))
```

Could also define a constructor function.

```
(defun make-dataset (data &key title doc)
   (cond ((one-sample-p data)
          (send one-sample-proto :new
                :data data
                :title title
                :documentation doc))
         ((two-sample-p data)
          (send two-sample-proto :new
                :data data
                :title title
                :documentation doc))
         (t (print "Structure not implemented"))))
```



```
(def shot-put1
     (send one-sample-proto
           :new shot-put
                :title
                   "Distances of Shot-Put Throws"))

(def gasdata
     (send two-sample-proto :new (list gas1 gas2)))

(send shot-put1 :summary)
Summary for Distances of Shot-Put Throws
The sample size is 20
The mean is 14.93
The standard deviation is 1.431

(send gasdata :summary)
Summary for a two-sample dataset
The sample sizes are 50 and 50
The means are 14.63 and 25.16
The standard deviations are 1.829 and 2.191
```


Saving Objects

- To save an object need to define a `:save` method.
- `:save` can take no arguments.
- Returns an expression which can be used to recreate the object.
- `savevar` send this message when it detects an object.
- Creates a file containing the returned expression

```
(defmeth data-set-proto :save ()
   `(make-dataset
       ',(slot-value 'data)
       :title ',(slot-value 'title)
       :documentation
         ',(slot-value 'documentation)))
```

A backquote (`) in front of a list means only elements preceded by a comma (,) are evaluated.

Chapter 4: Windows and Menus

Dialogs

- Modal Dialogs
  - User input or messages
  - Interrupt the interpreter
  - `message-dialog`, `ok-or-cancel-dialog`
  - `choose-item-dialog`, `choose-subset-dialog`
  - `get-string-dialog`, `get-value-dialog`
- Modeless Dialogs
  - Remain until dismissed.
  - Do not affect the interpreter.
  - `sequence-slider-dialog`
  - `interval-slider-dialog`

Creating Custom Dialogs

- `modal-dialog-proto` or `dialog-proto`.
- Require a list of dialog items
  - `text-item-proto`
  - `button-item-proto`
  - `modal-button-proto`
  - `choice-item-proto`
  - `toggle-item-proto`
  - `edit-text-item-proto`
  - `sequence-scroll-item-proto`
  - `interval-scroll-item-proto`
- Format defined by the order of list.

Menus

- Inherit from `menu-item`.
- Require a name only.
- Add items with `:append-items` message.
- Items inherit from `menu-item-proto`
- Must have a name and usually an `:action` function.
- `:install` and `:enabled` messages set up the menu.
- Items are enabled or disabled with the `:enabled` message.

Datasets Revisited

- Create a Dataset menu to send messages to dataset objects.
- Use dialogs to get the name of the dataset.

```
(def data-menu
     (send menu-proto :new "Data Menu"))
```

We will often need to get the dataset object so we define

```
(defun get-object ()
   (let ((name (get-string-dialog
                  "Name of Data Set")))
      (if name
          (eval (read-from-string name)))))
```

Documentation and Summary items for menu.

```
(def doc-item
     (send menu-item-proto
           :new "Documentation"
                :action
                  #'(lambda ()
                       (send (get-object) :datadoc)
                       (top-level nil))))

(def summ-item
     (send menu-item-proto
           :new "Summary"
                :action
                  #'(lambda ()
                       (send (get-object) :summary)
                       (top-level nil))))
```


Display and Boxplot menu items.

```
(def display-item
     (send menu-item-proto
           :new "Display"
                :action
                  #'(lambda ()
                       (send (get-object) :display)
                       (top-level nil))))

(def boxplot-item
     (send menu-item-proto
           :new "Boxplots"
                :action
                  #'(lambda ()
                       (send (get-object)
                             :boxplots))))
```


Now set up, install and create the menu.

```
(send data-menu :append-items
                  display-item summ-item
                  boxplot-item doc-item)
(send data-menu :enabled t)
(send data-menu :install)
(send display-item :enabled t)
(send summ-item :enabled t)
(send boxplot-item :enabled t)
(send doc-item :enabled t)
```


Statistical Graphics

- Graphics which operate on data.
- Prototypes for specific plots.
- Inherit from `graph-proto`.
- `:isnew` requires dimension of the space.
- Data supplied using `:addpoints`
- `:adjust-to-data` to redraw current version.
- Define new methods or redefine existing ones for instances of a plot.

- `x-axis` and `y-axis` supply current axes.
  Arguments : showing?, labelled?, number of ticks.

- `:range` returns or changes the range.
  Default is exact range of data.
  Reset by `:adjust-to-data`.

- Points can be changed individually.

- Line segments added with `:add-lines`.

- Fixed or variable scaling.

- Transformations performed using `:transformation` or `apply-transformation`.

- `:point-transformed-coordinate`.

Mouse modes and Plot Menus

- By default mouse mode is `selecting` or `brushing`.

- Add/delete modes with `:add-mouse-mode` or `delete-mouse-mode`.

- `:do-click` or `do-motion` messages are sent on a click or movement.

- Define new methods for click/motion actions of new mouse modes.

- Each plot has an associated menu.

- New items can be added.
  Items should inherit from `graph-item-proto`.
  Required arguments: title, graph name, action message

Example

Add an item to a plots menu to change the plot symbols.

```
(def plot1 (plot-points air temp))

(def change-item
     (send graph-item-proto
           :new "Change Symbol" plot1
                :change-symbols))

(defmeth plot1 :change-symbols ()
   (let ((i (choose-item-dialog
               "New Plot Symbol"
               '("Dot" "Dot1" "Dot2" "Dot3" "Dot4"
                 "Disk" "Diamond" "Cross" "Square"
                 "Wedge1" "Wedge2" "X"))))
      (send self :point-symbol
                 (iseq (send self :num-points))
                 (select
                    '(dot dot1 dot2 dot3 dot4
                      disk diamond cross square
                      wedge1 wedge2 x) i))
      (send self :redraw)))
(send (send plot1 :menu) :append-items change-item)
```