Some New Developments in Lisp-Stat

Luke Tierney
School of Statistics
University of Minnesota
Minneapolis, MN 55455
luke@umnstat.stat.umn.edu

Introduction

Lisp-Stat is a statistical computing environment based on the Lisp language (a subset of Common Lisp).

Lisp-Stat includes

- vectorized arithmetic
- basic statistics and linear algebra functions
- a window system interface
- access to user interface tools
- dynamic graphics tools
- an object-oriented programming system

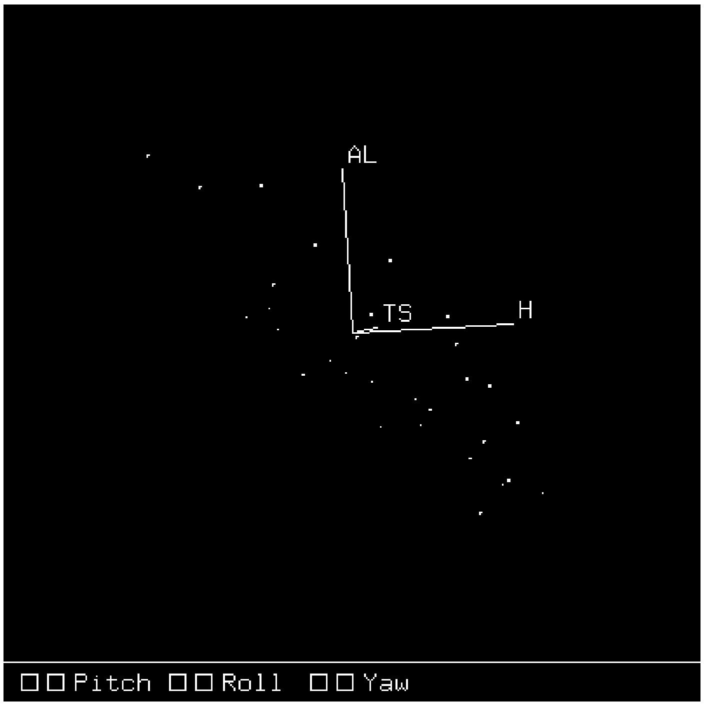
Basic Use of the System

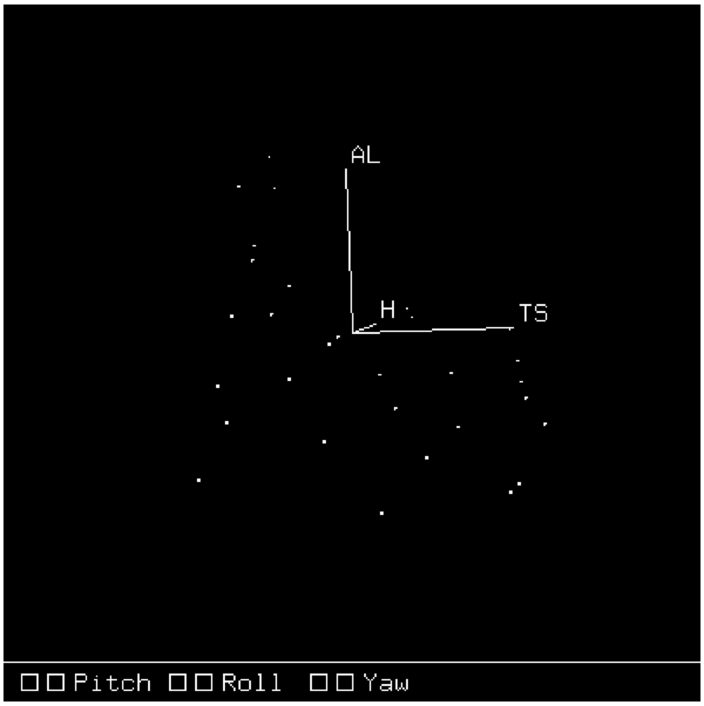
```
> (+ 1 2)
3

> (def abrasion-loss (list 372 206 ...))
ABRASION-LOSS
> (mean abrasion-loss)
175.4667
> (standard-deviation abrasion-loss)
88.12755
> (median abrasion-loss)
165

> (log abrasion-loss)
(5.918894 5.327876 5.164786 ...)
> (- abrasion-loss (mean abrasion-loss))
(196.5333 30.53333 -0.4666667 ...)
```

Basic Graphics

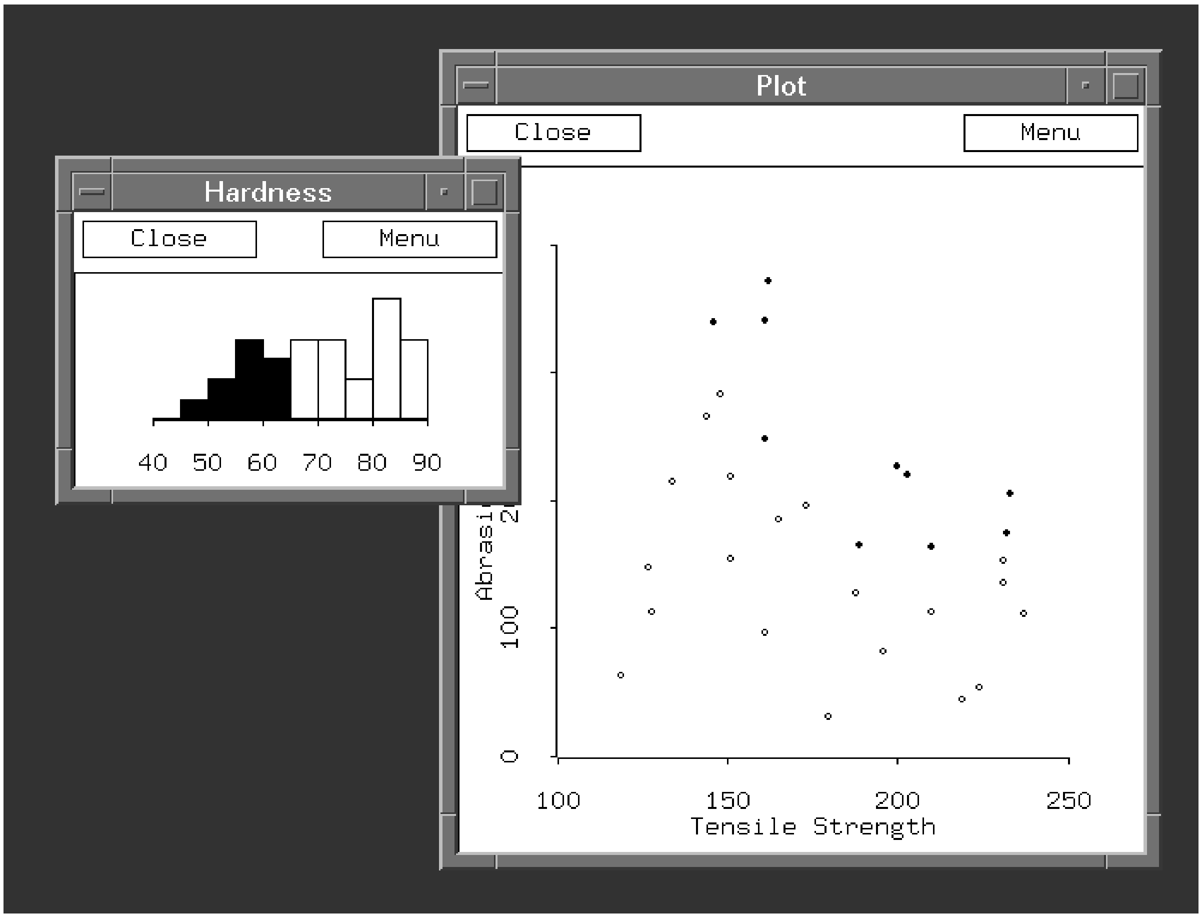
```
> (spin-plot
   (list tensile-strength
         hardness
         abrasion-loss)
   :variable-labels (list "T" "H" "A"))
#<OBJECT: ...>


> (histogram hardness :title "Hardness")
#<OBJECT: ...>


> (plot-points
   tensile-strength abrasion-loss
   :Variable-labels (list "T" "A"))
#<OBJECT...>
```

Rotating Plots





Linked Plots



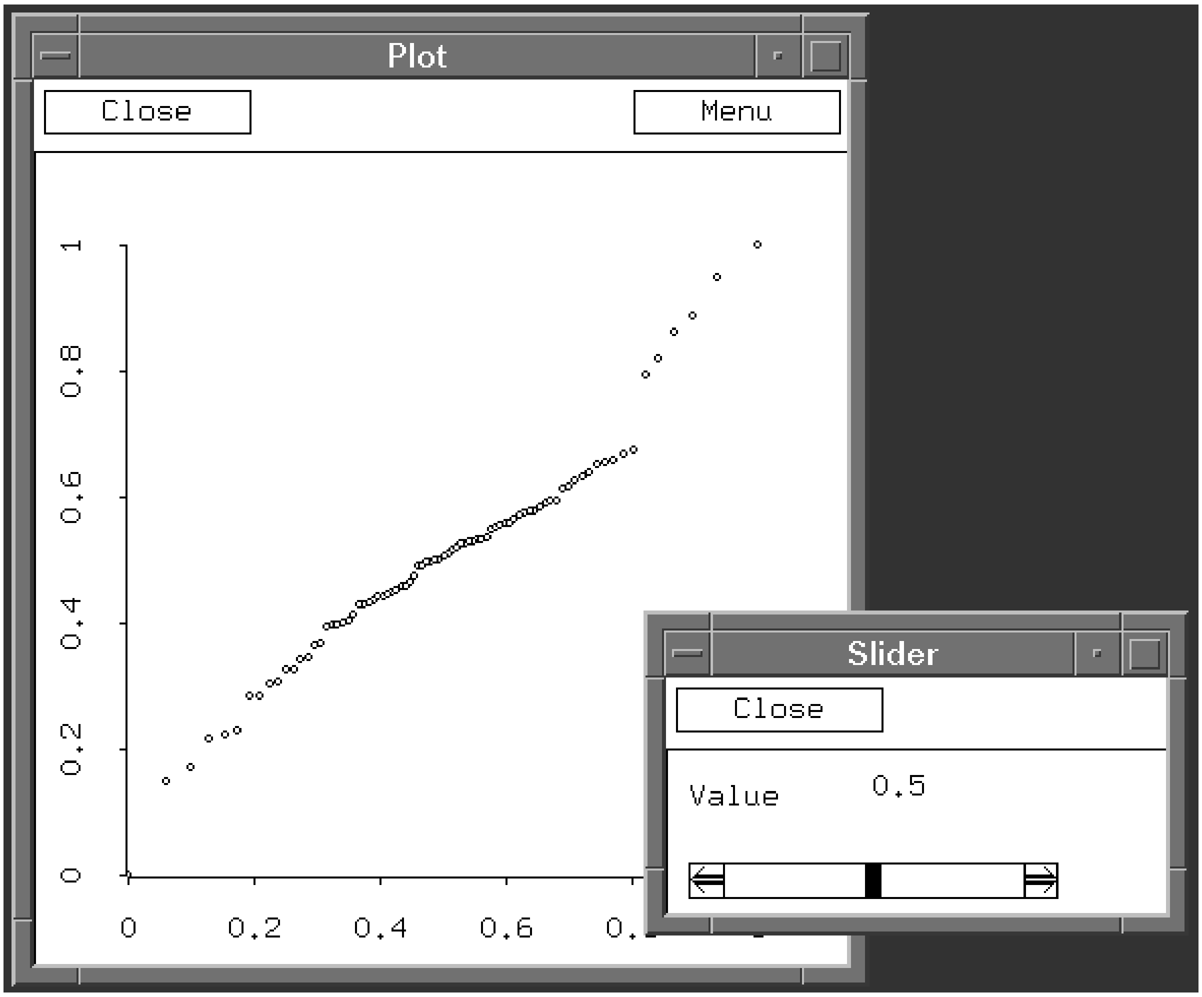
Modeling Tools

Several tools are available for fitting statistical models:

- liner regression models
- generalized linear models
- non-linear regression models
- maximizing functions
- approximating posterior distributions

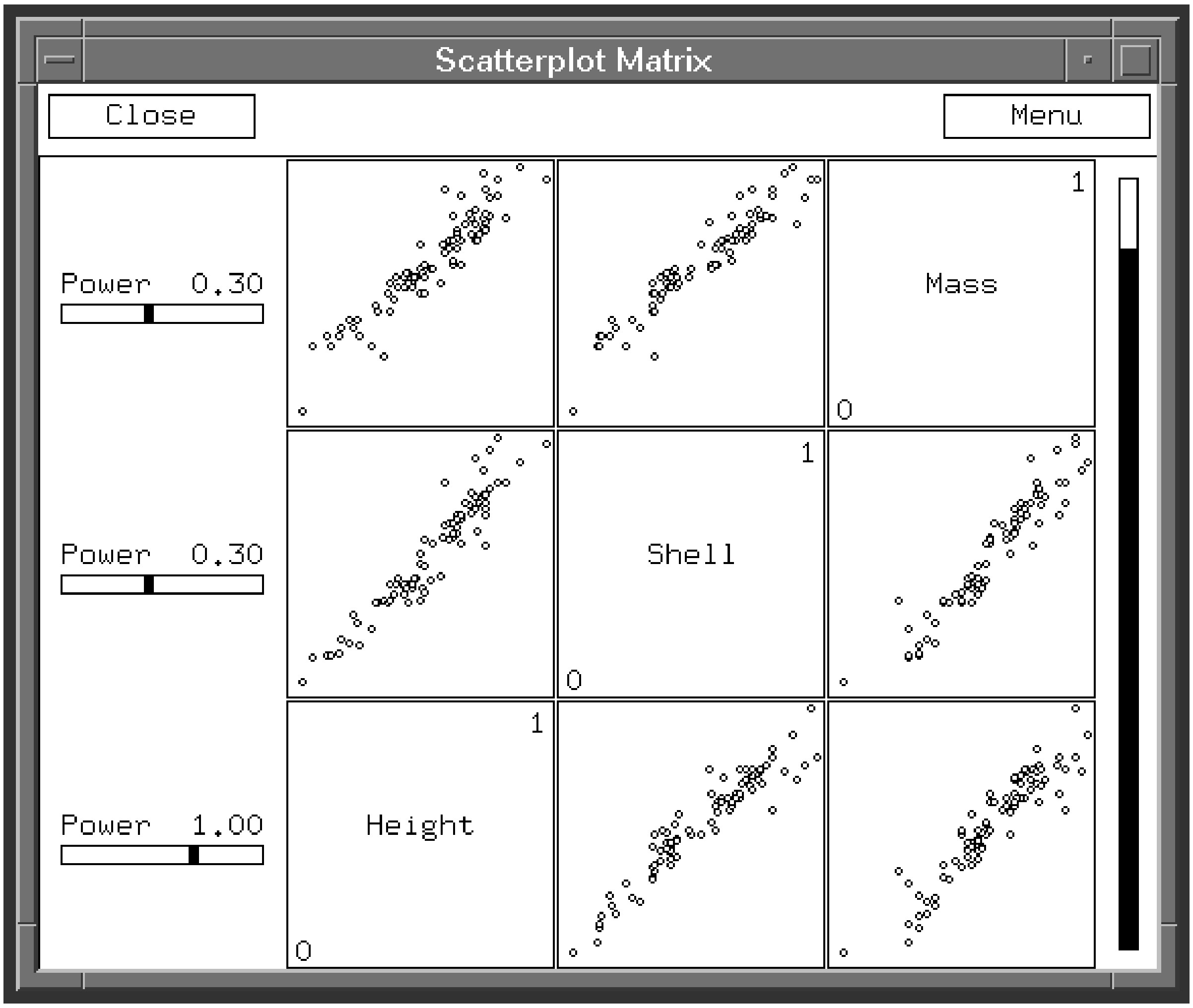
Adapting Dynamic Graphs

Dynamic Power Transformation

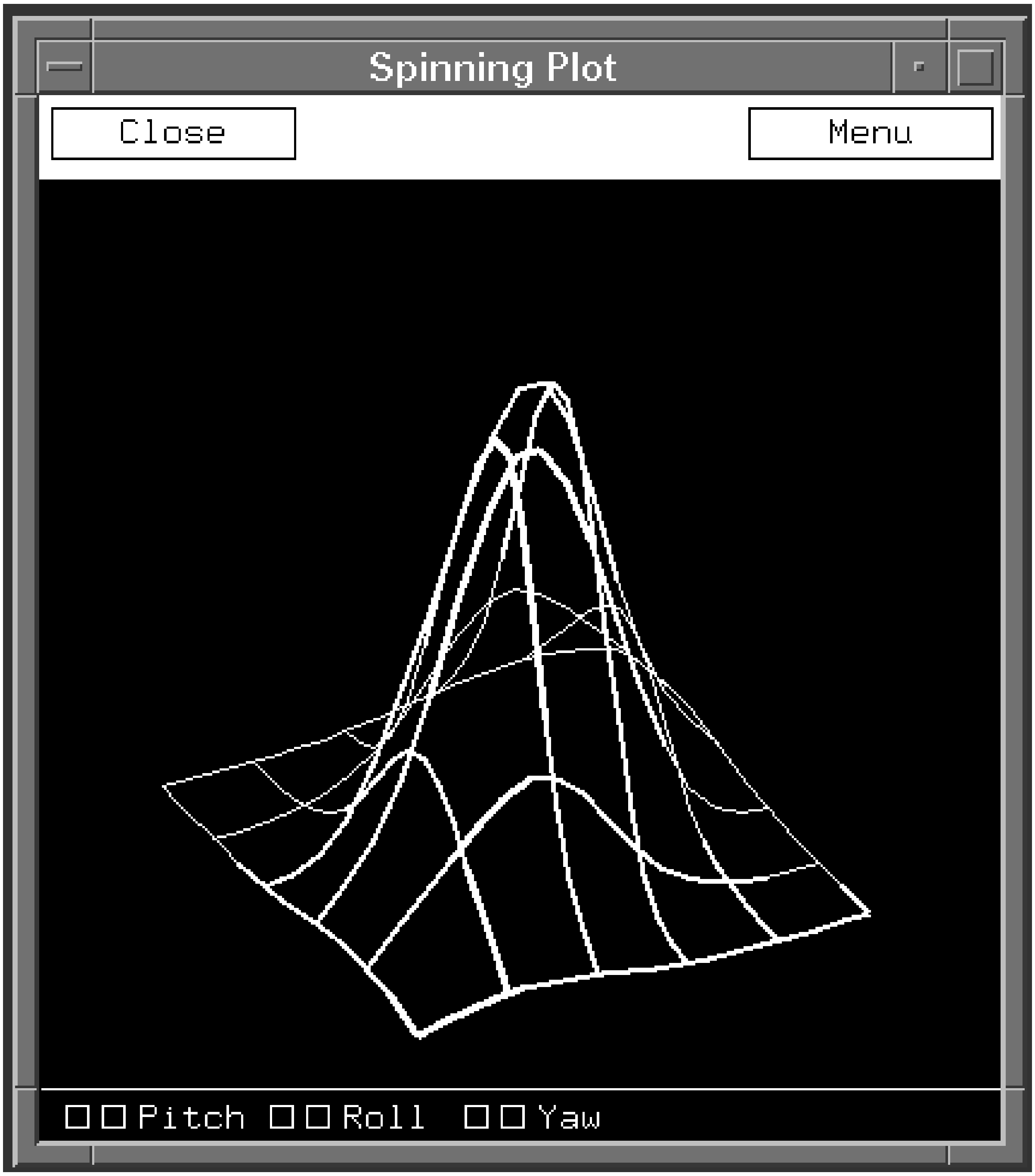


```
(defun bcr (x p)
  (if (< (abs p) 0.0001) (log x) (/ (^ x p) p)))

(defun sc (x)
  (let ((min (min x))
        (max (max x)))
    (/ (- x min) (- max min))))

(setf mussels (regression-model (list height shell) mass
                                :print nil))

(defmeth mussels :simple-bc ()
  (let* ((n (send self :num-cases))
         (nqr (sc (normal-quant (/ (iseq 1 n) (+ 1 n)))))
         (idx (iseq n))
         (r (send self :residuals))
         (p (plot-points (select nqr (rank r)) (sc r))))
    (defmeth p :change-power (pow)
      (send mussels :y (bcr mass pow))
      (let* ((r (send mussels :residuals))
             (r-nqr (select nqr (rank r))))
        (send self :point-coordinate 0 idx r-nqr)
        (send self :point-coordinate 1 idx (sc r))
        (send self :redraw-content)
        (pause 2)))
    (let ((s (interval-slider-dialog
              '(-1 2)
              :action #'(lambda (pow)
                          (send p :change-power pow)))))
      (send s :value 1)
      (send p :add-subordinate s))))
```

Variations and Elaborations



Hand Rotation



```
;; add a new "mouse mode", with menu title, cursor and
;; mouse method name
(send spin-proto :add-mouse-mode 'hand-rotate
      :title "Hand Rotate"
      :cursor 'hand
      :click :do-hand-rotate)

;; set up local environment with function to project
;; (x, y) point onto "globe" over the plot
(flet ((calcsphere (x y)
        (let* ((norm-2 (+ (* x x) (* y y)))
               (rad-2 (^ 1.7 2))
               (z (if (< norm-2 rad-2) (sqrt (- rad-2 norm-2)) 0)))
          (if (< norm-2 rad-2)
              (list x y z)
              (let ((r (sqrt (max norm-2 rad-2))))
                (list (/ x r) (/ y r) (/ z r)))))))

  ;; define the :DO-HAND-ROTATE method in the local environment
  (defmeth spin-proto :do-hand-rotate (x y m1 m2)
    (let* ((oldp (apply #'calcsphere
                        (send self :canvas-to-scaled x y)))
           (p oldp)
           (vars (send self :content-variables))
           (trans (identity-matrix (send self :num-variables))))
      (send self :idle-on nil)
      (send self :while-button-down
            #'(lambda (x y)
                (setf oldp p)
                (setf p (apply #'calcsphere
                               (send self :canvas-to-scaled x y)))
                (setf (select trans vars vars) (make-rotation oldp p))
                (when m1
                      (send self :slot-value 'rotation-type trans)
                      (send self :idle-on t))
                (send self
                      :apply-transformation
                      trans))))))
```

Additions Since First Release

- special variables
- multiple return values
- packages (name spaces)
- byte code compiler
- new random number generators
- ratios and unlimited precision integer arithmetic
- new garbage collector
- conditions

Byte Code Compiler

Converts list structure into a stream of bytes; also simplifies code:

- eliminates variable lookup
- eliminates unused features
- inlining
- tail recursive

An example:

```
> (defun f (x)
    (let ((s 0))
      (dolist (y x s)
        (setf s (+ s y)))))
F
> (def x (iseq 1 1000))
X
> (time (dotimes (i 10) (f x)))
The evaluation took 2.10 seconds
NIL
> (compile 'f)
F
> (time (dotimes (i 10) (f x)))
The evaluation took 0.20 seconds
```

Optimized native C code can be about another 100 times faster.

Assembly language for the example:

```
((#:MAIN
 (%INITIALIZE 0 2 1 0 3)
 (%COPY 2 3)
 (%COPY 1 4)
 (%CAR 4 5)                  ;(car x) -> y
 (%GOTO #:TEST)
 #:LOOP
 (%ARITH2 43 3 5 3)          ;(+ s y) -> s
 (%CDR 4 4)                  ;(cdr x) -> x
 (%CAR 4 5)                  ;(car x) -> y
 #:TEST
 (%TEST-1 1 #:LOOP #:ELSE 4)
 #:ELSE130132
 (%SET-ONE-VALUE-RETURN 0 3))
```

This code is translated to raw byte codes.

It could also be translated to C.

Packages

A file using a package:

```
(defpackage "MYPACKAGE" (:use "CL"))
(in-package "MYPACKAGE")

(export '(h k))
(defun g (x) (+ x 1))
(defun h (x) (+ (g x) 2))
(defun k (x) (h x))
```

Using the package:

```
> (mypackage:h 1)
4
> (mypackage:g 1)
Error: external symbol not found - "G"
> (mypackage::g 1)
2
```

Random Number Generators

Three generators are available:

0 The original lagged Fibonacci generator.

1 L'Ecuyer's version of Wichmann-Hill generator.

2 Marsaglia's Super-Duper, as used in S.

3 Combined Tausworthe generator. Tezuka and L'Ecuyer (1991).

The default generator is generator 1.

All three new generators have periods on the order of 2^{60}.

Random states are now printed as

```
> *random-state*
#$(1 #(2147483562 0 11716 54063))
```

Making new random states:

```
> (make-random-state t)
#$(1 #(2147483562 0 11716 54088))
> (make-random-state t 2)
#$(2 #(2147483647 0 0 11715 0 54105))
```

Recent Developments

- BLAS
- content backgrounds
- missing data in graphs
- shared libraries
- C code wrappers
- improved autoloading
- finalization

BLAS

Some examples:

```
> (def x (coerce #(1 2 3) '(vector c-double)))
X
> (def y (copy-vector x))
Y
> (blas-ddot 3 x 0 1 x 0 1)
14.0
> (blas-daxpy 3 4 x 0 1 y 0 1)
NIL
> y
#(5.0 10.0 15.0)
```

Most of level 1 and some of levels 2 and 3 are supported.

The interface is still experimental.

Shared Libraries

- Low level
  - interface to `dlopen`, `dlsym`
  - calling `void *f(void *, ...)` functions
  - calling internal Lisp functions

  Allows emulation of old `call-cfun`.

- Higher level
  - versioning
  - simple initialization
  - reflection

  Basic support for a rudimentary shared module system.

C Code Wrappers

- automates generating interfaces to C libraries
- builds on shared library system
- based in part on SWIG system

A simple wrapper file:

```
(in-package "POSIX")
(wrap:c-lines "#include <unistd.h>")
(wrap:c-function getuid "getuid" () :integer)
(wrap:c-function getlogin "getlogin" () :string)
```

Using the functions:

```
> (posix:getuid)
100
> (posix:getlogin)
"luke"
```

A more complex example:

```
(in-package "POSIX")
(wrap:c-lines "#include <pwd.h>")
(wrap:c-pointer (:struct "passwd")
                (:get pw-name "pw_name" :string)
                (:get pw-uid "pw_uid" :integer)
                (:get pw-gid "pw_gid" :integer)
                (:get pw-dir "pw_dir" :string)
                (:get pw-shell "pw_shell" :string))
(defun pw-values (pw)
  (when pw
        (values  (pw-name pw)
                 (pw-uid pw)
                 (pw-gid pw)
                 (pw-dir pw)
                 (pw-shell pw))))
(defun getpwnam (name)
  (pw-values (base-getpwnam name)))
(defun getpwuid (uid)
  (pw-values (base-getpwuid uid)))
```

and its use:

```
> (posix:getpwnam "fred")
NIL
> (posix:getpwnam "luke")
"luke"
100
1
"/NOKOMIS/users/luke"
"/usr/local/bin/tcsh"
> (posix:getpwuid 100)
"luke"
100
1
"/NOKOMIS/users/luke"
"/usr/local/bin/tcsh"
```

Some open issues:

- callback functions
- automatic makefile/project creation
- FORTRAN, C++ wrapping

Content Backgrounds

Modification to redraw protocol:

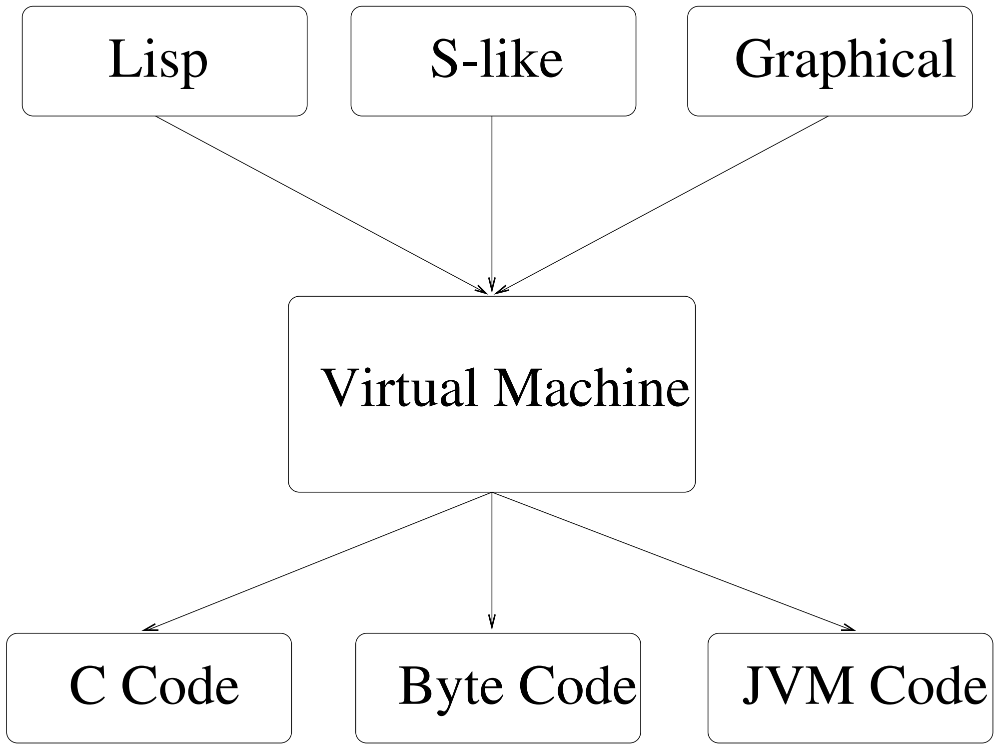
- `:clear-content` is called before drawing
- default method just erases
- can use to insert a background image

```
(setf p (plot-points (normal-rand '(10 10))))

(defmeth p :clear-content ()
  (call-next-method)
  (let ((col (send self :draw-color)))
    (send self :draw-color 'red)
    (send self :paint-rect 50 50 100 100)
    (send self :draw-color col)))
```

New Directions

- new virtual machine and runtime
- using/working with Java
- multithreading
- distributed computing
- revised graphics model
- alternate languages

<u>New Virtual Machine Design</u>



- multiple implementations
  - C byte code interpreter
  - native code from C
  - Java byte code
- language-independent core
  - Lisp syntax
  - S-like syntax
  - graphical syntax

Multithreading

- user-level multithreading
- multiple OS threads, processors
- CSP-style channels as main synchronization approach
- low level synchronization (mutexes/semaphores) if needed

Revised Graphics Model

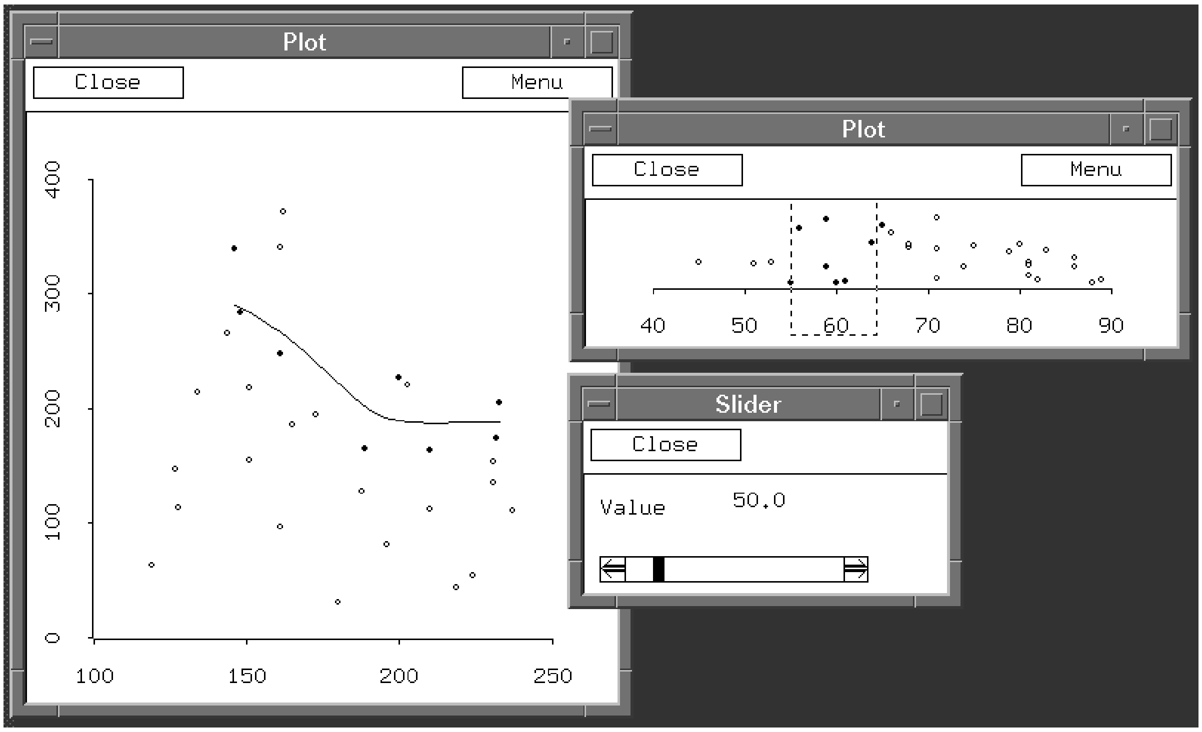
Simpler, lower level:

- hierarchical menus
- hierarchical/nested windows
- geometry managers
- native widgets
- construction tools

Higher level:

- new high-level model for graph organization
- graph customization by direct manipulation

An Example: Enhanced Linking



Start with a linked histogram and scatterplot of the abrasion loss data:

```
(setf h (histogram hardness))
(setf p (plot-points tensile-strength abrasion-loss))

(send h :linked t)
(send p :linked t)
```

Next add kernel smoothing controls:

```
(send p :add-slot 'kernel-width 50)

(defmeth p :kernel-width (&optional width)
  (when width
        (setf (slot-value 'kernel-width) width)
        (send self :adjust-screen))
  (slot-value 'kernel-width))


(send p :add-subordinate
      (interval-slider-dialog
       '(20 100)
       :action #'(lambda (w) (send p :kernel-width w)))))
```

Finally, add a new `:adjust-screen` method to add a smooth for the highlighted or selected points:

```
(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((i (union (send self :points-selected)
                  (send self :points-hilited))))
    (send self :clear-lines :draw nil)
    (if (< 1 (length i))
        (let ((x (select tensile-strength i))
              (y (select abrasion-loss i))
              (w (send self :kernel-width)))
          (send self :add-lines (kernel-smooth x y :width w)))
        (send self :redraw-content))))
```

Some Systems using Lisp-Stat

Some systems developed using Lisp-Stat include

- *R-code* – R. D. Cook and S. Weisberg

  `http://www.stat.umn.edu/~rcode/index.html`

- *ViSta* – F. W. Young

  `http://forrest.psych.unc.edu/research/ViSta.html`

- *StatObjects and StatInsight* – E. J. Harner

  `http://www.stat.wvu.edu/SRS/`

Some References

The XLISP-STAT home page is

`www.stat.umn.edu/~luke/xls/xls.html`

Recent developments are described on

`www.stat.umn.edu/~luke/xls/projects/projects.html`

Some `majordomo`-managed mailing lists at `stat.umn.edu`:

`stat-lisp-news`	general discussion
`stat-lisp-devel`	development issues

Book is available form Wiley.