

LISP-STAT

**AN OBJECT-ORIENTED ENVIRONMENT
FOR STATISTICAL COMPUTING
AND DYNAMIC GRAPHICS**

Edition

LUKE TIERNEY

**WILEY SERIES IN PROBABILITY
AND MATHEMATICAL STATISTICS**

LISP-STAT

LISP-STAT

**An Object-Oriented Environment
for Statistical Computing
and Dynamic Graphics**

LUKE TIERNEY

*School of Statistics
University of Minnesota
Minneapolis, Minnesota*



A Wiley-Interscience Publication

JOHN WILEY & SONS

New York / Chichester / Brisbane / Toronto / Singapore

Macintosh is a trademark of Macintosh Laboratory, Inc., licensed to Apple Computer Inc.
PostScript is a trademark of Adobe Systems Inc.
SunView is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of AT&T.
X Window System is a trademark of the Massachusetts Institute of Technology.

In recognition of the importance of preserving what has been written, it is a policy of John Wiley & Sons, Inc. to have books of enduring value published in the United States printed on acid-free paper, and we exert our best efforts to that end.

Copyright © 1990 by John Wiley & Sons

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by Section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging in Publication Data:

Tierney, Luke.

Lisp-Stat: an object-oriented environment for statistical computing and dynamic graphics / Luke Tierney.

p. cm. — (Wiley series in probability and mathematical statistics. Applied probability and statistics, ISSN 0271-6232)

"A Wiley Interscience publication."

Includes bibliographical references.

ISBN 0-471-50916-7

1. Lisp-Stat (Computer program) 2. XLisp-Stat (Computer program)

3. Mathematical statistics — Data processing. 4. LISP (Computer program language) 5. Object-oriented programming (Computer science)

I. Title. II. Series

QA276.4.T54 1990

519.502855369-dc20

90-39614

CIP

Contents

Preface	xi
1 Introduction	1
1.1 Environments for Statistical Computing	1
1.2 The Lisp-Stat Environment	3
1.2.1 Why Lisp?	3
1.2.2 Using Lisp-Stat	4
1.2.3 Some Design and Portability Issues	6
1.2.4 The Future of Lisp-Stat	7
2 A Lisp-Stat Tutorial	9
2.1 The Lisp Interpreter	9
2.2 Elementary Computations and Graphs	13
2.2.1 One-Dimensional Summaries and Plots	13
2.2.2 Two-Dimensional Plots	19
2.2.3 Plotting Functions	22
2.3 More on the Interpreter	23
2.3.1 Saving Your Work	23
2.3.2 A Command History Mechanism	24
2.3.3 Getting Help	25
2.3.4 Listing and Undefining Variables	28
2.3.5 Interrupting a Calculation	29
2.4 Some Data-Handling Functions	29
2.4.1 Generating Systematic Data	29
2.4.2 Generating Random Data	31
2.4.3 Forming Subsets and Deleting Cases	32
2.4.4 Combining Several Lists	34
2.4.5 Modifying Data	34
2.4.6 Reading Data Files	36
2.5 Dynamic Graphs	37
2.5.1 Spinning Plots	37
2.5.2 Scatterplot Matrices	41
2.5.3 Interacting with Individual Plots	45

2.5.4	Linked Plots	46
2.5.5	Modifying a Scatterplot	47
2.5.6	Dynamic Simulations	50
2.6	Regression	53
2.7	Defining Functions and Methods	58
2.7.1	Defining Functions	58
2.7.2	Functions as Arguments	59
2.7.3	Graphical Animation Control	60
2.7.4	Defining Methods	62
2.8	More Models and Techniques	64
2.8.1	Nonlinear Regression	64
2.8.2	Maximization and Maximum Likelihood Estimation	67
2.8.3	Approximate Bayesian Computations	70
3	Programming in Lisp	77
3.1	Writing Simple Functions	78
3.2	Predicates and Logical Expressions	79
3.3	Conditional Evaluation	80
3.4	Iteration and Recursion	82
3.5	Environments	86
3.5.1	Some Terminology	86
3.5.2	Local Variables	88
3.5.3	Local Functions	91
3.6	Functions and Expressions as Data	92
3.6.1	Anonymous Functions	92
3.6.2	Using Function Arguments	93
3.6.3	Returning Functions as Results	95
3.6.4	Expressions as Data	96
3.7	Mapping	98
3.8	Assignment and Destructive Modification	100
3.9	Equality	104
3.10	Some Examples	106
3.10.1	Newton's Method for Finding Roots	106
3.10.2	Symbolic Differentiation	109
4	Additional Lisp Features	119
4.1	Input/Output	119
4.1.1	The Lisp Reader	119
4.1.2	Basic Printing Functions	120
4.1.3	Format	121
4.1.4	Files and Streams	123
4.2	Defining More Flexible Functions	126
4.2.1	Keyword Arguments	126
4.2.2	Optional Arguments	127
4.2.3	Variable Number of Arguments	128

4.3	Control Structure	129
4.3.1	Conditional Evaluation	129
4.3.2	Looping	130
4.4	Basic Lisp Data and Functions	132
4.4.1	Numbers	132
4.4.2	Strings and Characters	134
4.4.3	Symbols	135
4.4.4	Lists	137
4.4.5	Vectors	140
4.4.6	Sequences	141
4.4.7	Arrays	144
4.4.8	Other Data Types	147
4.5	Odds and Ends	147
4.5.1	Errors	148
4.5.2	Code-Writing Support	149
4.5.3	Debugging Tools	151
4.5.4	Timing	154
4.5.5	Defsetf	154
4.5.6	Special Variables	155
5	Statistical Functions	157
5.1	Compound Data	157
5.1.1	Compound Data Properties	157
5.1.2	Vectorized Arithmetic	158
5.2	Data-Handling Functions	159
5.2.1	Basic Operations	159
5.2.2	Sorting Functions	160
5.2.3	Interpolation and Smoothing	161
5.3	Probability Distributions	162
5.4	Array and Linear Algebra Functions	164
5.4.1	Basic Matrix and Array Functions	164
5.4.2	Matrix Multiplication	166
5.4.3	Linear Algebra Functions	167
5.5	Regression Computation Functions	170
5.6	Some Examples	171
5.6.1	Constructing a Projection Operator	171
5.6.2	Robust Regression	173
6	Object-Oriented Programming	179
6.1	Some Motivation	179
6.2	Objects and Messages	180
6.2.1	Basic Structure and Terminology	180
6.2.2	Constructing New Objects	182
6.2.3	Defining New Methods	184
6.2.4	Printing Objects	186

6.3	Prototypes and Inheritance	187
6.3.1	Constructing Prototypes and Instances	187
6.3.2	Prototypes Inheriting from Prototypes	189
6.3.3	Overriding and Modifying Inherited Methods	191
6.4	Additional Details	193
6.4.1	Creating Objects and Prototypes	193
6.4.2	Additional Methods and Functions	194
6.4.3	Shared Slots	195
6.4.4	The Object Documentation System	195
6.4.5	Saving Objects	197
6.5	Some Built-in Prototypes	197
6.5.1	Compound Data Objects	199
6.5.2	Linear Regression Models	201
6.5.3	Nonlinear Regression Models	202
6.6	Multiple Inheritance	203
6.7	Other Object Systems	205
6.8	Some Examples	207
6.8.1	A Rectangular Data Set	207
6.8.2	An Alternate Data Representation	209
7	Windows, Menus, and Dialogs	213
7.1	The Window System Interface	213
7.2	Windows	215
7.3	Menus	218
7.4	Dialog Windows	222
7.4.1	Some Standard Modal Dialogs	222
7.4.2	Modeless Slider Dialogs	225
7.5	Constructing Custom Dialogs	226
7.5.1	The Dialog Prototypes	226
7.5.2	Dialog Items	228
7.6	Additional Details	233
8	Graphics Windows	235
8.1	The Graphical Model	235
8.1.1	Coordinates, Drawing Modes, and Colors	235
8.1.2	Drawable Objects	237
8.2	Graphical Messages	239
8.2.1	Initialization and State Messages	239
8.2.2	Basic Drawing Messages	240
8.2.3	Additional Messages	242
8.3	Double Buffering and Animation	244
8.4	Responding to Events	246
8.4.1	Resize and Exposure Events	247
8.4.2	Mouse Events	248
8.4.3	Key Events	251

8.4.4	Idle Actions	252
8.4.5	Menus	253
8.5	Additional Features	254
8.5.1	Canvas Dimensions	254
8.5.2	Clipping	255
8.5.3	Saving Graphs	255
8.5.4	Adding New Colors	256
8.5.5	Adding New Cursors	256
8.6	An Example	257
9	Statistical Graphics Windows	261
9.1	The Graph Prototype	261
9.1.1	Constructing a New Graph	262
9.1.2	Adding Data and Axes	263
9.1.3	Scaling and Transformations	269
9.1.4	Mouse Events and Mouse Modes	274
9.1.5	Linking	280
9.1.6	Window Layout, Resizing, and Redrawing	283
9.1.7	Plot Overlays	286
9.1.8	Menus and Menu Items	290
9.1.9	Miscellaneous Messages	292
9.2	Some Standard Graph Prototypes	292
9.2.1	Scatterplots	292
9.2.2	Scatterplot Matrices	293
9.2.3	Rotating Plots	293
9.2.4	Histograms	294
9.2.5	Name Lists	295
10	Some Dynamic Graphics Examples	297
10.1	Some Animations	297
10.1.1	Another Look at Power Transformations	297
10.1.2	Plot Interpolation	300
10.1.3	Choosing a Smoothing Parameter	304
10.2	Using New Mouse Modes	306
10.2.1	Sensitivity in Simple Linear Regression	306
10.2.2	Hand Rotation	309
10.2.3	Graphical Function Input	310
10.3	Plot Control Overlays	313
10.3.1	A Press Button Control	313
10.3.2	Two-Button Controls	317
10.4	Grand Tours	321
10.5	Parallel Coordinate Plots	329
10.6	An Alternative Linking Strategy	334
Bibliography		341

A Answers to Selected Exercises	347
B The XLISP-STAT Implementation	355
B.1 Obtaining the XLISP-STAT Software	356
B.2 XLISP-STAT on the Macintosh	356
B.2.1 Basics	356
B.2.2 Menus	357
B.2.3 Windows	358
B.2.4 More Advanced Features	361
B.3 XLISP-STAT on UNIX Systems	363
B.3.1 XLISP-STAT under <i>X11</i>	364
B.3.2 XLISP-STAT under <i>SunView</i>	365
B.3.3 Running UNIX Commands from XLISP-STAT	366
B.3.4 Dynamic Loading and Foreign Function Calling	366
B.4 Some Additional Features	368
Index	371

Preface

This book describes a statistical environment called Lisp-Stat. As its name suggests, Lisp-Stat is based on the Lisp language. It includes support for vectorized arithmetic operations, a comprehensive set of basic statistical operations, an object-oriented programming system, and support for dynamic graphics.

The primary objective of this book is to introduce the Lisp-Stat system and show how it can be used as an effective platform for a large number of statistical computing tasks, ranging from basic calculations to customizing dynamic graphs. A further objective is to introduce object-oriented programming and graphics programming in a statistical context. The discussion of these ideas is based on the Lisp-Stat system, so readers with access to such a system can reproduce the examples presented here and use them as a basis for further experimentation. But the issues presented are more general and should apply to other environments as well.

This book can be used as a supplement to several courses on statistical computing and computational statistics. A course emphasizing the use of different programming paradigms could be based on the material in Chapters 3-6. A course on dynamic graphics could use primarily the material in Chapters 6-10.

Prerequisites

The statistical background assumed for most of the book is a solid undergraduate sequence in statistical theory and methods, including basic regression analysis. A few sections assume somewhat more; these sections indicate the background they require.

No prior knowledge of Lisp is assumed. Some experience with a programming language like FORTRAN or C, or with a statistical system like *S* [6,7], would be helpful but is not essential.

Notational Conventions

Throughout this book mathematical symbols and formulas are written in italics, like x and $x + y$. Lisp symbols and expressions are written in a

typewriter font, like `x` and `(+ x y)`. Interactions with the computer are shown as

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

The prompt `>` is typed by the computer. The expression `(+ 1 2)` and the *return* at the end of the line are entered by the user, and the answer `3` is then typed by the computer.

Several features described in this book are available in any Lisp system, some are peculiar to particular Lisp dialects like Common Lisp, and some are only available in a Lisp-Stat system. I have attempted to identify these situations consistently by describing a feature as a Lisp, Common Lisp, or Lisp-Stat feature, respectively.

Software Availability

As with any book on computing, you will benefit most from reading this book if you can try out and experiment with the examples. Most sections also contain some exercises to try on a Lisp-Stat system.

At the time of writing, there is one implementation of Lisp-Stat, called XLISP-STAT. This implementation is based on the XLISP dialect of Lisp, a subset of Common Lisp developed by David Betz [10,11]. Versions of XLISP-STAT are available for the Apple Macintosh, for Sun workstations, and for many workstations running the UNIX operating system and the *X11* window system. A nongraphics version for generic UNIX systems is available as well. Further details on obtaining the XLISP-STAT system are given in Appendix B.1.

Version 2.1 of XLISP-STAT corresponds to the description in this book. Earlier versions may deviate from the description in some respects.

Illustrations reproduced in this book are taken from the Macintosh version of XLISP-STAT.

Acknowledgments

Many of the functions included in Lisp-Stat were motivated by similar functions in the *S* system [6,7]. Several of the examples used in this book have been taken from the introductory statistics text by Devore and Peck [26]. A number of programming examples have been adapted from examples in Abelson and Sussman [1]. Several ideas in the tutorial chapter were borrowed from Gary Oehlert's *MacAnova User's Guide* [47]. The XLISP-STAT help system is patterned after the help system provided by Kyoto Common Lisp.

Work on Lisp-Stat was supported in part by grants from the MinneMac Project at the University of Minnesota, by a single quarter leave granted by the University of Minnesota, by grant DMS-8705646 from the National

Science Foundation, and by a research contract with Bell Communications Research. Much of the work on this book was completed during a visit to the Statistics and Economics Research Division at Bellcore in the fall of 1989. I would like to express my gratitude to Bellcore for their hospitality and support.

I would like to thank Doug Bates, Rick Becker, Andreas Buja, Kathryn Chaloner, John Chambers, Dennis Cook, Rich Faldowski, Jim Harner, David Harris, Jay Kadane, Jim Lindsey, Wei-Yin Loh, Sharon Lohr, Michael Meyer, Gary Oehlert, Ron Pruitt, Anne Steuer, Debbie Swayne, Paul Tukey, Sandy Weisberg, Allan Wilks, and Forrest Young for comments and suggestions that have lead to improvements in the Lisp-Stat system and in the presentation of this book. I am particularly grateful to Jim Harner for his careful reading of the manuscript.

Finally, I would like to thank David Betz for developing XLISP and making the source code publically available.

LUKE TIERNEY

Minneapolis, Minnesota

June 1990

LISP-STAT

Chapter 1

Introduction

1.1 Environments for Statistical Computing

In the early days, statistical computing was mostly concerned with calculating numbers. Computers were slow and had to be programmed carefully, at a fairly low level, to get reasonable performance. As a result, most early statistical systems were simple front ends to efficient but rigid numerical algorithms. Users of these systems would submit requests for calculations, and would receive their results later the same day or on the following day.

As computers have become faster and cheaper, it is now possible to do in seconds or minutes computations that used to take hours or days. This presents new challenges to statistical software. Since computing hardware is now able to perform many different calculations and analyses in a short period of time, a supportive statistical software environment has to make it easy to express different computations to the computer, to pass results of one computation on to another, to examine results, and to keep track of computations. Furthermore, as it becomes apparent that some series of computations is used repeatedly, perhaps with minor variations, it should be possible to build a simple abstraction for describing and requesting that series of computations as a unit. In short, to complement the capabilities of modern hardware, statistical environments must become interactive statistical languages.

Another important development has taken place in parallel to the increase in raw computational speed. More and more statistical computing is done on personal computers and workstations equipped with high-resolution bitmapped displays. This display hardware opens up many new opportunities. One of the most exciting areas of activity in statistical research is the field of dynamic computer graphics and its use in statistics. Graphical methods have been an important tool in statistics for many years. But an inherent limitation of traditional, two-dimensional graphics is that they cannot easily represent more than two-dimensional information. The ability of modern

computer displays to show a rapidly changing series of two-dimensional plots opens up a third dimension: time. Dynamically changing plots can take several forms, ranging from systematic changes in a rotating point cloud to changes determined interactively by brushing operations in which points highlighted in one plot are simultaneously highlighted in several other plots.

Many interesting ideas for dynamic graphical displays have been proposed in recent years [20]. But to assess the full potential of these ideas, and to determine fruitful directions for future development, the new methods need to be made available to a wide range of users and tested in a variety of situations. Most dynamic graphical techniques proposed to date have been implemented on specialized hardware, often using complex programming methods. As a result, very few statisticians have been able to explore the use of these methods in practice, and still fewer have had the opportunity to experiment with variations on these methods. To support progress in this area, a statistical environment needs to implement standard dynamic graphical methods such as point cloud rotation and also provide a set of basic dynamic graphics building blocks that can be used to implement new ideas easily and portably.

Recent experience has shown that a new programming paradigm, object-oriented programming, is very useful for graphics programming in general and statistical graphics programming in particular [58,45]. Object-oriented methods also have many other applications within a statistical system. They can be used for developing flexible data structures [44,58] and for representing statistical models [48].

Tools for graphics programming and object-oriented programming are only two of many features needed in a modern statistical environment. Others include the ability to handle “nonstandard” statistical data. Standard data are lists or collections of numbers. Most data sets can be represented by such collections. Within the context of linear models, different models can also be represented by collections of numbers: the model coefficients and the matrix of predictors. But once a more general class of models is considered, it becomes difficult, if not impossible, to find numerical representations of different models. For example, a system for fitting nonlinear regression models would be severely limited if it required that the nonlinear mean function be selected from some fixed set of families. In this case it is necessary to be able to specify the mean function in a more general way, either as a procedure for evaluating the mean response for different parameter values, or as an expression. Once a language permits the use of such non-numerical data many new opportunities are opened up. For example, it becomes possible to have programs examine model expressions, compute expressions for derivatives, or try to determine structure that leads to simplifications in computation or interpretation.

1.2 The Lisp-Stat Environment

The requirements outlined above suggest that a flexible, modern statistical environment must be embedded in a high-level, interactive programming language. Two approaches can be taken in developing such an environment. The first approach, taken by the *S* system [6,7], is to develop a new, high-level language from scratch. The second approach is to adapt an existing language for statistical computation. The language most often suggested is Lisp.

Lisp-Stat represents an attempt to develop a complete statistical environment based on the Lisp language. It consists of a Lisp system with modifications to standard Lisp functions to support vectorized arithmetic, additional functions for basic statistical computations, an interface to a window system, and tools for constructing graphs, in particular dynamic graphs, within this window system. Lisp-Stat also contains an object-oriented programming system that is used to support graphics programming and to represent statistical models, such as linear and nonlinear regression models.

Lisp-Stat is not a single system or implementation. Instead, it is a specification of a set of essential ingredients that can be implemented using a variety of Lisp system and window system combinations. At the time of writing, one Lisp-Stat implementation exists, a system called XLISP-STAT. This system is based on the XLISP dialect of Lisp, and can be used with the Macintosh or the *X11* window systems. Over time, other implementations may become available. Versions of XLISP-STAT for use in several other window systems are currently being developed. A Common Lisp version of Lisp-Stat is also under development.

1.2.1 Why Lisp?

Lisp is a very rich general purpose programming language that is ideally suited for interactive, experimental programming. It allows the use of functions as data, and it is capable of supporting a number of different programming paradigms, including object-oriented programming. These points have lead to the use of Lisp as a tool for introducing basic programming concepts in computer science classes [1], and have lead a number of authors to use and recommend Lisp as an ideal base environment for statistical computing [45,48,58].

Lisp is actually quite an old language. It is the second oldest language in use today, second only to FORTRAN. Over the years a number of different Lisp dialects have developed, but since the early 1980s the Lisp community has been converging towards using a single dialect, Common Lisp [56]. Many excellent Common Lisp systems are now available, and code produced by Common Lisp compilers is competitive in speed with compiled C and FORTRAN code. A Lisp-Stat system is assumed to be based on the Common Lisp dialect, or a reasonable subset of Common Lisp.

All aspects of the Lisp language that are used in this book are described

here, primarily in Chapters 3 and 4. But there are many other features of the language that are not described or are only mentioned briefly. If you would like to explore the Lisp language further, there are several excellent books available. The definition of the Common Lisp language is given by Steele [56]. A complete documentation of all Common Lisp functions is contained in [31]. Two good introductions to the Common Lisp language are Winston and Horn [67] and Tater [59].

Several other languages have been suggested and used as the basis for statistical environments. A language that has received considerable attention is APL [3]. APL has many useful features, including a wide range of functions for handling matrices and arrays. But it does not have the ability to easily handle high-level data, such as functions or expressions, nor does it lend itself readily to supporting the object-oriented programming style that is so important for graphical programming. Adding these features to APL appears to be considerably harder than adding matrix and array functions to Lisp.

1.2.2 Using Lisp-Stat

A Lisp-Stat system can be used at a number of different levels, ranging from a basic statistical calculator and plotter to a platform for designing graphical user interfaces for statistics. The organization of this book is based on the natural progression through these levels.

Chapter 2 describes how to use Lisp-Stat as a statistical calculator and plotter. Much can be accomplished by using Lisp-Stat at this level. You can enter data, compute summary statistics, produce static and dynamic plots, and fit regression models. The commands used for these calculations, such as

`(mean precipitation)`

and

`(standard-deviation precipitation)`

to calculate the mean and standard deviation of a data set `precipitation`, happen to be Lisp expressions. But this does not mean you have to learn a large amount of Lisp before you can start to use Lisp-Stat. By learning a few basic commands, much like the basic commands in any other statistical system, you can start to analyze data in a matter of minutes.

But there are a number of benefits that derive from the fact that commands in Lisp-Stat are Lisp expressions. Even though you do not need to know Lisp to start using Lisp-Stat, as you learn to use the system you are learning the basics of Lisp. Having learned how to calculate means and standard deviations, it is a small step to calculating a standardized variable by an expression of the form

`(/ (- precipitation (mean precipitation))
 (standard-deviation precipitation))`

If you find that you often want to standardize a variable, then it is again a small step to encapsulating this process into a function defined by

```
(defun standardize (x)
  (/ (- x (mean x))
      (standard-deviation x)))
```

The transition from using Lisp-Stat as a calculator to customizing the system by writing your own functions is very natural and is outlined in the final sections of Chapter 2.

The tools described in Chapter 2 are already quite powerful. But you can do much more than simply combine a few operation into a new function. Using the Lisp programming language, you can implement new algorithms and design functions for fitting new classes of models. To take full advantage of these capabilities, you need to learn about the basic control structures and functions provided by Lisp and Lisp-Stat. Chapter 3 introduces control structures, such as conditional evaluation constructs, and also describes some of the programming styles that are frequently used in Lisp programming. Chapter 4 presents a number of additional standard Lisp functions, and Chapter 5 describes the statistical functions included in Lisp-Stat.

Lisp-Stat plots and regression models are implemented using an object-oriented programming system. Chapter 2 gives a brief introduction to the use of objects, and Chapter 6 presents a more detailed description of both the Lisp-Stat object system and the ideas behind the object-oriented paradigm. The Lisp-Stat object system is the foundation of the Lisp-Stat window system described in Chapter 7. In addition to forming the basis of the graphics system, the window system provides access from within Lisp-Stat to graphical user interface tools such as menus and dialogs. The graphics system consists of two additional layers built on top of the window system. The first layer, described in Chapter 8, contains tools needed to draw images on the screen. The second layer, presented in Chapter 9, contains support for managing plot data and standard interaction techniques.

The Lisp-Stat graphics system can be used in many different ways. Simple animations, such as a normal probability plot controlled by a scroll bar for interactively changing the power in a power transformation, can be constructed easily using only the tools introduced in Chapter 2. The menu and dialog objects described in Chapter 7 can be used to design a graphical interface to a set of functions or a set of modeling tools. The objects and ideas presented in Chapters 8 and 9 can be used either to customize standard plots in minor ways or to develop entirely new types of plots. Chapter 10 presents several examples that illustrate the range of possibilities.

A good strategy for using Lisp-Stat is to start by interactively trying out some expressions at the keyboard. Once you identify a set of expressions or operations that fit together naturally, you can encapsulate them by defining a new function or a new type of object. This new function or object can then be used as a basic building block in further explorations. This strategy is

useful whether you are exploring a new data set, a new numerical technique, or a new graphical idea. The ability to move easily through a series of exploration and encapsulation steps is one of the keys to the power of an interactive statistical environment like Lisp-Stat.

1.2.3 Some Design and Portability Issues

A general objective in designing Lisp-Stat was to develop a system that can be used interactively on a wide range of computers. Several design decisions and compromises needed to be made in order to achieve this objective.

User Interface Variations

A major issue in designing a graphical system to work on a range of different computers is the difference among user interface designs. For example, the Macintosh interface usually places menus in a menu bar, whereas many other interfaces pop up menus when the mouse is clicked in a window. To deal with this issue, I have tried to design the Lisp-Stat graphics system around a few general conventions, leaving the details of the interface to a particular implementation. For example, the Lisp-Stat graphics system assumes that each plot has a menu for controlling it. From within Lisp-Stat the menu is treated identically across all implementations. Individual implementations are responsible for such details as determining when to present the menu to the user. On a Macintosh the menu might be placed in the menu bar when the plot is the front window. On other systems the menu might be popped up when a particular mouse button is clicked in the plot window.

The advantage of using only a few general interface features is that Lisp-Stat code can be written to run on a wide range on computers. The disadvantage is that it is not possible to exploit all the features offered by a particular interface. Fortunately, most graphical user interfaces are quite similar and little is lost by adhering to a common denominator.

Operating System Variations

Differences among operating systems lead to variations among Lisp-Stat systems whenever Lisp-Stat needs to interface to the operating system. This occurs when Lisp-Stat starts up and when a file is accessed. Each version of Lisp-Stat should include information on how to start Lisp-Stat from the operating system and on the naming conventions for files and directories. On some operating systems it may be possible to call other operating system commands from within Lisp-Stat or to link code written in other languages into Lisp-Stat. The details will differ from one operating system to another.

Different Lisp Systems

The ideal base for Lisp-Stat is a Common Lisp system. Unfortunately, Common Lisp systems can be rather expensive. Some Common Lisp systems also do not work well with the amount of memory available on most microcomputers and workstations. The subset of Common Lisp provided by XLISP is adequate to support a Lisp-Stat system; the most important Common Lisp feature that is not provided in XLISP is a compiler. Since XLISP is available free of charge, it is possible to make a version of Lisp-Stat based on XLISP available free of charge as well.

The Hardware Model

Like the user interface model, the hardware model used in designing Lisp-Stat is based on a greatest common denominator. It assumes a computer has a bitmapped display and a pointing device. It does not assume any special purpose hardware, such as hardware support for depth cuing or double buffering. A Lisp-Stat implementation can take advantage of hardware double buffering without changing the way Lisp-Stat views the hardware, but some changes would be needed to take advantage of hardware depth cuing. It remains to be seen whether such modifications are worthwhile.

1.2.4 The Future of Lisp-Stat

Lisp-Stat is an evolving system. Its objective is to provide an environment for statistical computing that takes full advantage of the computational speed and graphical capabilities of modern computing hardware by

- providing a useful set of programming tools for numerical statistical computing
- providing a framework for using higher-level data such as functions and expressions
- providing a set of useful building blocks for customizing and developing dynamic graphical methods

Experience in using Lisp-Stat and new developments in statistical methodology and in computing hardware will lead to modifications and additions to the system. As Lisp-Stat evolves, new versions should remain compatible with previous ones whenever possible.

Chapter 2

A Lisp-Stat Tutorial

This chapter is intended as an introduction to using Lisp-Stat as a statistical calculator and plotter. After introducing basic numerical operations and plots, it shows how to construct random or systematic data sets, how to modify data sets, how to use some built-in dynamic plots, and how to construct linear regression models. The final two sections give a brief introduction to writing functions of your own, and to some more advanced modeling tools that use functional data.

2.1 The Lisp Interpreter

Your interaction with a Lisp-Stat system consists of a conversation between you and the Lisp interpreter. Imagine you are sitting at your computer and have started your system. Or better yet, get a version of Lisp-Stat and start it up. When it is ready to start the conversation, it gives you a prompt like

>

at the beginning of a blank line. If you type in an expression, the interpreter responds by printing the result of evaluating that expression. For example, if you give the interpreter a number and type *return*, then it responds by simply printing the number back to you on the following line

```
> 1  
1
```

and then gives you a new prompt.

Operations on numbers can be performed by combining numbers and a symbol, representing an operation, into *compound expressions* like (+ 1 2):

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

As you have probably guessed, this expression means that the numbers 1 and 2 are to be added together. The notation in which the operator is placed before the operands is called *prefix notation*. This notation may be confusing at first, since it departs from standard mathematical practice, but it does offer some significant advantages. One advantage is that operations taking an arbitrary number of arguments can be accommodated. For example,

```
> (+ 1 2 3)
6
> (* 2 3.7 8.2)
60.68
```

To understand how the interpreter works, the basic rule to remember is that everything is evaluated. Numbers evaluate to themselves¹:

```
> 416
416
> 3.14
3.14
> -1
-1
```

Several other basic forms of data evaluate to themselves as well. These include logical values

```
> t ; true
T
> nil ; false
NIL
```

and strings

```
> "This is a string 1 2 3 4"
"This is a string 1 2 3 4"
```

always enclosed in double quotes. The semicolon ";" is the Lisp comment character. Anything you type after a semicolon, up to the next time you type a *return*, is ignored by the interpreter.

If the interpreter is given a symbol, like

pi

or

this-is-a-symbol

¹When typing a negative number, be sure not to leave any space between the minus sign and the number, or the interpreter will get confused.

to evaluate, it checks if there is a value associated with the symbol, and returns that value if there is one:

```
> pi  
3.141593
```

If there is no value associated with the symbol, the interpreter signals an error:

```
> x  
error: unbound variable - x
```

The symbol `pi` is predefined as an approximation to the number π . At this point the symbol `x` does not have a value. We will see how to give a symbol a value in the next section.

When typing a symbol's name you can use either upper case or lower case letters. Lisp internally converts lower case letters in a symbol's name to upper case.

Evaluating compound expressions is a bit more complicated. A compound expression is a list of elements separated by *white space* (spaces, tabs, or returns) and enclosed in parentheses. The elements of the list can be strings, numbers, or other compound expressions. In evaluating a compound expression like

```
(+ 1 2 3)
```

the interpreter treats the list as representing a *function application*. The first element of the list represents a Lisp function, a piece of code that can be applied to some arguments and that returns a result. In this case the function is the addition function, represented by the symbol `+`. The remaining elements of the list are the arguments; here the arguments are the numbers 1, 2 and 3. When asked to evaluate this expression, the interpreter takes the addition function, applies it to the arguments, and returns the result:

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

The arguments to a function are always evaluated before the function is applied. In the previous example the arguments were all numbers and thus evaluated to themselves. But in

```
> (+ (* 2 3) 4)  
10
```

the interpreter has to evaluate the first argument `(* 2 3)` to the function `+` before it can apply the function.

Numbers, strings, and symbols are some of the basic data types available in Lisp. I will refer to these basic data types jointly as *simple data*. In addition, Lisp provides several methods for forming more complex data structures, called *compound data*. The most basic form of compound data is the list. A list can be constructed using the `list` function:

```
> (list 1 2 3 4)
(1 2 3 4)
```

The result printed by the interpreter looks a lot like the compound expressions we have been using: a sequence of elements separated by white space and enclosed in parentheses. In fact, Lisp expressions are simply lists.²

Several other forms of compound data are available as well, including vectors and multi-dimensional arrays. These are described in Section 4.4.

Before going on to our first statistical applications, let's look at one more feature of the interpreter. Suppose we would like the interpreter to print the list `(+ 1 2)` back at us. Typing the list directly into the interpreter won't work because the interpreter will insist on evaluating the list:

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

The solution is to tell the interpreter *not* to evaluate the list. This process is called *quoting*. It is like putting quotation marks around an expression in a written sentence to say: take this expression literally. For example, using quotation marks gives the two sentences

Say your name!

and

Say “your name”!

very different meanings. Here is how you would quote our expression in Lisp:

```
> (quote (+ 1 2))
(+ 1 2)
```

`quote` is not a function. It does not obey the rules of function evaluation described above: its argument is not evaluated. `quote` is called a *special form* – special because it has special rules for the treatment of its arguments. I will introduce other special forms as they are needed. Together with the basic evaluation rules described here, these special forms make up the syntax of the Lisp language.

The special form `quote` is used so often that a shorthand notation has been developed, a single quote placed before the expression:

```
> '(+ 1 2)      ; single quote shorthand
(+ 1 2)
```

²The name Lisp is an acronym for LISt Processing. The language was originally developed as a tool for symbolic computation tasks, such as writing programs to symbolically differentiate an expression.

This is equivalent to `(quote (+ 1 2))`. There is no matching single quote following the expression.

Once you have learned how to start up a computer program, it is a good idea to make sure you know how to get back out of it. On most Lisp systems you can get out by using the `exit` function. On a UNIX system with a shell prompt `%`, using `exit` should return you to the shell:

```
> (exit)  
%
```

Other systems may offer additional ways of quitting; for example, on a Macintosh you can select the **Quit** item from the **File** menu.

Exercise 2.1

For each of the following expressions try to predict what the interpreter will return. Then, if you have a Lisp-Stat system available, type them in, see what happens, and try to explain any differences.

- a) `(+ 3 5 6)`
- b) `(+ (- 1 2) 3)`
- c) `'(+ 3 5 6)`
- d) `(+ (- (* 2 3) (/ 6 2)) 7)`
- e) `'x`
- f) `''x`

2.2 Elementary Computations and Graphs

This section introduces some of the basic numerical and graphical operations that are available in Lisp-Stat.

2.2.1 One-Dimensional Summaries and Plots

Statistical data usually consist of groups of numbers. As an example, the following table shows precipitation levels in inches recorded during the month of March in the Minneapolis-St. Paul area over a 30-year period [34].

0.77	1.74	0.81	1.20	1.95	1.20
0.47	1.43	3.37	2.20	3.00	3.09
1.51	2.10	0.52	1.62	1.31	0.32
0.59	0.81	2.81	1.87	1.18	1.35
4.75	2.48	0.96	1.89	0.90	2.05

To examine these data in Lisp-Stat, we can represent them as a list of numbers using the `list` function and the expression

```
(list .77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 2.20
      3.00 3.09 1.51 2.10 .52 1.62 1.31 .32 .59 .81
      2.81 1.87 1.18 1.35 4.75 2.48 .96 1.89 .90 2.05)
```

The numbers must be separated by white space, not commas. The interpreter lets you continue an expression over several lines, and does not try to evaluate it until you type a *return* after completing the expression.

The **mean** function can be used to compute the average of a list of numbers. We can combine it with the **list** function to find the average amount of precipitation for our sample:

```
> (mean (list .77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 2.20
            3.00 3.09 1.51 2.10 .52 1.62 1.31 .32 .59 .81
            2.81 1.87 1.18 1.35 4.75 2.48 .96 1.89 .90 2.05))
1.675
```

The median amount of precipitation can be computed as

```
> (median (list .77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 2.20
            3.00 3.09 1.51 2.10 .52 1.62 1.31 .32 .59 .81
            2.81 1.87 1.18 1.35 4.75 2.48 .96 1.89 .90
            2.05))
1.47
```

It is of course a nuisance to have to type in the list of 30 numbers every time we want to compute a statistic for this sample. To avoid having to do this, we can give this list a name using the Lisp-Stat special form **def**³:

```
> (def precipitation
    (list .77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 2.20
          3.00 3.09 1.51 2.10 .52 1.62 1.31 .32 .59 .81
          2.81 1.87 1.18 1.35 4.75 2.48 .96 1.89 .90 2.05))
```

PRECIPITATION

Now the symbol **precipitation** has a value associated with it, our list of 30 numbers:

```
> precipitation
(.77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 ...)
```

It is now much easier to compute various numerical descriptive statistics for these data:

```
> (mean precipitation)
1.675
```

³**def** acts like a special form, rather than a function, since its first argument is not evaluated (otherwise you would have to quote the symbol). Technically, **def** is a macro, not a special form, but I will not worry about this distinction in this chapter.

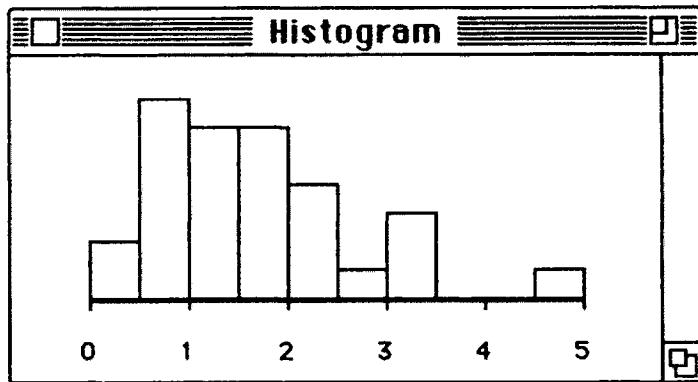


Figure 2.1: Histogram of precipitation levels.

```
> (median precipitation)
1.47

> (standard-deviation precipitation)
1.00062

> (interquartile-range precipitation)
1.145
```

The `histogram` and `boxplot` functions can be used to obtain graphical representations of these data. The expressions

`(histogram precipitation)`

and

`(boxplot precipitation)`

cause windows with graphs, as shown in Figures 2.1 and 2.2, to appear on the screen. These two functions return results that are printed something like this:

```
#<Object: 3564170, prototype = HISTOGRAM-PROTO>
```

These results will be used later to identify the windows containing the plots. For the moment you can ignore them.

Lisp-Stat also supports elementwise arithmetic operations on lists of numbers. For example, we can add 1 to each of the precipitation values:

```
> (+ 1 precipitation)      i
(1.77 2.74 1.81 2.2 2.95 ...)
```

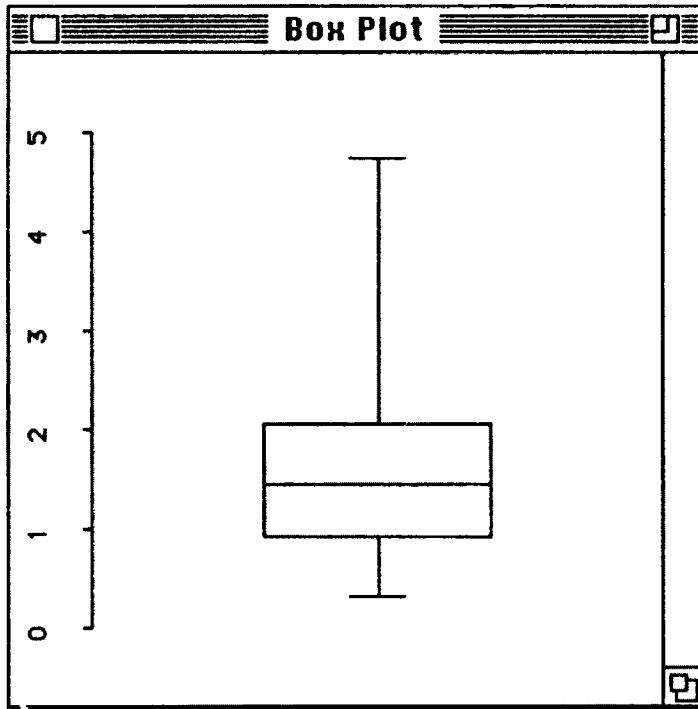


Figure 2.2: Boxplot of precipitation levels.

or we can compute their natural logarithms,

```
> (log precipitation)
(-0.2613648 0.5538851 -0.210721 0.1823216 ...)
```

or their square roots:

```
> (sqrt precipitation)
(0.877496 1.31909 0.9 1.09545 1.39642 ...)
```

The distribution of the precipitation data is somewhat skewed to the right and the mean and the median are separated. You might like to try a few simple transformations, such as square roots or logarithms, to see if you can symmetrize the data. You can look at plots and summary statistics to see if these transformations do lead to a more symmetric shape. For example, the mean of the square roots of the data can be calculated using

```
(mean (sqrt precipitation))
```

and you can obtain a histogram using the expression

```
(histogram (sqrt precipitation))
```

The `boxplot` function can also be used to produce parallel boxplots of two or more samples. This is done by giving `boxplot` a list of lists as its argument instead of a single list. As an example, let's use this function to examine some data from a study of the blood chemistry of several socioeconomic groups in Guatemala [60]. Serum total cholesterol values for samples of high-income urban and low-income rural individuals can be entered using the expressions

```
(def urban (list 206 170 155 155 134 239 234 228 330 284
                 201 241 179 244 200 205 279 227 197 242))
```

and

```
(def rural (list 108 152 129 146 174 194 152 223 231 131
                 142 173 155 220 172 148 143 158 108 136))
```

The parallel boxplot is obtained with the expression

```
(boxplot (list urban rural))
```

and is shown in Figure 2.3.

Exercise 2.2

For each of the following expressions try to predict what the interpreter will return. Then, if you have a Lisp-Stat system available, type them in, see what happens, and try to explain any differences.

- a) `(mean (list 1 2 3))`
- b) `(+ (list 1 2 3) 4)`
- c) `(* (list 1 2 3) (list 4 5 6))`
- d) `(+ (list 1 2 3) (list 4 5))`

Exercise 2.3

A study of acid rain [32] reported the average emission rates of SO₂, in pounds per million BTU, from utility boilers for 47 states and the District of Columbia. States not included are Idaho, Alaska, and Hawaii. The data are given in the following table.

2.3	0.6	0.5	0.2	0.7	0.5	1.5	1.0	1.7	2.9
2.7	4.2	2.2	1.0	3.6	0.1	1.4	2.1	1.8	1.8
1.5	1.3	4.5	0.7	1.0	0.6	2.9	0.9	0.6	1.4
1.5	1.2	3.8	0.2	0.7	2.5	1.0	1.9	1.7	3.7
0.3	0.4	1.2	1.4	1.7	2.7	3.4	1.0		

Obtain some plots and summary statistics for these data. Experiment with some transformations of the data as well.

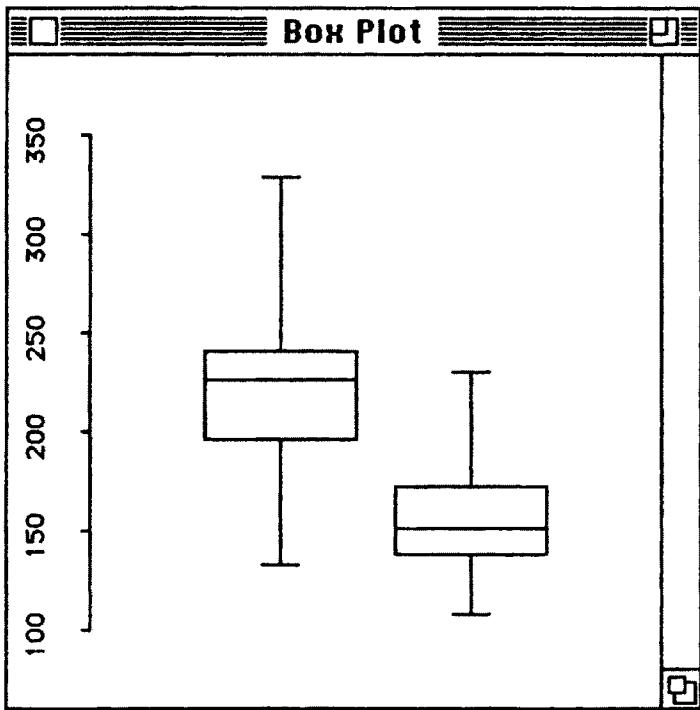


Figure 2.3: Parallel boxplots of cholesterol levels for urban (*left*) and rural (*right*) Guatemalans.

Exercise 2.4

The following data give the results of fuel economy tests (in miles per gallon) for 45 cars tested in 1965 and 45 cars tested in 1975 [14, page 152].

1965									
17.14	12.17	12.22	13.89	16.47	15.88	16.10	16.74	17.54	
14.57	12.90	12.81	14.95	16.25	17.13	14.46	14.20	16.90	
12.57	13.15	16.53	13.60	13.34	13.67	14.23	15.81	16.63	
14.94	13.66	9.79	13.08	14.57	14.93	14.01	14.43	16.35	
11.52	17.46	14.67	15.92	16.02	13.46	13.70	14.98	14.57	
1975									
24.57	24.79	22.21	25.84	25.35	22.19	24.37	21.32	22.74	
25.10	28.03	29.09	29.34	24.41	25.12	25.27	27.46	27.65	
21.67	22.15	24.36	26.32	24.05	28.27	26.57	26.10	24.35	
25.18	27.42	24.50	23.21	25.10	23.59	26.98	22.64	25.27	
27.18	24.69	26.35	23.05	23.37	25.46	28.84	22.14	25.42	

Obtain plots and summary statistics for the two samples separately, and look at a parallel box plot for the two samples.

2.2.2 Two-Dimensional Plots

Many single samples are actually collected over time. The precipitation data used above are an example of this kind of data. In some cases it is reasonable to assume that the observations are independent of one another, but in other cases it is not. One way to check the data for some form of serial correlation or trend is to plot the observations against time, or against the order in which they were obtained. We can use the **plot-points** function to produce a scatterplot of the precipitation data versus time. The **plot-points** function is called with an expression of the form

```
(plot-points <x-variable> <y-variable>)
```

Our *<y-variable>* will be **precipitation**, the variable we defined earlier. As our *<x-variable>* we would like to use a sequence of integers from 1 to 30. We could type these in ourselves, but there is an easier way. The function **iseq**, short for *integer-sequence*, generates a list of consecutive integers between two specified values. The general form for a call to this function is

```
(iseq <start> <end>)
```

To generate the sequence we need, we use

```
(iseq 1 30)
```

Thus to produce the scatterplot, we type the expression

```
(plot-points (iseq 1 30) precipitation)
```

The result is shown in Figure 2.4. There does not appear to be much of a pattern to the data; an independence assumption may be reasonable.

Sometimes it is easier to see temporal patterns in a plot if the points are connected by lines. A connected lines plot can be constructed by using the function **plot-lines** instead of **plot-points**. The **plot-lines** function can also be used to construct a graph of a function. Suppose you would like a plot of $\sin(x)$ from $-\pi$ to $+\pi$. The constant π is predefined as the variable **pi**. You can use the expression

```
(rseq <a> <b> <n>)
```

to construct a list of *<n>* equally spaced real numbers between *<a>* and **. Thus to plot the graph of $\sin(x)$ using 50 equally spaced points you can type

```
(plot-lines (rseq (- pi) pi 50) (sin (rseq (- pi) pi 50)))
```

You can simplify this expression by first defining a variable **x** as

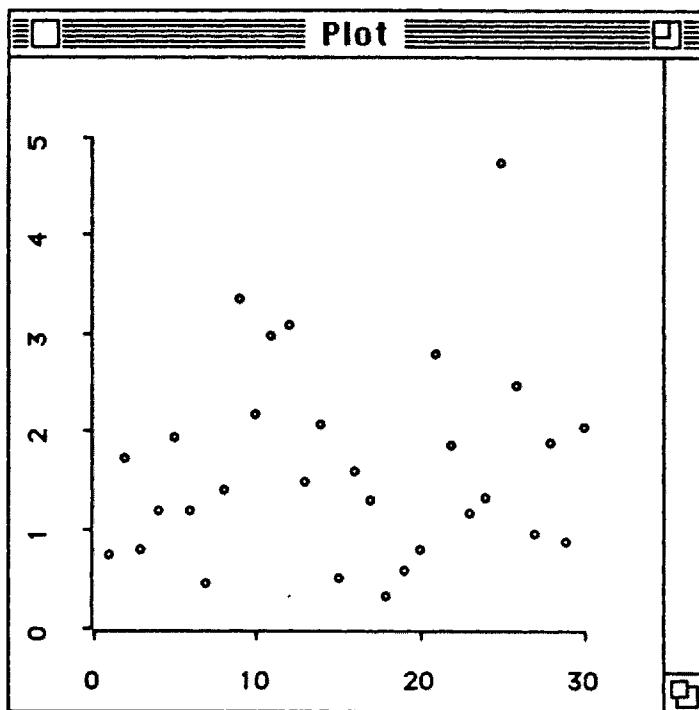


Figure 2.4: Scatterplot of precipitation levels against time.

```
(def x (rseq (- pi) pi 50))
```

and then constructing the plot with

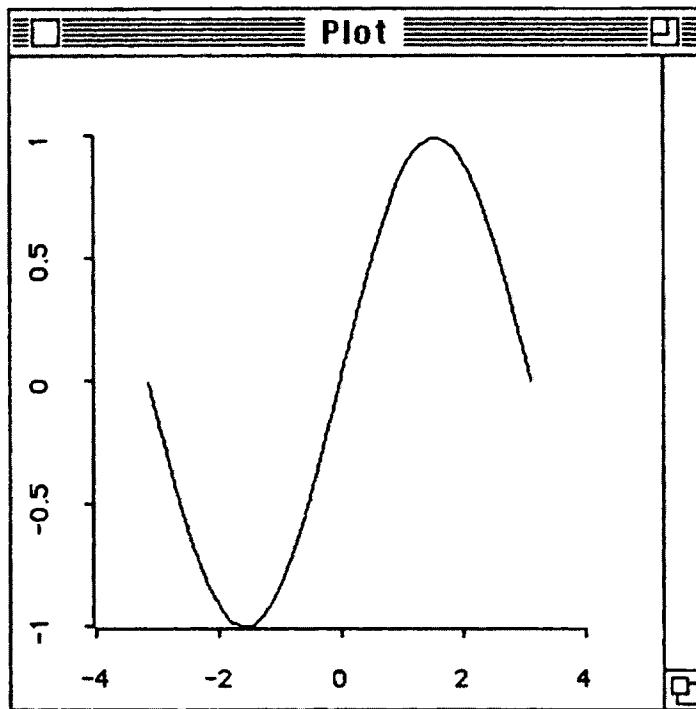
```
(plot-lines x (sin x))
```

The plot is shown in Figure 2.5.

Scatterplots are of course particularly useful for examining the relationship between two numerical observations taken on the same subject. A study of a procedure for monitoring motor vehicle emissions [41] gives data for HC and CO emissions recorded for 46 automobiles. The results can be placed in two variables, hc and co, by

```
(def hc (list .50 .65 .46 .41 .41 .39 .44 .55 .72 .64 .83 .38
      .38 .50 .60 .73 .83 .57 .34 .41 .37 1.02 .87 1.10
      .65 .43 .48 .41 .51 .41 .47 .52 .56 .70 .51 .52
      .57 .51 .36 .48 .52 .61 .58 .46 .47 .55))
```

and

Figure 2.5: A plot of $\sin(x)$.

```
(def co (list 5.01 14.67 8.60 4.42 4.95 7.24 7.51 12.30
             14.59 7.98 11.53 4.10 5.21 12.10 9.62 14.97
             15.13 5.04 3.95 3.38 4.12 23.53 19.00 22.92
             11.20 3.81 3.45 1.85 4.10 2.26 4.74 4.29
             5.36 14.83 5.69 6.35 6.02 5.79 2.03 4.62
             6.78 8.43 6.02 3.99 5.22 7.47))
```

They can then be plotted against one another using the `plot-points` function in the expression

```
(plot-points hc co)
```

Figure 2.6 shows the resulting plot.

Exercise 2.5

Draw a graph of the function $f(x) = 2x + x^2$ between -2 and 3.

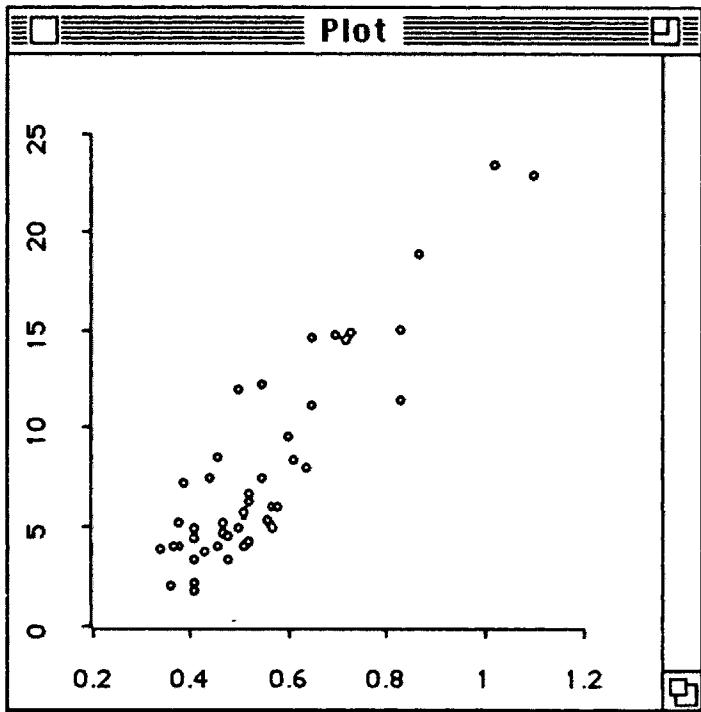


Figure 2.6: Plot of CO against HC for motor vehicle emissions.

Exercise 2.6

A study of the effect of cross-country skiing on the metabolism [69] gives the age and CPK concentration, a measure of metabolic activity, recorded for 18 cross-country skiers during a relay race:

Age	19	21	24	24	24	25	32	33	35
CPK	520	300	480	1040	1360	580	440	180	490
Age	37	37	44	50	51	52	55	57	62
CPK	520	380	640	360	240	420	280	400	260

Plot the data and describe any relationship you observe between age and CPK concentration.

2.2.3 Plotting Functions

Plotting the sine function in the previous section was rather cumbersome. As an alternative, we can use `plot-function` to plot a function of one argument over a specified range. To produce the plot in Figure 2.5, we can use the expression

```
(plot-function (function sin) (- pi) pi)
```

The expression `(function sin)` is used to extract the Lisp function associated with the symbol `sin`. Just using `sin` will not work. The reason is that a symbol in Lisp can have both a *value*, perhaps set using `def`, and a *function definition* at the same time. This may seem a bit peculiar at first, but it has one great advantage: typing an innocent expression like

```
(def list '(2 3 4))
```

will not destroy the `list` function.

Extracting a function definition from a symbol is done almost as often as quoting an expression, so again a simple shorthand notation is available. The expression

```
#'sin
```

is equivalent to the expression `(function sin)`. The short form `#'` is usually pronounced *sharp-quote*. Using this abbreviation, the expression for producing the sine plot can be written as

```
(plot-function #'sin (- pi) pi)
```

2.3 More on the Interpreter

The Lisp-Stat interpreter allows you to save variables you have created and to record all or part of a session with the interpreter. In addition, it provides mechanisms for accessing the last few computed results and for getting on-line help. Some of these features, in particular the on-line help system, may work slightly differently on different systems. Some systems may, for example, offer a separate help window. I will base my description on the XLISP-STAT implementation.

2.3.1 Saving Your Work

If you want to record a session with the interpreter, you can do so using the `dribble` function. The expression

```
(dribble "myfile")
```

starts a recording. All expressions typed by you and all results typed by the interpreter will be entered into the file named `myfile`. The expression

```
(dribble)
```

stops the recording. The expression `(dribble "myfile")` always starts a new file by the name `myfile`. If you already have a file by that name, its contents will be lost. Thus you can't use `dribble` to toggle on and off recording to a single file.

dribble only records text that is typed, not plots. In XLISP-STAT on the Macintosh you can use the standard Macintosh shortcut COMMAND-SHIFT-3 to save a MacPaint image of the current screen. You can also choose the **Copy** command from the **Edit** menu, or its command key shortcut COMMAND-C, while a plot window is the active window to save the contents of the plot window to the clipboard. On *X11* versions of XLISP-STAT, plot menus contain an item that saves a *PostScript* version of the plot to a file.

Variables you define in Lisp-Stat only exist for the duration of the current session. If you quit from Lisp-Stat, or the program crashes, your data will be lost. To preserve your data you can use the **savevar** function. This function allows you to save one or more variables into a file. Again a new file is created and any existing file by the same name is destroyed. To save the variable **precipitation** in a file called **precipitation.lsp**, type

```
(savevar 'precipitation "precipitation")
```

Do not add the **.lsp** suffix yourself; **savevar** will supply it. To save the two variables **hc** and **co** in the file **examples.lsp**, type

```
(savevar '(hc co) "examples")
```

I have used a quoted list '**(hc co)**' in this expression to pass the list of symbols to the **savevar** function. A longer alternative would be the expression **(list 'hc 'co)**.

The files **precipitation.lsp** and **examples.lsp** now contain a set of expressions that, when read in with the **load** function, will recreate the variables **precipitation**, **hc** and **co**. To load the file **precipitation.lsp**, for example, you can use the expression

```
(load "precipitation")
```

2.3.2 A Command History Mechanism

Common Lisp provides a simple command history mechanism. The symbols **-**, *****, ******, *******, **+**, **++**, and **+++** are used for this purpose. The interpreter binds these symbols as follows:

- the current input expression
- + the last expression read
- ++ the previous value of +
- +++ the previous value of ++
- * the result of the last evaluation
- ** the previous value of *
- *** the previous value of **

The variables *****, ******, and ******* are probably most useful. For example, if you take the logarithms of a variable but forget to give the result a name, you can use one of these history variables to avoid having to recompute the logarithms:

```
> (log precipitation)
(-0.2613648 0.5538851 -0.210721 0.1823216 ...)
> (def log-precip *)
LOG-PRECIP
```

The variable `log-precip` now contains the logarithms of the precipitation values.

This system takes advantage of the fact, discussed in Section 2.2.3, that symbols can have both function definitions and values at the same time. The function definition of `*` is the multiplication function; its value is the result of the last evaluation by the interpreter.

2.3.3 Getting Help

Lisp-Stat provides a large number of different functions. It would be impossible to remember exactly how to use each function. An interactive help facility can make the task of finding the right function to use and how to use it a lot easier. As an example, here is how you would get help for the function `median`⁴:

```
> (help 'median)
MEDIAN
[function-doc]
Args: (x)
Returns the median of the elements of X.
```

The quote in front of `median` is essential. `help` is itself a function, and its argument is the symbol representing the function `median`. To make sure `help` receives the symbol, not the value of the symbol, you need to quote the symbol.

If you are not sure about the name of a function, you may still be able to get some help by using the `help*` function. Suppose you want to find out about functions related to the normal distribution. Most such functions will have “norm” as part of their name. The expression

```
(help* 'norm)
```

prints the help information for all symbols whose names contain the string “norm”:

```
> (help* 'norm)
```

```
BIVNORM-CDF
[function-doc]
Args: (x y r)
Returns the value of the standard bivariate normal distribution
function with correlation R at (X, Y). Vectorized.
```

⁴As a reminder, the `help` function may work a little differently on different systems.

Sorry, no help available on NORM

Sorry, no help available on NORMAL

NORMAL-CDF [function-doc]

Args: (x)

Returns the value of the standard normal distribution function at X. Vectorized.

NORMAL-DENS [function-doc]

Args: (x)

Returns the density at X of the standard normal distribution. Vectorized.

NORMAL-QUANT [function-doc]

Args (p)

Returns the P-th quantile of the standard normal distribution. Vectorized.

NORMAL-RAND [function-doc]

Args: (n)

Returns a list of N standard normal random numbers. Vectorized.

The symbols norm and normal do not have function definitions and thus there is no help available for them. The term *vectorized* in a function's documentation means the function can be applied to arguments that are lists; the result is a list of the results of applying the function to each element. A related term appearing in the documentation of some functions is *vector reducing*. A function is vector reducing if it is applied recursively to its arguments until a single number results. The functions sum, prod, max, and min are vector reducing.

As an alternative to using **help***, you can use the function **apropos** to obtain a listing of the symbols that contain "norm" as part of their name

```
> (apropos 'norm)
NORMAL-QUANT
NORMAL-RAND
NORMAL-CDF
NORMAL-DENS
NORMAL
BIVNORM-CDF
NORM
```

and then use **help** to ask for more information about each of these symbols.

Let me briefly explain the notation used in the information printed by `help` to describe the arguments a function expects. This notation corresponds to the specification of the argument lists in Lisp function definitions described in Section 4.2. Most functions expect a fixed set of arguments, described in the help message by a line like

Args: (*x y z*)

Some functions can take one or more optional arguments. The arguments for such a function might be listed as

Args: (*x &optional y (z t)*)

This means that *x* is required and *y* and *z* are optional. If the function is named *f*, it can be called as (*f x-val*), (*f x-val y-val*), or (*f x-val y-val z-val*). The list (*z t*) means that if *z* is not supplied its default value is *t*. No explicit default value is specified for *y*; its default value is therefore *nil*. The arguments must be supplied in the order in which they are listed. Thus if you want to give the argument *z*, you must also give a value for *y*.

Another form of optional argument is the *keyword argument*. The histogram function, for example, takes arguments

Args: (*data &key (title "Histogram")*)

The *data* argument is required, the *title* argument is an optional keyword argument. The default title is "Histogram". If you want to create a histogram of the precipitation data used in Section 2.2.1 with title "Precipitation" use the expression

(*histogram precipitation :title "Precipitation"*)

Thus to give a value for a keyword argument, you give the *keyword* for the argument, a symbol consisting of a colon followed by the argument name, and then the value for the argument.⁵ If a function can take several keyword arguments, then these may be specified in any order, following the required and optional arguments.

Finally, some functions take an arbitrary number of arguments. This is denoted by a line like

Args: (*x &rest args*)

The argument *x* is required, and zero or more additional arguments can be supplied.

In addition to functions, `help` also gives information about data types and certain variables. For example,

⁵The keyword `:title` has not been quoted. *Keyword symbols*, symbols starting with a colon, are somewhat special. When a keyword symbol is created, its value is set to itself. Thus a keyword symbol effectively evaluates to itself and does not need to be quoted.

```
> (help 'complex)
COMPLEX [function-doc]
Args: (realpart &optional (imagpart 0))
Returns a complex number with the given real and imaginary
parts.
COMPLEX [type-doc]
A complex number
```

shows the function and type documentation for `complex`, and

```
> (help 'pi)
PI [variable-doc]
The floating-point number that is approximately equal to the
ratio of the circumference of a circle to its diameter.
```

shows the variable documentation for `pi`.⁶

2.3.4 Listing and Undefining Variables

After you have been working for a while, you may want to find out what variables you have defined using `def`. The function `variables` returns a list of these variables:

```
> (variables)
(CO HC PRECIPITATION RURAL URBAN)
```

You may occasionally want to free up some space by getting rid of some variables you no longer need. You can do this using the `undef` function:

```
> (undef 'co)
CO
> (variables)
(HC PRECIPITATION RURAL URBAN)
```

The argument to `undef` is a symbol; it has to be quoted in this example to prevent the interpreter from evaluating it. `undef` can also be given a list of symbols as its argument. Thus to undefine all currently defined variables, you can use

```
> (undef (variables))
(HC PRECIPITATION RURAL URBAN)
> (variables)
NIL
```

⁶Actually `pi` represents a constant, which is produced with `defconstant`. Its value can't be changed by simple assignment.

2.3.5 Interrupting a Calculation

Occasionally you may accidentally start the system off on a calculation that takes too long or seems to be producing unreasonable results. Every system provides a mechanism for interrupting such a calculation, but the particular mechanism used differs from system to system. In XLISP-STAT on the Macintosh you can interrupt a calculation by holding down the COMMAND key and the PERIOD key until the system returns you to the interpreter. On UNIX versions of XLISP-STAT you can interrupt a calculation by hitting the interrupt key combination for your system, usually CONTROL-C.

2.4 Some Data-Handling Functions

Until now we have typed data sets directly into the interpreter. The data sets were small, so this was not much of a problem. It is, however, useful to have some other alternatives. Lisp-Stat provides functions to generate systematic or random data, extract parts of data sets, modify data sets, and read in data from files.

2.4.1 Generating Systematic Data

The functions `iseq`, for generating sequences of consecutive integers, and `rseq`, for generating sequences of equally spaced real numbers, were already introduced in Section 2.2.2. The function `iseq` can also be used with a single argument. In this case the argument should be a positive integer n , and the result is a list of the integers $0, 1, \dots, n - 1$. For example,

```
> (iseq 10)
(0 1 2 3 4 5 6 7 8 9)
```

The function `repeat` is useful for generating sequences with a particular pattern. The general form of a call to `repeat` is

```
(repeat <vals> <pattern>)
```

The parameter `<vals>` can be a simple data item or a list of items. The parameter `<pattern>` must be either a single positive integer or a list of positive integers. If `<pattern>` is a single positive integer, then `repeat` simply repeats `<vals>` `<pattern>` times. For example,

```
> (repeat 2 3)
(2 2 2)
> (repeat (list 1 2 3) 2)
(1 2 3 1 2 3)
```

If `<pattern>` is a list, then `<vals>` should be a list of the same length as `<pattern>`. In this case each element of `<vals>` is repeated the number of times indicated by the corresponding element of `<pattern>`. For example,

```
> (repeat (list 1 2 3) (list 3 2 1))
(1 1 1 2 2 3)
```

The function **repeat** is particularly useful for coding treatment levels in designed experiments. As an example, in an experiment to examine the effect of plant density on tomato yields [2], three varieties of tomato plants were planted at four different planting densities. Each combination was replicated three times. The resulting yields are given in the following table:

Density	Variety								
	1			2			3		
1	9.2	12.4	5.0	8.9	9.2	6.0	16.3	15.2	9.4
2	12.4	14.5	8.6	12.7	14.0	12.3	18.2	18.0	16.9
3	12.9	16.4	12.1	14.6	16.0	14.7	20.8	20.6	18.7
4	10.9	14.3	9.2	12.6	13.0	13.0	18.3	16.0	13.0

We can enter the yields, working through the table by rows, as

```
(def yield (list 9.2 12.4 5.0 8.9 9.2 6.0 16.3 15.2 9.4
                 12.4 14.5 8.6 12.7 14.0 12.3 18.2 18.0 16.9
                 12.9 16.4 12.1 14.6 16.0 14.7 20.8 20.6 18.7
                 10.9 14.3 9.2 12.6 13.0 13.0 18.3 16.0 13.0))
```

Using **repeat**, we can generate variables to represent the different varieties and density levels used as

```
(def variety (repeat (repeat (list 1 2 3) (list 3 3 3)) 4))
```

and

```
(def density (repeat (list 1 2 3 4) (list 9 9 9 9)))
```

Exercise 2.7

For each of the following expressions try to predict what the interpreter will return. Then, if you have a Lisp-Stat system available, type them in, see what happens, and try to explain any differences.

- a) (repeat (iseq 1 4) (repeat 3 4))
- b) (repeat (repeat 3 4) (iseq 1 4))
- c) (repeat (repeat '(a b c) 2) (repeat (list 1 2) 3))

Exercise 2.8

An experiment on the quality of soldered connections on printed circuit boards uses five boards and four different kinds of chips. Each type of chip is used twice on each board. The results are recorded by board, chip type within board, and replication within chip type. Generate lists to represent the boards and chip types used in each replication.

2.4.2 Generating Random Data

Several functions can be used to generate pseudorandom numbers. For example, the expression

```
(uniform-rand 50)
```

generates a list of 50 independent random numbers distributed uniformly between 0 and 1. The functions `normal-rand` and `cauchy-rand` work similarly to generate standard normal or standard Cauchy random variates.

The expression

```
(gamma-rand 50 4)
```

generates a sample of size 50 from a gamma distribution with unit scale and exponent 4. Random variates from a beta distribution with exponents 3.5 and 7.2 can be generated by

```
(beta-rand 50 3.5 7.2)
```

Samples of size 50 from a t distribution with 4 degrees of freedom, a χ^2 distribution with 4 degrees of freedom, and an F distribution with 3 degrees of freedom for the numerator and 12 degrees of freedom for the denominator can be generated using the expressions

```
(t-rand 50 4)
(chisq-rand 50 4)
(f-rand 50 3 12)
```

respectively.

Samples from a binomial distribution with parameters $n = 5$ and $p = 0.3$ are generated by

```
(binomial-rand 50 5 .3)
```

and the expression

```
(poisson-rand 50 4.3)
```

generates a sample of size 50 from a Poisson distribution with mean $\lambda = 4.3$.

The sample size argument to these generator functions may also be a list of integers. The result will then be a list of lists of samples. For example,

```
(normal-rand (list 10 10 10))
```

generates a list of three lists of ten normal variates each.

Finally, you can use the function `sample` to select a random sample from a list. The expression

```
(sample (iseq 1 20) 5)
```

returns a list of a random sample of size 5 drawn without replacement from the integers $1, 2, \dots, 20$:

```
> (sample (iseq 1 20) 5)
(20 11 3 14 10)
```

`sample` can be given an optional third argument to specify that sampling is to be done with replacement. Thus

```
(sample (iseq 1 20) 5 t)
```

draws a sample with replacement:

```
> (sample (iseq 1 20) 5 t)
(2 14 14 11 18)
```

Giving `nil` as the third argument requests sampling without replacement and is thus equivalent to omitting the argument.

Exercise 2.9

Generate a random sample of size 30 from a normal distribution with mean 3.2 and standard deviation 1.7.

2.4.3 Forming Subsets and Deleting Cases

The `select` function allows you to select a single element or a group of elements from a list. For example, if we define `x` by

```
(def x (list 3 7 5 9 12 3 14 2))
```

then `(select x i)` returns the i -th element of `x`. Lisp, like the language C but in contrast to FORTRAN, numbers elements of lists starting at zero. Thus the indices for the elements of `x` are 0, 1, 2, 3, 4, 5, 6, 7. So

```
> (select x 0)
3
> (select x 2)
5
```

To get a group of elements at once we can use a list of indices instead of a single index:

```
> (select x (list 0 2))
(3 5)
```

If you want to select all elements of `x` except the element with index 2, you can use the expression

```
(remove 2 (iseq 8))
```

to produce the indices to use as the second argument to the function **select**:

```
> (remove 2 (iseq 8))
(0 1 3 4 5 6 7)
> (select x (remove 2 (iseq 8)))
(3 7 9 12 3 14 2)
```

The expression `(iseq 8)` generates a list of integers from 0 through 7 (i.e. a list of all the indices to a list of 8 elements). The function **remove** returns a list with all occurrences of a specified element removed; thus

```
> (remove 3 (list 1 3 4 3 2 5))
(1 4 2 5)
```

The remaining elements are returned in the order in which they appeared in the original list.

Another approach to selecting all elements of **x** except the element with index 2 is to use the comparison function `/=` (meaning *not equal*) to tell you which indices are not equal to 2:

```
> (/= 2 (iseq 8))
(T T NIL T T T T T)
```

The function **which** can then be used to return a list of all the indices for which the elements of its argument are not `nil`,

```
> (which (/= 2 (iseq 8)))
(0 1 3 4 5 6 7)
```

The result can then be passed to **select**:

```
> (select x (which (/= 2 (iseq 8))))
(3 7 9 12 3 14 2)
```

This approach is a little more cumbersome for deleting a single element, but it is more general. For example, the expression

```
(select x (which (< 3 x)))
```

returns all elements in **x** that are greater than 3:

```
> (select x (which (< 3 x)))
(7 5 9 12 14)
```

Since the list **x** used in these examples is so short, it is easy to determine its length by just counting the elements. For longer lists this may not be so easy, and we can use the function **length** instead. Using **length**, we can write the expression for obtaining all elements of **x** except element 2 as

```
(select x (remove 2 (iseq (length x))))
```

Exercise 2.10

Write an expression to obtain all elements of a list with values between 3 and 5, inclusive. Use the function `<=`. It takes two or more arguments. If its arguments are numbers, it returns `t` if the numbers are in nondecreasing order; otherwise it returns `nil`. If one or more of the arguments is a list, then the function is applied elementwise and returns a list of the results.

2.4.4 Combining Several Lists

At times you may want to combine several short lists into a single longer list. This can be done using the functions `append` and `combine`. For example,

```
> (append (list 1 2 3) (list 4) (list 5 6 7 8))
(1 2 3 4 5 6 7 8)
```

or

```
> (combine (list 1 2 3) (list 4) (list 5 6 7 8))
(1 2 3 4 5 6 7 8)
```

The difference between these two functions is in the way they handle lists of lists. `append` combines only the outermost lists, while `combine` recursively works through sublists, returning as its result a list with noncompound elements:

```
> (append (list (list 1 2) 3) (list 4 5))
((1 2) 3 4 5)
> (combine (list (list 1 2) 3) (list 4 5))
(1 2 3 4 5)
```

`combine` also allows its arguments to be numbers or other simple data items

```
> (combine 1 2 (list 3 4))
(1 2 3 4)
```

while `append` requires all its arguments to be lists.

2.4.5 Modifying Data

The special form `setf` can be used to modify individual elements or sets of elements of a list. Suppose again that we have set up a list `x` as

```
(def x (list 3 7 5 9 12 3 14 2))
```

and would like to change the 12 to 11. The expression

```
(setf (select x 4) 11)
```

makes this replacement:

```
> (setf (select x 4) 11)
11
> x
(3 7 5 9 11 3 14 2)
```

The general form of **setf** is

```
(setf <form> <value>)
```

where *<form>* is the expression you would use to select a single element or a group of elements from a list and *<value>* is the value you would like that element to have, or the list of the values for the elements in the group. Thus the expression

```
(setf (select x (list 0 2)) (list 15 16))
```

changes the values of elements 0 and 2 to 15 and 16:

```
> (setf (select x (list 0 2)) (list 15 16))
(15 16)
> x
(15 7 16 9 11 3 14 2)
```

A note of caution is needed here. Lisp symbols are merely labels for different items. When you assign a name to an item with the **def** command, you are not producing a new item. Thus after evaluating the expressions

```
(def x (list 1 2 3 4))
```

and

```
(def y x)
```

the symbols **x** and **y** are two different names for the same list. As a result, if we change an element of (the list referred to by) **x** with **setf**, then we are also changing the element of (the list referred to by) **y**, since both **x** and **y** refer to the same list. If you want to make a copy of **x** and store it in **y** before you make changes to **x**, then you must do so explicitly, using the **copy-list** function. The expression

```
(def y (copy-list x))
```

makes a copy of **x** and sets the value of **y** to that copy. Now **x** and **y** refer to different items, and changes to **x** will not affect **y**.

The special form **setf** can also be used to assign a value to a symbol, as in

```
> (setf z 3)
3
> z
3
```

The advantage to using `def` instead of `setf` for defining variables is that `def` records the names of the variables it defines, allowing you to find out which variables you have available using the function `variables`. There are several other uses for `setf`; we will look at them later in Section 3.8.

Exercise 2.11

Suppose the expressions

```
(def y (list 1 (list 2 3) 4))
(def x (select y 1))
(setf (select x 0) 'a)
```

are evaluated in order. What are the values of `x` and `y` after these evaluations?

2.4.6 Reading Data Files

Two functions are available for reading in data from files. The first function, `read-data-columns`, is designed for reading files containing several columns of data. This function takes two arguments: a string representing the name of the file and an integer, the number of columns in the file. The result returned is a list of lists, one for each column in the file. Items can be separated by any amount of white space, but not by commas or other punctuation marks. Items can be any valid Lisp expressions. In particular, they can be numbers, strings or symbols. All columns must be of equal length. The individual lists in the result can then be extracted using the `select` function.

Suppose, for example, that we have a file named "mydata" containing two columns of numbers. We would like to read these in and name the columns `x` and `y`. We can do this by first reading in the entire file and saving the result in a variable named `mydata`:

```
(def mydata (read-data-columns "mydata" 2))
```

The value of the variable `mydata` is now a list of two lists of numbers. To extract and name these two lists, we can use the expressions

```
(def x (select mydata 0))
```

and

```
(def y (select mydata 1))
```

The second argument to `read-data-columns` is optional. If it is not supplied, the number of columns is determined from the number of entries on the first nonblank line of the file.

On some systems the file name argument may be optional as well. If it is omitted, a dialog for selecting the file to read is presented.

If your data file is not arranged in columns, you can use `read-data-file` to read in the entire contents of a file and return it as a single list. This list

can then be manipulated into an appropriate form using Lisp-Stat functions. The entries of the file can be as for `read-data-columns`. The function `read-data-file` takes a single argument, a string with the name of the file to read. On some systems you can omit this argument and select the file to read with a dialog.

These two functions should be adequate for most purposes. If you have to read a file that does not fit into the form considered here, you can use Lisp's raw file-handling functions. Some of these are described below in Section 4.5.

2.5 Dynamic Graphs

In Section 2.2 we used histograms and scatterplots to examine the distribution of a single variable and the relationship between two variables. This section introduces some dynamic plots and techniques that can be used to explore relationships among three or more variables. The techniques and plots described in the first four subsections are simple, requiring only mouse actions and simple commands to the interpreter. The fifth subsection introduces the concept of a *plot object* and the idea of *sending messages* to an object from the interpreter. These ideas are used to add lines and curves to scatterplots. The basic concepts of objects and messages will be used again in the next section on regression models. The final subsection shows how Lisp iteration can be used to construct a dynamic simulation – a plot movie.

2.5.1 Spinning Plots

In exploring the relationship among three variables, it is natural to imagine constructing a three-dimensional scatterplot of the data. Of course we can only see a two-dimensional projection of the plot on a computer screen – any depth that you might be able to perceive by looking at a wire model of the data is lost. One way to try to recover some of this depth perception is to rotate the points around some axis. The `spin-plot` function allows you to construct a rotatable three-dimensional plot.

As an example, let's look at some data collected to examine the relationship between the amount of material lost when rubber specimens are rubbed with an abrasive material, and two characteristics of the specimens, their hardness and tensile strength [24, page 239]. Hardness is measured in degrees Shore, tensile strength in kilograms per square centimeter, and abrasion loss in grams per horsepower-hour. The data can be entered as

```
(def hardness
  (list 45 55 61 66 71 71 81 86 53 60 64 68 79 81 56 68 75
        83 88 59 71 80 82 89 51 59 65 74 81 86))
```

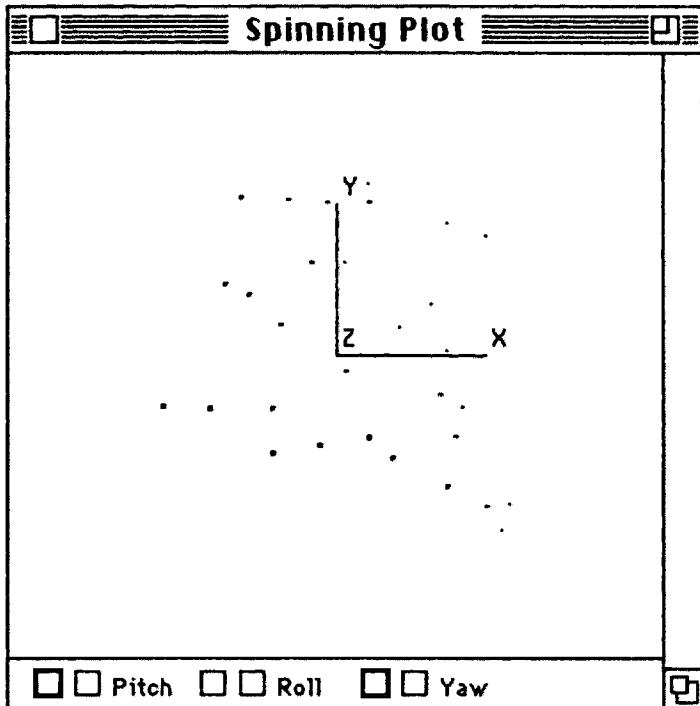


Figure 2.7: A rotatable plot of the abrasion loss data.

```
(def tensile-strength
  (list 162 233 232 231 231 237 224 219 203 189 210 210 196
        180 200 173 188 161 119 161 151 165 151 128 161 146
        148 144 134 127))
```

and

```
(def abrasion-loss
  (list 372 206 175 154 136 112 55 45 221 166 164 113 82
        32 228 196 128 97 64 249 219 186 155 114 341 340
        283 267 215 148))
```

The expression

```
(spin-plot (list hardness tensile-strength abrasion-loss))
```

produces the plot in Figure 2.7. The argument to `spin-plot` is a list of three lists or vectors, representing the x , y , and z variables. The z axis initially points out of the screen. You can rotate the plot by placing the mouse cursor in one of the `Pitch`, `Roll`, or `Yaw` squares and pressing the mouse button. By rotating the plot, you can see that the points seem to fall close to a plane.

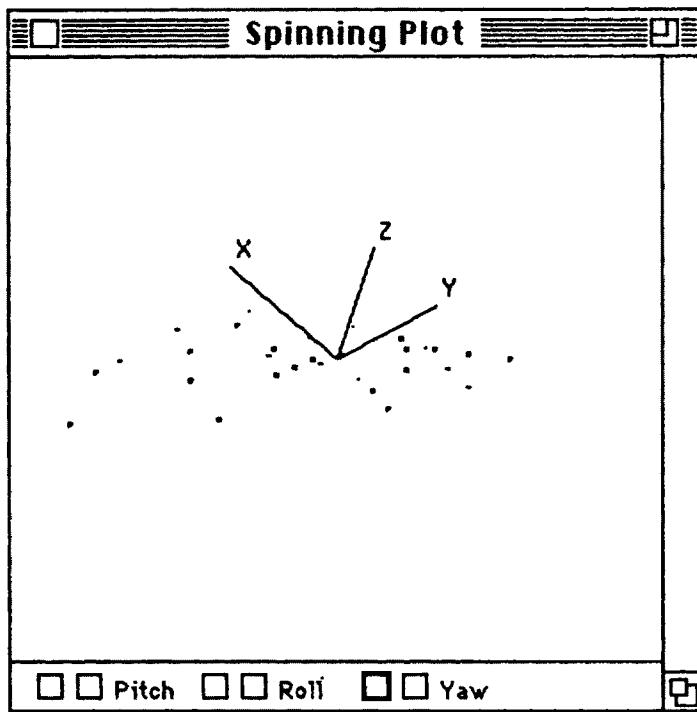


Figure 2.8: A second view of the abrasion loss data.

The plot in Figure 2.8 shows the data viewed along the plane. Thus a linear model should provide a reasonable fit. In fact, after rotating the plot for a while, you may notice a slight twist in the points, suggesting the addition of some second order terms. Unfortunately, this is very difficult to convey in print.

Several options to `spin-plot` can be specified as keyword arguments. Giving the `:title` keyword with a string allows you to specify an alternative title for the plot window. The keyword `:variable-labels` can be used to provide a list of alternative axis labels as a list of strings. For example, for the abrasion loss data you might use the expression

```
(spin-plot (list hardness tensile-strength abrasion-loss)
          :title "Abrasion Loss Data"
          :variable-labels (list "Hardness"
                                 "Tensile Strength"
                                 "Abrasion Loss"))
```

to construct the plot in place of the expression given above.

The function `spin-plot` also accepts the keyword argument `:scale`. If `:scale` is supplied as `t`, the default, then the data are centered at the

midranges of the three variables, and all three variables are scaled to fit the plot. If `:scale` is given as `nil`, the data are assumed to be scaled between -1 and 1, and the plot is rotated about the origin. Thus if you want to center your plot at the means of the variables and to scale all observations by the same amount, you can use the expression

```
(spin-plot
  (list (/ (- hardness (mean hardness)) 140)
        (/ (- tensile-strength (mean tensile-strength)) 140)
        (/ (- abrasion-loss (mean abrasion-loss)) 140))
  :scale nil)
```

Every Lisp-Stat plot window provides a menu for communicating with the plot. The precise way in which this menu is made available depends on the window management system. In XLISP-STAT on the Macintosh a plot's menu is placed in the menu bar when the plot becomes the front window; in the *X11* version of XLISP-STAT a button on the window is used to pop up the menu. The menu on a rotating plot allows you to change the speed of rotation, or to choose whether to use depth cuing or whether to show the coordinate axes. By default, the plot uses depth cuing: points closer to the viewer are drawn larger than points farther away.

Most window systems provide a convention for modifying mouse clicks in order to distinguish between clicks intended to start a selection of text and clicks intended to extend an existing selection. On the Macintosh the convention is to signal an extension by holding down the *shift* key while clicking the mouse; in other words, the *shift* key is the *extend modifier* for the Macintosh. In the rotating plot, clicking the mouse button on one of the *Pitch*, *Roll*, or *Yaw* squares without the extend modifier causes the plot to rotate while the mouse button is held down. Clicking with the appropriate extend modifier causes rotation to continue after the button is released. Rotation stops the next time the mouse is clicked in the rotation control panel.

Exercise 2.12

An experiment was conducted to examine the relationship between the rate of flow of sulphate of ammonia crystals in a packing process, the initial moisture content (in units of 0.01%) and the shape of the crystals (measured by the ratio of their length and breadth) [24, page 251]. The data can be entered with the expressions

```
(def flow (list 5.00 4.81 4.46 4.84 4.46 3.85 3.21 3.25 4.55
               4.85 4.00 3.62 5.15 3.76 4.90 4.13 5.10 5.05
               4.27 4.90 4.55 4.39 4.85 4.59 5.00 3.82 3.68
               5.15 2.94 5.00 4.10 1.15 1.72 4.20 5.00))
```

```
(def moist (list 21 20 16 18 16 18 12 12 13 13 17 24 11 10 17
                14 14 14 20 12 11 10 16 17 17 17 15 17 21 21
                21 26 21 17 11))
```

and

```
(def ratio (list 2.4 2.4 2.4 2.5 3.2 3.1 3.2 2.7 2.7 2.7
                 2.8 2.5 2.6 2.0 2.0 2.0 1.9 2.1 1.9 2.0 2.0
                 2.0 2.2 2.4 2.4 2.4 2.2 2.2 1.9 2.4 3.5 3.0
                 3.5 3.2))
```

Use `spin-plot` to examine these data.

Exercise 2.13

A large computer manufacturer once included a random number generator called RANDU in its software library. This generator was supposed to produce numbers that behaved like *i.i.d.* uniform random variables. The expression

```
(defun randu (n &optional (seed 12345))
  (let ((a (+ (^ 2 16) 3))
        (m (^ 2 31)))
    (flet ((mod (x m) (- x (* m (floor (/ x m))))))
      (do ((i 0 (+ i 1))
            (res (list seed)
                  (cons (mod (* a (first res)) m) res)))
           ((>= i n)
            (/ (rest (reverse res)) m))))))
```

defines a function implementing the generator. The expression

```
(def randu
  (apply #'mapcar #'list (split-list (randu 300) 3)))
```

uses the function to generate a list of 300 numbers, and splits the list into three lists using consecutive triples from the original sequence for the corresponding elements in the lists. You will be able to understand these expressions after reading Sections 3.1 and 4.2. Use `spin-plot` to see if you can spot any unusual features in these data.

2.5.2 Scatterplot Matrices

Another approach to graphing a set of variables is to look at a matrix of all possible pairwise scatterplots of the variables. The `scatterplot-matrix` function produces such a plot. Using the abrasion loss data of the previous subsection, the expression

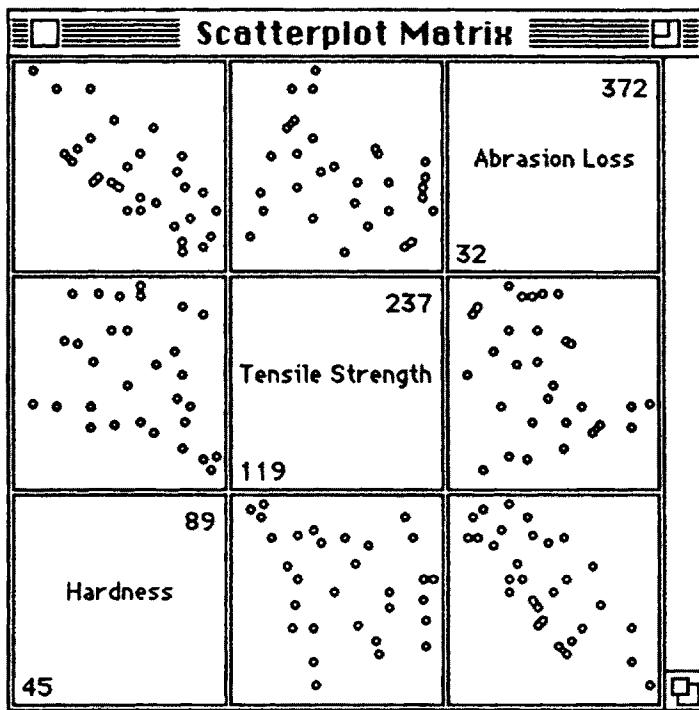


Figure 2.9: Scatterplot matrix of the abrasion loss data.

```
(scatterplot-matrix
  (list hardness tensile-strength abrasion-loss)
  :variable-labels
  (list "Hardness" "Tensile Strength" "Abrasion Loss"))
```

produces the scatterplot matrix in Figure 2.9.

The plot of **abrasion-loss** against **tensile-strength** gives you an idea of the joint variation in these two variables. But **hardness** varies from point to point as well. To get an understanding of the relationship among all three variables, it would be nice to be able to fix **hardness** at various levels and look at the way the plot of **abrasion-loss** against **tensile-strength** changes as you change these levels. You can do this kind of exploration in a scatterplot matrix by using two highlighting techniques called *selecting* and *brushing*.

Selecting. Your plot is in selecting mode when the cursor is an arrow. This is the default setting. In this mode you can select a point by moving the arrow cursor onto the point and clicking the mouse. To select a group of points, drag a selection rectangle around the group. If the group does not fit in a rectangle, you can build up your selection

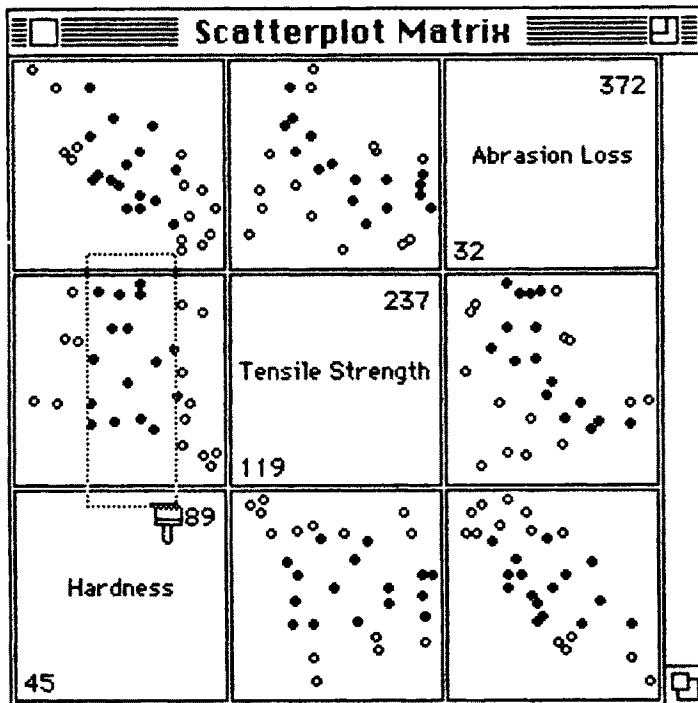


Figure 2.10: Scatterplot matrix of the abrasion loss data with middle hardness values highlighted.

by using *extend* dragging. On the Macintosh this is done by holding down the *shift* key as you click or drag. If you click without the extend modifier, any existing selection is unselected; when you use the extend modifier, selected points remain selected.

Brushing. You can enter the brushing mode by choosing **Mouse Mode** from the plot menu, and then selecting **Brushing** from the mode dialog that is presented. In this mode the cursor looks like a paint brush and a dashed rectangle, the *brush*, is attached to it. As you move the brush across the plot, points in the brush are highlighted. Points outside of the brush are not highlighted unless they are marked as selected. To select points in the brushing mode (make their highlighting permanent) hold the mouse button down as you move the brush.

In the plot in Figure 2.10 the points within the middle of the **hardness** range have been highlighted using a long, thin brush. You can change the size of your brush by using the **Resize Brush** command on the plot menu. In the plot of **abrasion-loss** against **tensile-strength** you can see that the highlighted points seem to follow a curve. If you want to fit a model

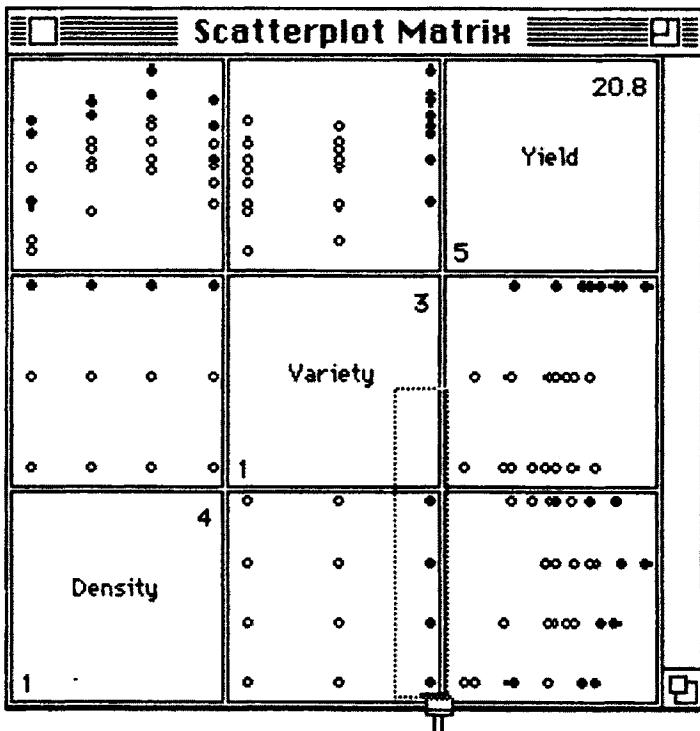


Figure 2.11: Scatterplot matrix for the tomato yield data with points from the third variety highlighted.

to these data, this observation suggests fitting a model that accounts for this curvature. This is similar to the conclusion reached from looking at a rotating plot of these data.

Highlighting by selection can also be useful in a rotating plot. By selecting a slice of points with approximately the same value for one of the variables and then rotating the plot, you can examine the three-dimensional structure of the conditional distribution of the other two variables given the one fixed by the selection.

A scatterplot matrix is also useful for examining the relationship between a quantitative variable and several categorical variables. In Section 2.4.1 we looked at data from an experiment on tomato plants using three varieties of tomatoes and four planting densities. Figure 2.11 shows a scatterplot matrix for these data. In this plot a long, thin brush has been used to highlight the points in the third variety. If there is no interaction between the varieties and the density, then the shape of the highlighted points should move approximately in parallel as the brush is moved from one variety to another.

Like `spin-plot`, the function `scatterplot-matrix` also accepts the optional keyword arguments `:title`, `:variable-labels` and `:scale`.

Exercise 2.14

An experiment to determine the effect of oxygen concentration on fermentation end products used four oxygen concentrations and two types of sugar, and recorded the amount of ethanol produced for two replications at each oxygen-sugar combination [68]. The data are given in the following table:

Oxygen	Sugar	
	Galactose	Glucose
1	0.59 0.30	0.25 0.03
2	0.44 0.18	0.13 0.02
3	0.22 0.23	0.07 0.00
4	0.12 0.13	0.00 0.01

Use a scatterplot matrix to examine these data.

Exercise 2.15

Use scatterplot matrices to examine the data in the exercises of Section 2.5.1.

2.5.3 Interacting with Individual Plots

Rotating plots and scatterplot matrices are interactive plots. Simple scatterplots also allow some interaction. If you select the **Show Labels** option in the plot menu, a label will appear next to a highlighted point. You can use either the selecting or the brushing mode to highlight points. The default labels are of the form “0”, “1”, Most plotting functions allow you to supply a list of alternative labels using the `:point-labels` keyword.

Another option that is useful in viewing large data sets is to remove a subset of the points from your plot. This can be done by selecting the points you want to remove and then choosing **Remove Selection** from the plot menu. The plot can then be rescaled using the **Rescale Plot** menu option.

When a set of points is selected in a plot, you can change the symbol used to display the points using the **Selection Symbol** item.⁷ If your system supports the use of color, you can use the **Selection Color** item to set the color used to draw the selected points.

You can save the indices of the currently selected points in a variable by choosing the **Selection...** item in the plot menu. A dialog box will ask you for a name for the selection. When no points are selected, you can use the **Selection...** menu item to specify the indices of the points to select. A

⁷The rotating plot uses different symbols as part of its depth cuing strategy. As a result, you can not change the symbols used in a rotating plot when the plot is using depth cuing.

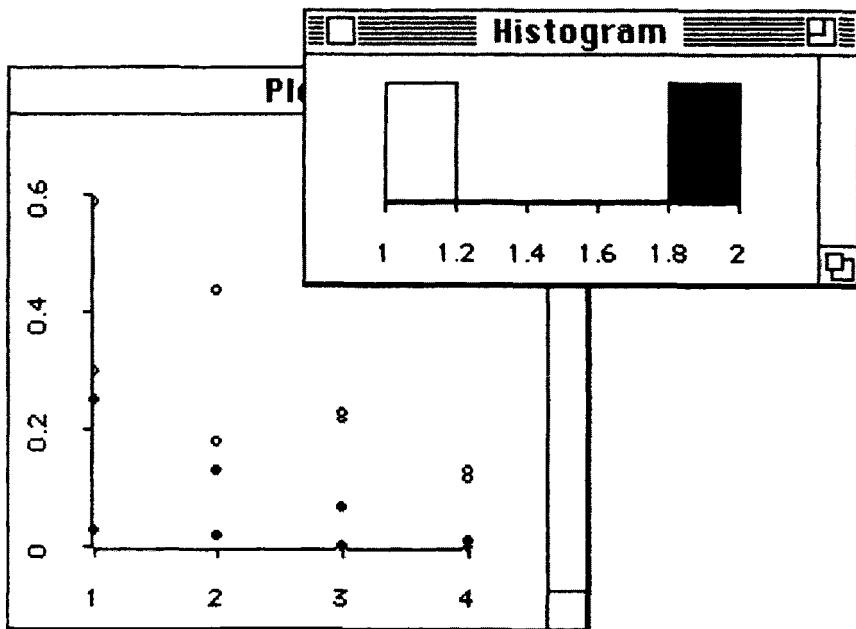


Figure 2.12: Linked scatterplot of ethanol against oxygen level and histogram of sugar type index for the fermentation data. Points from one sugar group are highlighted.

dialog box will ask you for an expression for determining the indices of the points to select. The expression can be any Lisp expression that evaluates to a single index or a list of indices.

2.5.4 Linked Plots

When you brush or select in a scatterplot matrix, you are looking at the interaction of a set of separate scatterplots. You can construct your own set of interacting plots by choosing the **Link View** option from the menus of the plots you want to link. For example, using the data from Exercise 2.14 in Section 2.5.2, we can put **ethanol** and **oxygen** in a scatterplot and **sugar** in a histogram. If we link these two plots, then selecting one of the two sugar groups in the histogram highlights the corresponding points in the scatterplot, as shown in Figure 2.12.

If you want to be able to select points with particular labels, you can use the **name-list** function to generate a window with a list of labels in it. This window can be linked with any plot, and selecting a label in a name list will then highlight the corresponding points in the linked plots. You can use the

`name-list` function with a numerical argument; for example,

```
(name-list 10)
```

generates a name list with the labels “0”, ..., “9”, or you can give it a list of labels of your own.

Exercise 2.16

Use linked scatterplots and histograms to examine the data in the examples and exercises of Sections 2.5.1 and 2.5.2.

2.5.5 Modifying a Scatterplot

After producing a scatterplot of a data set, you might like to add a line, for example, a regression line, to the plot. As an illustration, a study of the effect of bicycle lanes on drivers and bicyclists [37] reported the distance in feet between a cyclist and the roadway center line, and the distance between the cyclist and a passing car, recorded for 10 cyclists. The data can be entered as

```
(def travel-space
  (list 12.8 12.9 12.9 13.6 14.5 14.6 15.1 17.5 19.5 20.8))
```

and

```
(def separation
  (list 5.5 6.2 6.3 7.0 7.8 8.3 7.1 10.0 10.8 11.0))
```

A regression line fit to these data, with `separation` as the dependent variable, has an intercept of -2.18 and a slope of 0.66. Let's see how to add this line to a scatterplot of the data.

We can use the expression

```
(plot-points travel-space separation)
```

to produce a scatterplot of these points. To be able to add a line to the plot, however, we must be able to refer to it within Lisp-Stat. To accomplish this, we can assign the result returned by the `plot-points` function to a symbol:

```
(def myplot (plot-points travel-space separation))
```

The result returned by `plot-points` is a Lisp-Stat *object*.⁸ To use an object, you have to send it *messages*. This is done using the `send` function, as in the expression

⁸The result returned by `plot-points` is printed something like `*<Object: ...>` This is not the *value* returned by the function, just its *printed representation*. There are several other data types that are printed in this way; *file streams*, as returned by the `open` function, are one example. For the most part you can ignore these printed results. There is one unfortunate feature, however: the form `#<...>` means that there is no printed form of this data type that the Lisp reader can understand. As a result, if you forget to give your plot a name, you can't cut and paste the result into a `def` expression – you have to redo the plot or use the history mechanism.

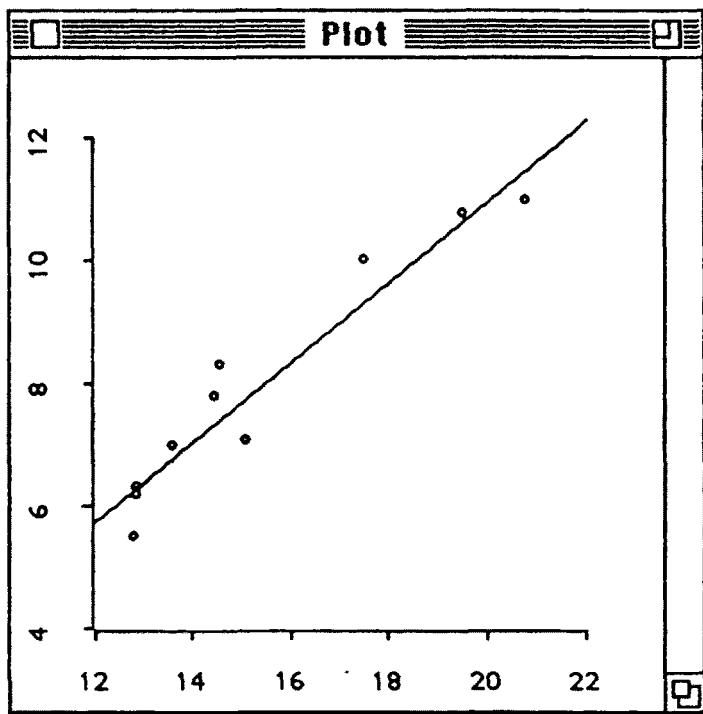


Figure 2.13: Scatterplot of the bicycle data with a fitted line.

```
(send <object> <message selector> <argument 1> ...)
```

The expression

```
(send myplot :abline -2.18 0.66)
```

tells `myplot` to add the graph of a line $a + bx$, with $a = -2.18$ and $b = 0.66$, to itself. The `<message selector>` is `:abline`; the numbers `-2.18` and `0.66` are the arguments. The message consists of the selector and the arguments. Message selectors are always Lisp keywords; that is, they are symbols that start with a colon. The resulting plot is shown in Figure 2.13.

Scatterplot objects understand a number of other messages, including the `:help` message⁹:

```
> (send myplot :help)
SCATTERPLOT-PROTO
```

⁹To keep things simple, I will use the term *message* to refer to a message corresponding to a message selector.

Scatterplot.

Help is available on the following:

```
:ABLINE :ACTIVATE :ADD-FUNCTION :ADD-LINES :ADD-METHOD
:ADD-.MOUSE-MODE :ADD-POINTS :ADD-SLOT :ADD-STRINGS
:ADJUST-HILITE-STATE :ADJUST-POINT-SCREEN-STATES
:ADJUST-POINTS-IN-RECT :ADJUST-TO-DATA :ALL-POINTS-SHOWING-P
:ALL-POINTS-UNMASKED-P :ALLOCATE :ANY-POINTS-SELECTED-P
:APPLY-TRANSFORMATION :BACK-COLOR :BRUSH :BUFFER-TO-SCREEN
:CANVAS-HEIGHT :CANVAS-WIDTH :CLEAR :CLEAR-LINES :CLEAR-MASKS
:CLEAR-POINTS :CLEAR-STRINGS ....
```

The list of topics is the same for all scatterplots, but is somewhat different for rotating plots, scatterplot matrices, and histograms.

The `:clear` message, as its name suggests, clears the plot and allows you to build up a new plot from scratch. Two other useful messages are `:add-points` and `:add-lines`. To find out how to use them, we can use the `:help` message with `:add-points` or `:add-lines` as arguments:

```
> (send myplot :help :add-points)
:ADD-POINTS
Method args: (points &key point-labels (draw t))
or:          (x y &key point-labels (draw t))
Adds points to plot. POINTS is a list of sequences,
POINT-LABELS a list of strings. If DRAW is true the new points
are added to the screen. For a 2D plot POINTS can be replaced
by two sequences X and Y.

> (send myplot :help :add-lines)
:ADD-LINES
Method args: (lines &key type (draw t))
or:          (x y &key point-labels (draw t))
Adds lines to plot. LINES is a list of sequences, the
coordinates of the line starts. TYPE is normal or dashed. If
DRAW is true the new lines are added to the screen. For a 2D
plot LINES can be replaced by two sequences X and Y.
```

The term *sequence* in these descriptions means a list or a vector.

The plot in Figure 2.13 shows some curvature in the data. A regression of `separation` on a linear and a quadratic term in `travel-space` produces estimates of -16.42 for the constant, 2.433 as the coefficient of the linear term, and -0.05339 as the coefficient of the quadratic term. Let's use the `:clear`, `:add-points`, and `:add-lines` messages to change `myplot` to show the data along with the fitted quadratic model. First we use the expressions

```
(def x (rseq 12 22 50))
(def y (+ -16.42 (* 2.433 x) (* -0.05339 (* x x))))
```

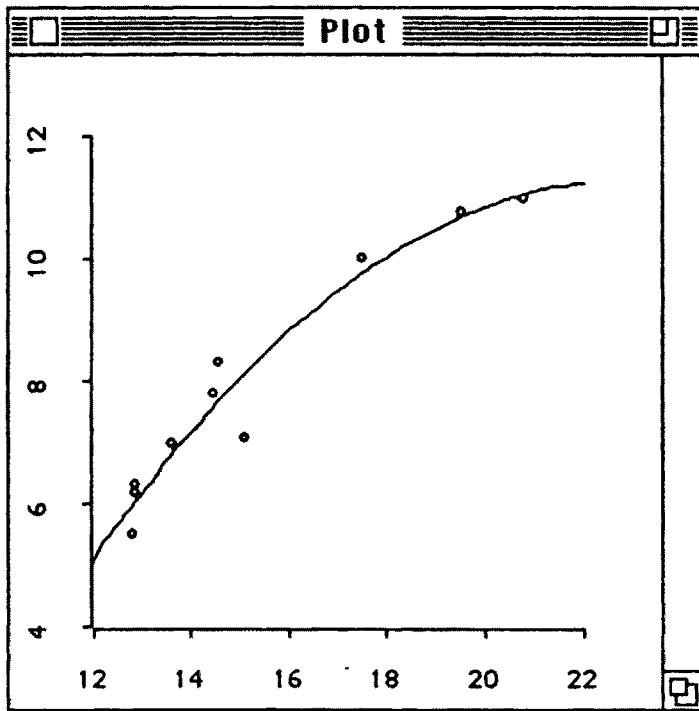


Figure 2.14: Scatterplot of the bicycle data with a fitted curve.

to define `x` as a grid of 50 equally spaced points between 12 and 22, and `y` as the fitted response at these `x` values. Then the expressions

```
(send myplot :clear)
(send myplot :add-points travel-space separation)
(send myplot :add-lines x y)
```

change `myplot` to look like Figure 2.14. We could have used `plot-points` to get a new plot and then modified that plot with `:add-lines`, but the approach used here allowed us to try out all three messages.

2.5.6 Dynamic Simulations

As another illustration of what you can do by modifying existing plots, let's construct a dynamic simulation – a movie – to examine the variation in the shape of histograms of samples from a standard normal distribution. To start off, use the expression

```
(def h (histogram (normal-rand 20)))
```

to construct a single histogram and save its plot object as `h`. The `:clear` message is available for histograms as well. As you can see from its help information,

```
> (send h :help :clear)
:CLEAR
Message args: (&key (draw t))
Clears the plot data. If DRAW is not nil the plot is redrawn;
otherwise its current screen image remains unchanged.
```

it takes an optional keyword argument. If this argument is `nil`, then the plot is not actually redrawn until some other message causes it to be redrawn. This is useful for dynamic simulations. Rearrange and resize your windows until you can see the histogram window while typing instructions into the interpreter. Then type the expression¹⁰

```
(dotimes (i 50)
    (send h :clear :draw nil)
    (send h :add-points (normal-rand 20)))
```

This should produce a sequence of 50 histograms, each based on a sample of size 20. By giving the keyword argument `:draw` with value `nil` to the `:clear` message, you have ensured that each histogram stays on the screen until the next one is ready to be drawn. Try the example again without this argument and see what difference it makes.

Programmed dynamic simulations provide another approach to viewing the relationship among several variables. As a simple example, suppose we want to examine the relation between the variables `abrasion-loss` and `hardness` introduced in Section 2.5.2 above. Let's start with a histogram of `abrasion-loss` produced by the expression

```
(def h (histogram abrasion-loss))
```

The messages `:point-selected`, `:point-hilited`, and `:point-showing` are useful for dynamic simulations. Here is the help information for `:point-selected` in a histogram:

```
> (send h :help :point-selected)
:POINT-SELECTED
Method args: (point &optional selected)
Sets or returns selection status (true or NIL) of POINT.
Sends :ADJUST-SCREEN message if states are set. Vectorized.
```

¹⁰`dotimes` is one of several Lisp looping constructs. It is a special form with the syntax `(dotimes ((var) (count)) (expr))`. The loop is repeated `(count)` times, with `(var)` bound to 0, 1, ..., `(count) - 1`. Other looping constructs are `dolist`, `do`, and `do*`.

Thus you can use this message to determine whether a point is currently selected and also to select or unselect it. Again rearrange the windows so that you can see the histogram while typing into the interpreter, and type the expression

```
(dolist (i (order hardness))
  (send h :point-selected i t)
  (send h :point-selected i nil))
```

The expression `(order hardness)` produces the list of indices of the ordered values of `hardness`. Thus the first element of the result is the index of the smallest element of `hardness`, the second element is the index of the second smallest element of `hardness`, etc.. The loop moves through each of these indices and selects and unselects the corresponding point.

The result on the screen is very similar to the result of brushing a histogram of `hardness` linked to a histogram of `abrasion-loss` from left to right. The drawback of this approach is that it is harder to write an expression than to use a mouse. On the other hand, when brushing with a mouse, you tend to focus your attention on the plot you are brushing rather than on the other linked plots. When you run a dynamic simulation, you do not have to do anything while the simulation is running, and can therefore concentrate fully on the results.

An intermediate solution is possible: you can set up a dialog with a scroll bar that runs through the indices in the list `(order hardness)`, selecting the corresponding point as it is scrolled. An example in Section 2.7 shows how this is done.

In many Lisp systems, simulations like the ones described here will occasionally pause briefly because of the need for *garbage collection* to reclaim dynamically allocated storage. Nevertheless, in simple simulations like these each iteration may still be too fast for you to be able to pick up any pattern. To slow things down, you can add a short delay using the function `pause`. Called with an integer n , `pause` causes execution to be suspended for approximately $n/60$ seconds. For example, you could use

```
(dolist (i (order hardness))
  (send h :point-selected i t)
  (pause 10)
  (send h :point-selected i nil))
```

in place of the previous expression. This ensures that the loop moves through the points at a rate of approximately 6 per second.

Exercise 2.17

The `:add-points` message also allows you to add points to a plot without removing points it already contains. Use this to create a simulation that shows how a histogram changes as more and more observations are added to

it. Start with a histogram of 50 normal random variables, to set the scale of the histogram. Then clear the plot and write a loop to add 100 values to the histogram, five at a time.

Exercise 2.18

Repeat the dynamic simulation for the abrasion loss data using a scatterplot of **abrasion-loss** versus **tensile-strength**.

2.6 Regression

Regression models are implemented using Lisp-Stat's object and message-sending facilities. These were introduced above in Section 2.5.5. You might want to review that section briefly before reading on.

Let's fit a simple regression model to the bicycle data of Section 2.5.5. The dependent variable is **separation**, and the independent variable is **travel-space**. To form a regression model, use the **regression-model** function:

```
> (regression-model travel-space separation)
```

Least Squares Estimates:

Constant	-2.182472	(1.056688)
Variable 0	0.6603419	(0.06747931)

R Squared:	0.922901
Sigma hat:	0.5821083
Number of cases:	10
Degrees of freedom:	8

```
#<Object: 1966006, prototype = REGRESSION-MODEL-PROTO>
```

The basic syntax for the **regression-model** function is

```
(regression-model <x> <y>)
```

For a simple regression **<x>** can be a single list or vector. For a multiple regression **<x>** can be a list of lists, a list of vectors, or a matrix. The **regression-model** function also takes several optional keyword arguments, including **:intercept**, **:print**, and **:weights**. Both **:intercept** and **:print** are **t** by default. To get a model without an intercept, use the expression

```
(regression-model x y :intercept nil)
```

To form a weighted regression model, use the expression

```
(regression-model x y :weights w)
```

where *w* is a list or vector of weights the same length as *y*. The variances of the errors are then assumed to be inversely proportional to the weights *w*.

The **regression-model** function prints a simple summary of the fit model and returns a model object as its result. To be able to examine the model further, assign the returned model object to a variable using an expression like

```
(def bikes
  (regression-model travel-space separation :print nil))
```

The keyword argument :*print* with the value *nil* suppresses the printing of the summary, which we have already seen. To find out what messages are available, use the :*help* message:

```
> (send bikes :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-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 :R-SQUARED :RAW-RESIDUALS :REARENT
: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
```

Many of these messages are self-explanatory, and many have already been used by the :*display* message, which **regression-model** sends to the new model to print the summary. As examples, let's try the :*coef-estimates* and :*coef-standard-errors* messages:

```
> (send bikes :coef-estimates)
(-2.182472 0.6603419)
> (send bikes :coef-standard-errors)
(1.056688 0.06747931)
```

Ordinarily the entries in the lists returned by these messages correspond to the intercept, if one is included in the model, followed by the independent variables as they were supplied to **regression-model**. However, if degeneracy is detected during computation, some variables will not be used in the

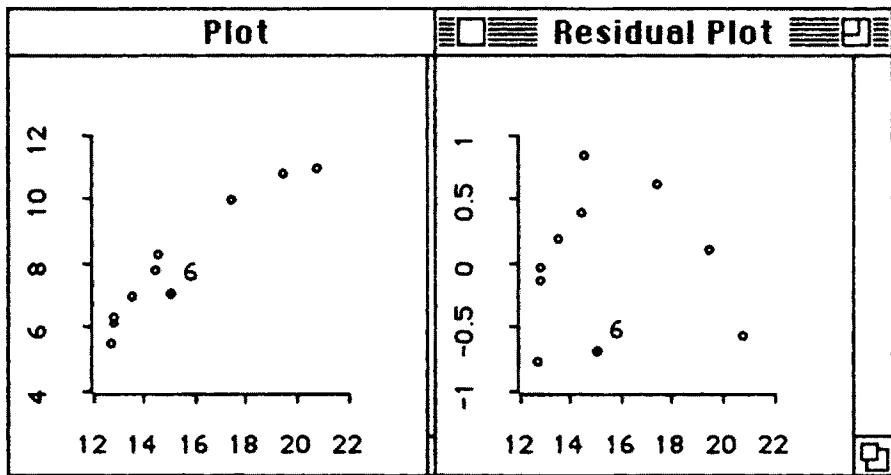


Figure 2.15: Linked raw data and residual plots for the bicycles example.

fit; they will be marked as *aliased* in the printed summary. The indices of the variables used can be obtained by the `:basis` message; the entries in the list returned by `:coef-estimates` correspond to the intercept, if appropriate, followed by the coefficients of the elements in the basis. The messages `:x-matrix` and `:xtxinv` are similar in that they use only the variables in the basis.

The `:plot-residuals` message produces a residual plot. To find out what residuals are plotted against, let's look at the help information:

```
> (send bikes :help :plot-residuals)
:PLOT-RESIDUALS
Message args: (&optional x-values)
Opens a window with a plot of the residuals. If X-VALUES are
not supplied the fitted values are used. Returns a plot object.
```

Using the expressions

```
(plot-points travel-space separation)
```

and

```
(send bikes :plot-residuals travel-space)
```

we can construct two plots of the data as shown in Figure 2.15. By linking the plots, we can use the mouse to identify points in both plots simultaneously. A point that stands out is observation 6 (starting the count at 0, as usual).

Both plots suggest that there is some curvature in the data; this curvature is particularly pronounced in the residual plot if you ignore observation 6 for the moment. To allow for this curvature, we might try to fit a model with a quadratic term in `travel-space`:

```
> (def bikes2
  (regression-model (list travel-space (^ travel-space 2))
                     separation))
```

Least Squares Estimates:

Constant	-16.41924	(7.848271)
Variable 0:	2.432667	(0.9719628)
Variable 1:	-0.05339121	(0.02922567)
 R Squared:	0.9477923	
Sigma hat:	0.5120859	
Number of cases:	10	
Degrees of freedom:	7	

BIKES2

I have used the exponentiation function “ $^$ ” to compute the square of travel-space. Since I am now forming a multiple regression model, the first argument to `regression-model` is a list of the independent variables.

You can proceed in many directions from this point. If you want to calculate Cook’s distances for the observations, you can send `bikes2` the `:cooks-distances` message:

```
> (send bikes2 :cooks-distances)
(0.166673 0.00918596 0.030268 0.011099 0.00958442
 0.120665 0.581929 0.0460179 0.00640447 0.0940081)
```

To check these distances, you can calculate them directly by first computing internally studentized residuals as

```
(def studres (/ (send bikes2 :residuals)
                  (* (send bikes2 :sigma-hat)
                     (sqrt (- 1 (send bikes2 :leverages))))))
```

and then obtaining the distances as¹¹

```
> (* (^ studres 2)
    (/ (send bikes2 :leverages)
        (- 1 (send bikes2 :leverages))
        3))
(0.166673 0.00918596 0.03026801 0.01109897 0.009584418
 0.1206654 0.581929 0.0460179 0.006404474 0.09400811)
```

¹¹The `/` function is used here with three arguments. The first argument is divided by the second, and the result is then divided by the third. Thus the result of the expression `(/ 6 3 2)` is 1.

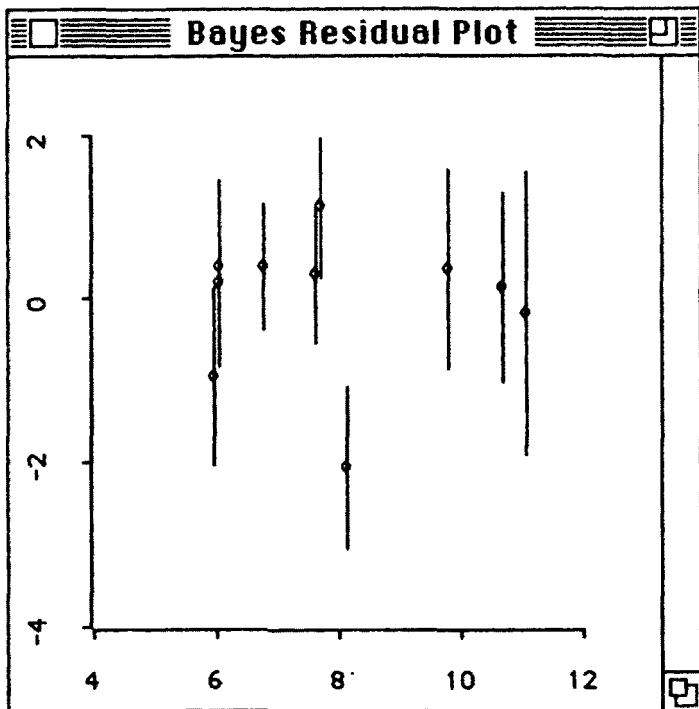


Figure 2.16: Bayesian residual plot for the bicycle data.

The seventh entry – observation 6, counting from zero – clearly stands out.

Another approach to examining residuals for possible outliers is to use the Bayesian residual plot proposed by Chaloner and Brant [17], which can be obtained using the message :plot-bayes-residuals. The expression

```
(send bikes2 :plot-bayes-residuals)
```

produces the plot in Figure 2.16. The bars represent mean $\pm 2SD$ of the posterior distribution of the actual realized errors, based on an improper uniform prior distribution on the regression coefficients. The y axis is in units of $\hat{\sigma}$. Thus this plot suggests the probability that point 6 is three or more standard deviations from the mean is about 3%; the probability that it is at least two standard deviations from the mean is around 50%.

Exercise 2.19

Using the variables `abrasion-loss`, `tensile-strength`, and `hardness` introduced in Section 2.5.2, construct and examine a model with `abrasion-loss` as the dependent variable.

Exercise 2.20

Find an expression for calculating the externally studentized residuals for a regression model. Compare the results produced by your expression to the results returned by the :externally-studentized-residuals message for the `bikes2` model.

2.7 Defining Functions and Methods

This section gives a brief introduction to programming Lisp-Stat. The most basic programming operation is to define a new function. Closely related is the idea of defining a new method for an object.

2.7.1 Defining Functions

The special form used for defining functions is called `defun`. The simplest form of the `defun` syntax is

```
(defun <name> <parameters> <body>)
```

where `<name>` is the symbol used as the function name, `<parameters>` is a list of the symbols naming the arguments, and `<body>` represents one or more expressions that make up the body of the function. Suppose, for example, that you want to define a function to delete a case from a list. This function should take as its arguments the list and the index of the case you want to delete. The body of the function can be based on either of the two approaches described in Section 2.4.3 above. Here is one approach:

```
(defun delete-case (x i)
  (select x (remove i (iseq (length x)))))
```

None of the arguments to `defun` are quoted: `defun` is a special form that does not evaluate its arguments.

You can also write functions that send messages to objects. Here is a function that takes two regression models, assumed to be nested, and computes the F statistic for comparing these models:

```
(defun f-statistic (m1 m2)
  "Args: (m1 m2)
  Computes the F statistic for testing model m1 within model m2."
  (let ((ss1 (send m1 :sum-of-squares))
        (df1 (send m1 :df))
        (ss2 (send m2 :sum-of-squares))
        (df2 (send m2 :df)))
    (/ (/ (- ss1 ss2) (- df1 df2)) (/ ss2 df2))))
```

This definition uses the Lisp `let` construct to establish some local *variable bindings*. The variables `ss1`, `df1`, `ss2`, and `df2` are defined in terms of the two model arguments, `m1` and `m2`, and are then used to compute the F statistic. The string following the argument list is a *documentation string*. When a documentation string is present in a `defun` expression, `defun` installs it so the `help` function can retrieve it.

2.7.2 Functions as Arguments

In Section 2.2.3 we used `plot-function` to plot the sine function over the range $[-\pi, \pi]$. You can use the same approach to plot functions of your own. For example, suppose you would like to plot the function $f(x) = 2x + x^2$ over the range $-2 \leq x \leq 3$. You can do this by defining a function `f` as

```
(defun f (x) (+ (* 2 x) (^ x 2)))
```

and then using `plot-function`:

```
(plot-function #'f -2 3)
```

Recall that the expression `#'f` is short for `(function f)`, and is used for obtaining the Lisp function associated with the symbol `f`.

You can construct a rotating plot of a function of two variables using the function `spin-function`. For example, after defining `f` as

```
(defun f (x y) (+ (^ x 2) (^ y 2)))
```

the expression

```
(spin-function #'f -1 1 -1 1)
```

constructs a plot of the function $f(x, y) = x^2 + y^2$ over the range $-1 \leq x \leq 1$, $-1 \leq y \leq 1$ using a 6×6 grid. The number of grid points can be changed using the `:num-points` keyword. The result is shown in Figure 2.17.

The function `contour-function` takes the same required arguments as `spin-function` and produces a contour plot of the function argument. In addition to the `:num-points` keyword, you can give `contour-function` the `:levels` keyword followed by a list of contour levels to use.

It is rather awkward in these examples to have to define a function `f` just to be able to use it as an argument to another function. It would be more convenient to be able to write an expression that produces the desired function, and use that expression as an argument to `plot-function` or `spin-function`. Section 3.6.1 shows how this can be done.

Exercise 2.21

Construct a rotatable plot and a contour plot of the density of a bivariate normal distribution with means equal to 0, standard deviations equal to 1, and a correlation coefficient of 0.5.

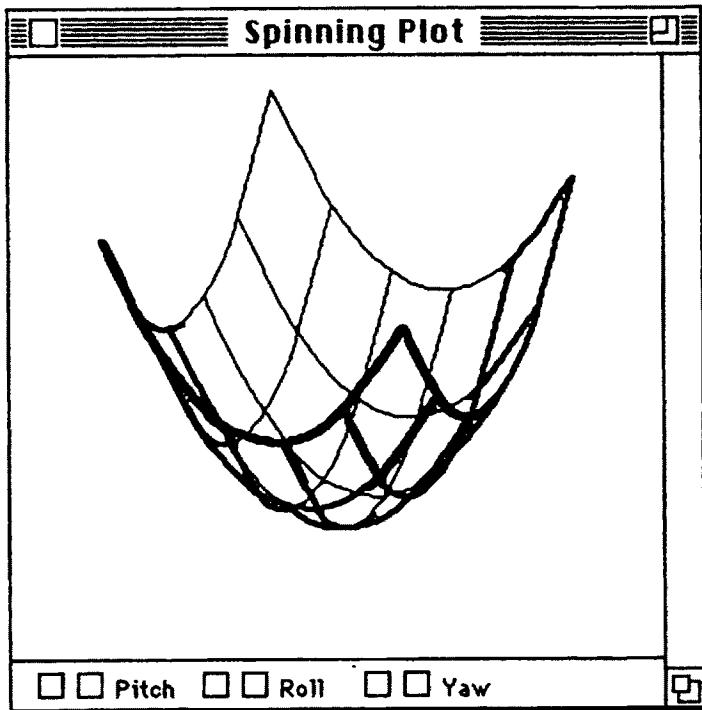


Figure 2.17: Rotatable plot of $f(x, y) = x^2 + y^2$.

2.7.3 Graphical Animation Control

As another illustration of the use of functions, suppose we would like to examine the effect of changing the exponent in a Box-Cox power transformation

$$h(x) = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(x) & \text{otherwise} \end{cases}$$

on a normal probability plot. As a first step, define a function to compute the power transformation and normalize the result to fall between zero and one:

```
(defun bc (x p)
  (let* ((bcx (if (< (abs p) .0001) (log x) (/ (^ x p) p)))
         (min (min bcx))
         (max (max bcx)))
    (/ (- bcx min) (- max min))))
```

This definition uses the `let*` form to establish some local variable bindings. The `let*` form is like the `let` form used above, except that `let*` defines

its variables sequentially, allowing a variable to be defined in terms of other variables already defined in the `let*` expression. In contrast, `let` creates its assignments in parallel. In this case the variables `min` and `max` are defined in terms of the variable `bcx`.

Next, we need a set of positive numbers to transform. Let's use the sample of precipitation values introduced in Section 2.2.1. We need to sort the observations:

```
(def x (sort-data precipitation))
```

The normal quantiles of the expected uniform order statistics are given by

```
(def nq (normal-quant (/ (iseq 1 30) 31)))
```

and a probability plot of the untransformed data is constructed using

```
(def myplot (plot-points nq (bc x 1)))
```

Since the power used is 1, the function `bc` just rescales the data.

To change the power in the transformation shown in `myplot`, we can define a function `change-power` as

```
(defun change-power (p)
  (send myplot :clear :draw nil)
  (send myplot :add-points nq (bc x p)))
```

Evaluating an expression like

```
(change-power .5)
```

redraws the plot for a new power, the square root transformation in this case.

We can repeat this process using a number of different powers to get a sense for the effect of the transformation, but this approach is rather clumsy. An alternative is to set up a *slider* dialog to control the power. A slider is a modeless dialog box containing a scroll bar and a value display field. As the scroll bar is moved, the displayed value is changed and an action is taken. The action is defined by an *action function*. This function is called with a single argument, the current slider value, each time the value is changed by the user. We can construct an appropriate slider for our problem with the expression

```
(sequence-slider-dialog (rseq -1 2 31) :action #'change-power)
```

This slider can be moved through the values $-1.0, -0.8, \dots, 1.8, 2.0$, and calls the `change-power` function with the current power value each time it is moved. The plot and slider are shown in Figure 2.18.

Exercise 2.22

Replace the loops in the animation examples in Section 2.5.6 by sliders.

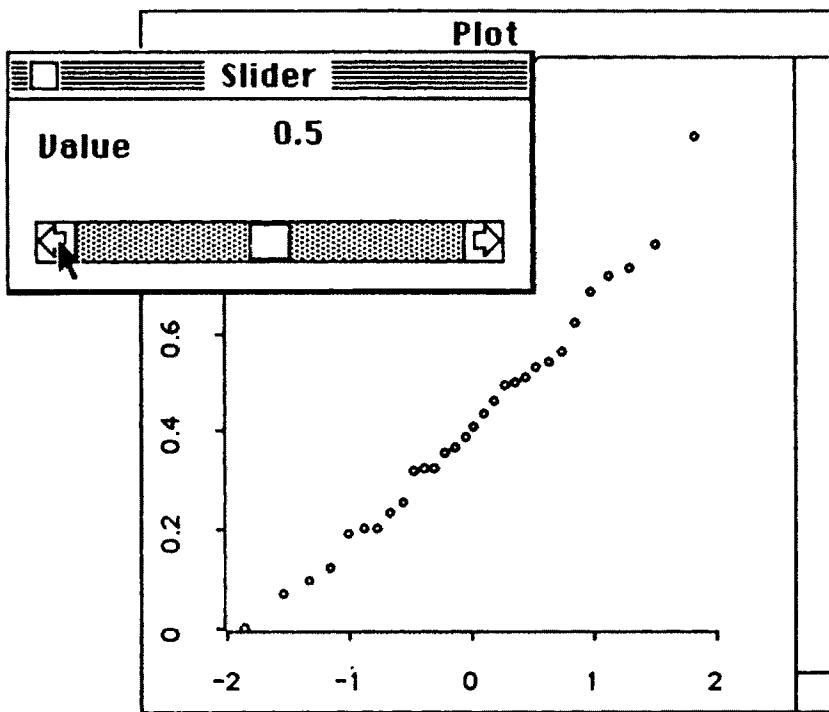


Figure 2.18: A slider-controlled power transformation plot.

2.7.4 Defining Methods

When a message is sent to an object, the object system uses the object and the message selector to find the appropriate piece of code to execute. Different objects may thus respond differently to the same message. Both a linear regression model and a nonlinear regression model might respond to a `:coef-estimates` message, but they will execute different code to compute their responses.

The code used by an object to respond to a message is called a *method*. Objects are organized in a hierarchy in which objects *inherit* from other objects, their ancestors. If an object does not have a method of its own for responding to a message, it uses a method inherited from one of its ancestors. The `send` function moves up the *precedence list* of an object's ancestors until a method for a message is found.

Most of the objects encountered so far inherit directly from *prototype* objects. Scatterplots inherit from `scatterplot-proto`, histograms from `histogram-proto`, and regression models from `regression-model-proto`. Prototypes are just like any other objects. They are essentially *typical* versions of a certain kind of object that define default behavior. Almost all methods are owned by prototypes. But any object can own a method, and

in the process of debugging a new method, it is often better to attach the method to a separate object constructed for that purpose instead of the prototype.

The `defmeth` macro is used to add a method to an object. The general form of a method definition is

```
(defmeth <object> <selector> <parameters> <body>)
```

`<object>` is the object that will own the new method. The argument `<selector>` is the message selector keyword for the method. The argument `<parameters>` is the list of argument names for your method, and `<body>` is one or more expressions making up the body of the method. When the method is used, each of these expressions is evaluated in the order in which they appear.

As a simple illustration, in Section 2.5.6 we used a loop to run a simulation in which a series of samples of size 20 from a normal distribution were displayed in a histogram. The original histogram was constructed by

```
(def h (histogram (normal-rand 20)))
```

On each iteration we first cleared the histogram and then added a new sample. We can simplify this a bit by defining a method for a new message, `:new-sample`, for our histogram as

```
(defmeth h :new-sample ()
  (send self :clear :draw nil)
  (send self :add-points (normal-rand 20)))
```

The loop for running our animation 50 times can now be written as

```
(dotimes (i 50) (send h :new-sample))
```

One point in the definition of the method for `:new-sample` requires explanation: the use of the variable `self`. Methods for objects usually need to be able to refer to the object receiving the message. Since methods can be inherited, you can not be certain of the identity of the receiving object when you write a method. The object system therefore uses the convention that the receiving object is the value of the variable `self` when the expressions in a method for a message are evaluated.

Now that we have a simple message to send our histogram asking it to display a new sample, it would be nice to have a simple way of sending this message that does not require typing instructions into the interpreter. The histogram's menu provides a convenient tool for this purpose, and the expression

```
(let ((m (send h :menu))
      (i (send menu-item-proto :new "New Sample"
              :action #'(lambda () (send h :new-sample)))))
  (send m :append-items i))
```

adds a new item to the end of the histogram's menu that changes the sample displayed each time the item is selected. You should be able to understand this expression after reading Chapters 6 and 7.

2.8 More Models and Techniques

Lisp-Stat provides some tools for nonlinear regression modeling, maximum likelihood estimation, and approximate Bayesian inference. A common feature of these three sets of tools is that part of the information needed to describe a particular problem is a function, for example, a mean function for nonlinear regression. This section assumes that you are familiar with the basics of nonlinear regression, maximum likelihood estimation, and Bayesian inference.

2.8.1 Nonlinear Regression

Lisp-Stat allows the construction of nonlinear, normal regression models. As an example, Bates and Watts [5, A1.3] describe an experiment on the relation between the velocity of an enzymatic reaction, y , and the substrate concentration, x . The concentrations and observed velocities for an experiment in which the enzyme was treated with Puromycin are given by

```
(def x1 (list .02 .02 .06 .06 .11 .11 .22 .22 .56 .56 1.1 1.1))
```

and

```
(def y1 (list 76 47 97 107 123 139 159 152 191 201 207 200))
```

The Michaelis-Menten function

$$\eta(x) = \frac{\theta_0 x}{\theta_1 + x}$$

often provides a good model for the dependence of velocity on substrate concentration. Assuming the Michaelis-Menten function as the mean velocity at a given concentration level, the function `f1` defined by

```
(defun f1 (theta)
  (/ (* (select theta 0) x1) (+ (select theta 1) x1)))
```

computes the list of mean response values at the points in `x1` for a parameter list `theta`.

Using these definitions, we can construct a nonlinear regression model with the `nreg-model` function. First, we need initial estimates for the two model parameters. Examining the expression for the Michaelis-Menten model shows that as x increases, the function approaches an asymptote, θ_0 . The second parameter, θ_1 , can be interpreted as the value of x at which the function has reached half its asymptotic value. Using these interpretations for the parameters, and a plot constructed by the expression

```
(plot-points x1 y1)
```

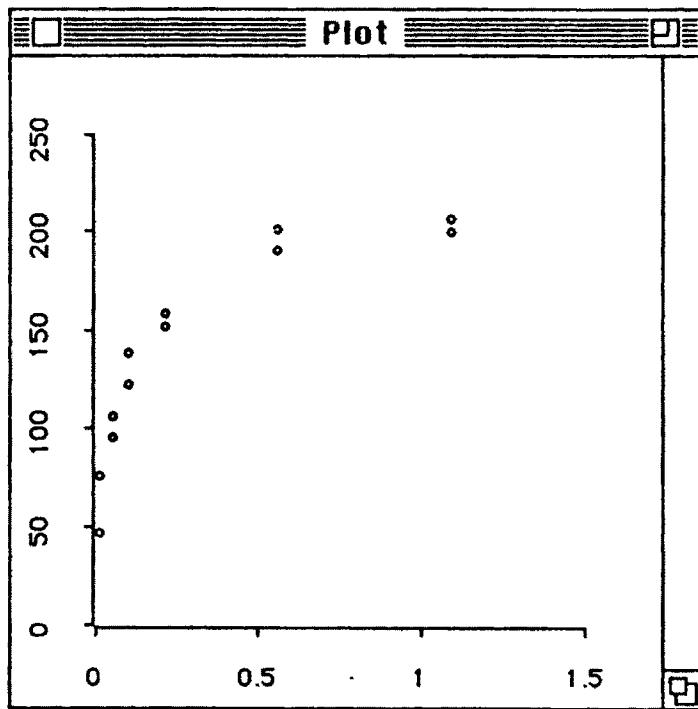


Figure 2.19: Plot of reaction velocity against substrate concentration for the Puromycin experiment.

shown in Figure 2.19, we can read off reasonable initial estimates of 200 for θ_0 and 0.1 for θ_1 . The arguments required by `nreg-model` are the mean function, the response vector, and a list of initial estimates of the parameters. `nreg-model` computes more accurate estimates using an iterative algorithm starting from the initial guess, prints a summary of the results, and returns a nonlinear regression model object:

```
> (def puromycin (nreg-model #'f1 y1 (list 200 .1)))
Residual sum of squares:    7964.19
Residual sum of squares:    1593.16
Residual sum of squares:    1201.03
Residual sum of squares:    1195.51
Residual sum of squares:    1195.45
Least Squares Estimates:
```

Parameter 0	212.684	(6.94715)
Parameter 1	0.0641213	(0.00828094)
R Squared:	0.961261	
Sigma hat:	10.9337	
Number of cases:	12	
Degrees of freedom:	10	

PUROMYCIN

The function `nreg-model` also takes several keyword arguments, including `:epsilon`, to specify a convergence criterion, and `:count-limit`, a limit on the number of iterations. By default these values are .0001 and 20, respectively. Supplying the `:verbose` keyword with value `nil` suppresses printing the residual sum of squares information on each iteration, and the `:print` keyword with value `nil` suppresses the summary. The algorithm for fitting the model is a simple Gauss-Newton algorithm with backtracking; derivatives are computed numerically.

To see how you can analyze the model further, you can send `puromycin` the `:help` message. The result is very similar to the help information for a linear regression model. The reason is simple: nonlinear regression models are implemented as objects, with the nonlinear regression model prototype `nreg-model-proto` inheriting from the linear regression model prototype. The internal data, the method for computing estimates, and the method of computing fitted values have been modified for nonlinear models, but most other methods remain unchanged. Once the model has been fit, the Jacobian of the mean function at the estimated parameter values is used as the X matrix. In terms of this definition, most of the methods for linear regression, such as the methods `:coef-standard-errors` and `:leverages`, still make sense, at least as first order asymptotic approximations.

In addition to the messages for linear regression models, a nonlinear regression model can respond to the messages

- `:COUNT-LIMIT`
- `:EPSILON`
- `:MEAN-FUNCTION`
- `:NEW-INITIAL-GUESS`
- `:PARAMETER-NAMES`
- `:THETA-HAT`
- `:VERBOSE`

Exercise 2.23

Examine the residuals of the `puromycin` model.

Exercise 2.24

The experiment described above was also conducted without the Puromycin treatment using concentrations of

```
(def x2 (list .02 .02 .06 .06 .11 .11 .22 .22 .56 .56 1.1))
```

The resulting observed velocities were

```
(def y2 (list 67 51 84 86 98 115 131 124 144 158 160))
```

Fit a Michaelis-Menten model to these data, and compare the results to the experiment with Puromycin.

2.8.2 Maximization and Maximum Likelihood Estimation

Lisp-Stat includes two functions for maximizing functions of several variables. The first is `newtonmax`, which takes a function and a list or vector representing an initial guess for the location of the maximum, and attempts to find the maximum using an algorithm based on Newton's method with backtracking. The algorithm is based on the unconstrained minimization system described in Dennis and Schnabel [25].

As an example, Proschan [51] describes data collected on times (in operating hours) between failures of air-conditioning units on several aircraft. The data for one of the aircraft can be entered as

```
(def x (list 90 10 60 186 61 49 14 24 56 20 79 84 44 59 29 118
      25 156 310 76 26 44 23 62 130 208 70 101 208))
```

A simple model for these data assumes that the times between failures are independent random variables with a common gamma distribution. The density of the gamma distribution can be written as

$$\frac{(\beta/\mu)(\beta x/\mu)^{\beta-1} e^{-\beta x/\mu}}{\Gamma(\beta)}$$

where μ is the mean time between failures and β is the gamma shape parameter. Thus the log likelihood for the sample is given by

$$n[\log(\beta) - \log(\mu) - \log(\Gamma(\beta))] + \sum_{i=1}^n (\beta - 1) \log\left(\frac{\beta x_i}{\mu}\right) - \sum_{i=1}^n \frac{\beta x_i}{\mu}.$$

We can define a Lisp function to evaluate this log likelihood by

```
(defun gllik (theta)
  (let* ((mu (select theta 0))
         (beta (select theta 1))
         (n (length x)))
```

```
(bym (* x (/ beta mu)))
(+ (* n (- (log beta) (log mu) (log-gamma beta)))
  (sum (* (- beta 1) (log bym))))
  (sum (- bym))))
```

This definition uses the function `log-gamma` to evaluate $\log(\Gamma(\beta))$.

Closed form maximum likelihood estimates are not available for the shape parameter of this distribution, but we can use `newtonmax` to compute estimates numerically.¹² We need an initial guess to use as a starting value in the maximization. To get initial estimates, we can compute the mean and standard deviation of `x`

```
> (mean x)
83.5172
> (standard-deviation x)
70.8059
```

and use method of moments estimates $\hat{\mu} = 83.52$ and $\hat{\beta} = (\hat{\mu}/\hat{\sigma})^2$, calculated as

```
> (^ (/ (mean x) (standard-deviation x)) 2)
1.39128
```

Using these starting values we can maximize the log likelihood function:

```
> (newtonmax #'gllik (list 83.5 1.4))
maximizing...
Iteration 0.
Criterion value = -155.603
Iteration 1.
Criterion value = -155.354
Iteration 2.
Criterion value = -155.347
Iteration 3.
Criterion value = -155.347
Reason for termination: gradient size is less than
gradient tolerance.
(83.5173 1.67099)
```

Some status information is printed as the optimization proceeds. You can turn this off by supplying the keyword argument :`verbose` with value `nil`.

You might want to check that the gradient of the function is indeed close to zero. If you do not have a closed form expression for the gradient, you can use `numgrad` to calculate a numerical approximation. For our example,

¹²The maximizing value for μ is always the sample mean. We could take advantage of this fact and reduce the problem to a one dimensional maximization problem, but it is simpler to just maximize over both parameters.

```
> (numgrad #'gllik (list 83.5173 1.67099))
(-4.07269e-07 -1.25755e-05)
```

The elements of the gradient are indeed quite close to zero.

You can also compute the second derivative, or Hessian, matrix using **numhess**. Approximate standard errors of the maximum likelihood estimates are given by the square roots of the diagonal entries of the inverse of the negative Hessian matrix evaluated at the maximum:

```
> (sqrt
  (diagonal
   (inverse (- (numhess #'gllik (list 83.5173 1.67099))))))
(11.9976 0.402648)
```

Instead of calculating the maximum using **newtonmax** and then calculating the derivatives separately, you can have **newtonmax** return a list of the location of the maximum, the optimal function value, the gradient, and the Hessian by supplying the keyword argument **:return-derivs** as t.¹³

Newton's method assumes a function is twice continuously differentiable. If your function is not smooth, or if you are having trouble with **newtonmax** for some other reason, you might try a second maximization function, the function **nelmeadmax**. **nelmeadmax** takes a function and an initial guess and attempts to find the maximum using the Nelder-Mead simplex method as described in Press, Flannery, Teukolsky, and Vetterling [50]. The initial guess can consist of a single point, represented by a list or vector of n numbers. The initial guess can also be a simplex, a list of $n + 1$ points for an n -dimensional problem. If you specify a single point you should also use the keyword argument **:size** to specify as a list or vector of length n the size in each dimension of the initial simplex. This should represent the size of an initial range in which the algorithm is to start its search for a maximum. We can use this method in our gamma example:

```
> (nelmeadmax #'gllik (list 83.5 1.4) :size (list 5 .2))
Value = -155.603
Value = -155.603
Value = -155.603
Value = -155.587
Value = -155.53
Value = -155.522
...
Value = -155.347
```

¹³The function **newtonmax** ordinarily uses numerical derivatives in its computations. Occasionally this may not be accurate enough or may take too long. If you have an expression for computing the gradient or the Hessian, then you can use these by having your function return a list of the function value and the gradient, or a list of the function value, the gradient, and the Hessian matrix, instead of just returning the function value.

```
Value = -155.347
Value = -155.347
Value = -155.347
(83.5181 1.6709)
```

It takes somewhat longer than Newton's method but it does reach the same result.

Exercise 2.25

Use maximum likelihood estimation to fit a Weibull model to the data used in this subsection. For this model, compare exact derivatives and numerical derivatives of the log likelihood at the maximum likelihood estimates.

2.8.3 Approximate Bayesian Computations

The functions described in this subsection can be used to compute first and second order approximations to posterior moments, and saddlepoint-like approximations to one-dimensional marginal posterior densities. The approximations, based primarily on the results in [62,63,64], assume that the posterior density is smooth and dominated by a single mode.

Let's start with a simple example, a data set used to study the relation between survival time, in weeks, of leukemia patients and white blood cell counts recorded for the patients at their entry into the study [29]. The data consist of two groups of patients classified as AG positive and AG negative. The data for the 17 AG positive patients can be entered as¹⁴

```
(def wbc-pos (list 2300 750 4300 2600 6000 10500 10000 17000
                  5400 7000 9400 32000 35000 100000 100000
                  52000 100000))
```

and

```
(def times-pos (list 65 156 100 134 16 108 121 4 39 143 56
                  26 22 1 1 5 65))
```

A high white blood cell count indicates a more serious stage of the disease, and thus a lower chance of survival.

A model often used for these data assumes that the survival times are exponentially distributed, with a mean that is log linear in the logarithm of the white blood cell count. For convenience I will scale the white blood cell counts by 10,000. That is, the mean survival time for a patient with white blood cell count WBC_i is

$$\mu_i = \theta_0 \exp\{-\theta_1 x_i\}$$

¹⁴Reproduced from: P. Feigl and M. Zelen, "Estimation of Exponential Survival Probabilities with Concomitant Information". *Biometrics* 21: 826-838. 1965. With permission from The Biometric Society.

with $x_i = \log(WBC_i/10,000)$. The log likelihood function is thus given by

$$\sum_{i=1}^n \theta_1 x_i - n \log(\theta_0) - \frac{1}{\theta_0} \sum_{i=1}^n y_i e^{\theta_1 x_i}$$

with y_i representing the survival times. After computing the transformed WBC variable as

```
(def transformed-wbc-pos (- (log wbc-pos) (log 10000)))
```

the log likelihood can be computed using the function

```
(defun llik-pos (theta)
  (let* ((x transformed-wbc-pos)
         (y times-pos)
         (theta0 (select theta 0))
         (theta1 (select theta 1))
         (t1x (* theta1 x)))
    (- (sum t1x)
        (* (length x) (log theta0))
        (/ (sum (* y (exp t1x)))
            theta0))))
```

I will look at this problem using a vague, improper prior distribution that is constant over the range $\theta_i > 0$; thus the log posterior density is equal to the log likelihood constructed above, up to an additive constant. The first step is to construct a Bayes model object using the function `bayes-model`. This function takes a function for computing the log posterior density and an initial guess for the posterior mode, computes the posterior mode by an iterative method starting with the initial guess, prints a short summary of the information in the posterior distribution, and returns a model object. We can use the function `llik-pos` to compute the log posterior density, so all we need is an initial estimate for the posterior mode. Since the model we are using assumes a linear relationship between the logarithm of the mean survival time and the transformed WBC variable, a linear regression of the logarithms of the survival times on `transformed-wbc-pos` should provide reasonable estimates. The linear regression gives

```
> (regression-model transformed-wbc-pos (log times-pos))
```

Least Squares Estimates:

Constant	3.54234	(0.302699)
Variable 0	-0.817801	(0.214047)

R Squared: 0.4932

Sigma hat: 1.23274

Number of cases:	17
Degrees of freedom:	15

so reasonable initial estimates of the mode are $\hat{\theta}_0 = \exp(3.5)$ and $\hat{\theta}_1 = 0.8$. Now we can use these estimates in the `bayes-model` function:

```
> (def lk (bayes-model #'llik-pos (list (exp 3.5) .8)))
maximizing...
Iteration 0.
Criterion value = -90.8662
Iteration 1.
Criterion value = -85.4065
Iteration 2.
Criterion value = -84.0944
Iteration 3.
Criterion value = -83.8882
Iteration 4.
Criterion value = -83.8774
Iteration 5.
Criterion value = -83.8774
Iteration 6.
Criterion value = -83.8774
Reason for termination: gradient size is less than
gradient tolerance.
```

First Order Approximations to Posterior Moments:

Parameter 0	56.8489 (13.9713)
Parameter 1	0.481829 (0.179694)

```
#<Object: 1565592, prototype = BAYES-MODEL-PROTO>
>
```

It is possible to suppress the summary information by supplying `nil` as the value of the `:print` keyword argument. The iteration information can be suppressed using the `:verbose` keyword.

The summary printed by `bayes-model` gives first order approximations to the posterior means and standard deviations of the parameters. That is, the means are approximated by the elements of the posterior mode, and the standard deviations by the square roots of the diagonal elements of the inverse of the negative Hessian matrix of the log posterior at the mode. These approximations can also be obtained from the model by sending it the `:1stmoments` message:

```
> (send lk :1stmoments)
((56.8489 0.481829) (13.9713 0.179694))
```

The result is a list of two lists, the means and the standard deviations.

A slower but generally more accurate second order approximation is available as well. It can be obtained using the message :moments:

```
> (send lk :moments)
((65.3085 0.485295) (17.158 0.186587))
```

Both of these messages allow you to compute moments for individual parameters or groups of parameters by specifying an individual parameter index or a list of indices:

```
> (send lk :moments 0)
((65.3085) (17.158))
> (send lk :moments (list 0 1))
((65.3085 0.485295) (17.158 0.186587))
```

The first and second order approximations to the moments of θ_0 are somewhat different; in particular, the mean appears to be somewhat larger than the mode. This suggests that the posterior distribution of this parameter is skewed to the right. We can confirm this by looking at a plot of the approximate marginal posterior density. The message :margin1 takes a parameter index, and a sequence of points at which to evaluate the density, and returns as its value a list of the supplied sequence and the approximate density values at these points. This list can be given to plot-lines to produce a plot of the marginal density:

```
> (plot-lines (send lk :margin1 0 (rseq 30 120 30)))
#<Object: 1623804, prototype = SCATTERPLOT-PROTO>
```

The result is shown in Figure 2.20 and does indeed show some skewness to the right.

In addition to examining individual parameters, it is also possible to look at the posterior distribution of smooth functions of the parameters.¹⁵ For example, you might want to ask what information the data contains about the probability of a patient with $WBC = 50,000$ surviving a year or more. This probability is given by

$$\frac{1}{\mu(x)} e^{-52/\mu(x)}$$

with

$$\mu(x) = \theta_0 e^{-\theta_1 x}$$

and $x = \log(5)$, and can be computed by the function

¹⁵The approximation methods assume these functions are twice continuously differentiable; thus they cannot be indicator functions.

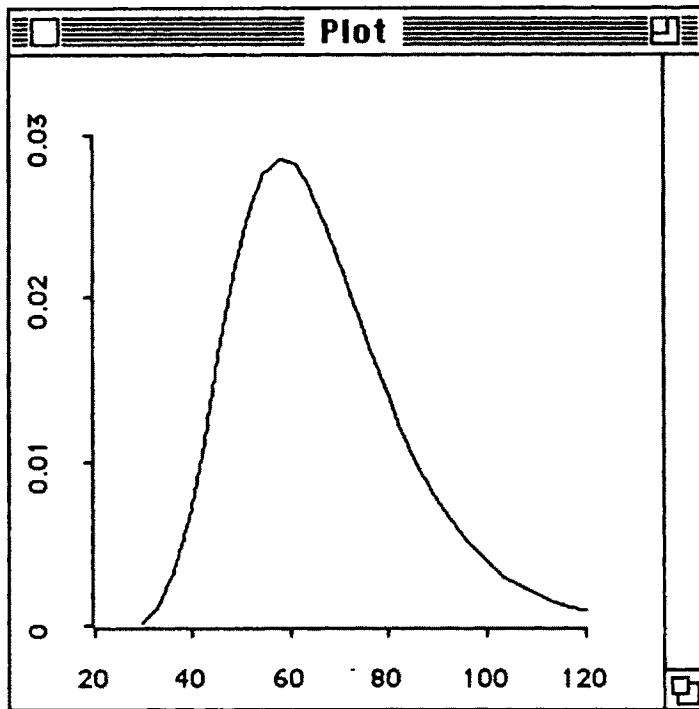


Figure 2.20: Approximate marginal posterior density for θ_0 .

```
(defun lk-sprob (theta)
  (let* ((time 52.0)
         (x (log 5))
         (mu (* (select theta 0)
                 (exp (- (* (select theta 1) x)))))))
    (exp (- (/ time mu)))))
```

This function can be given to the :*1stmoments*, :*moments*, and :*margin1* methods to approximate the posterior moments and marginal posterior density of this function. For the moments the results are

```
> (send lk :1stmoments #'lk-sprob)
((0.137189) (0.0948248))
> (send lk :moments #'lk-sprob)
((0.184275) (0.111182))
```

with the difference again suggesting some positive skewness. The marginal density produced by the expression

```
(plot-lines (send lk :margin1 #'lk-sprob (rseq .01 .8 30)))
```

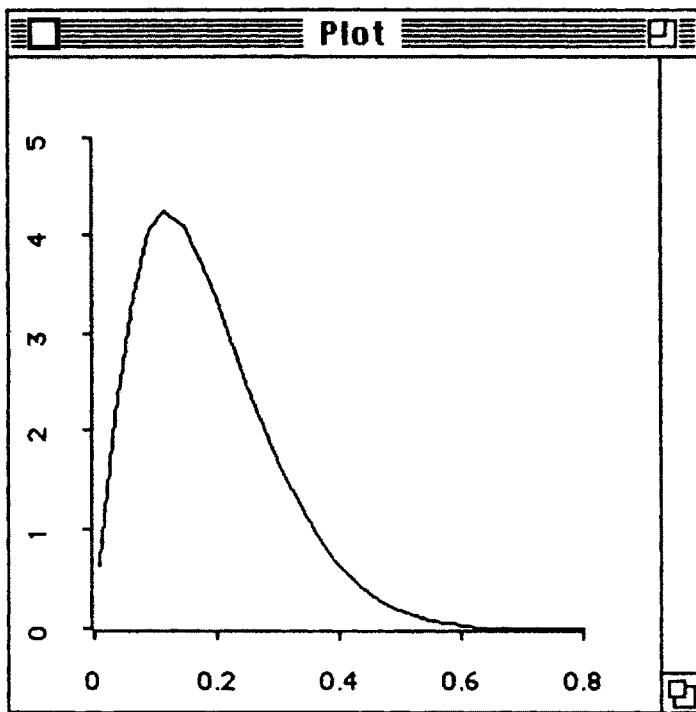


Figure 2.21: Approximate marginal posterior density of the one-year survival probability of a patient with $WBC = 50,000$.

is shown in Figure 2.21. Based on this picture, the data suggests that this survival probability is almost certainly below 0.5, but it is difficult to make a more precise statement than that.

The functions described in this subsection are based on the optimization code described in the previous subsection. By default, derivatives are computed numerically. If you can compute derivatives yourself, you can have your log posterior function return a list of the function value and the gradient, or a list of the function value, the gradient, and the Hessian matrix. Using exact derivatives may be necessary for second order approximations in larger problems.

Exercise 2.26

To be able to think about prior distributions for the two parameters in this problem, we need to understand what the parameters represent. The parameter θ_0 is fairly easy to understand: it is the mean survival time for patients with $WBC = 10,000$. The parameter θ_1 is a little more difficult to think about. It represents the approximate percent difference in mean survival

time for patients with WBC differing by 1%. Because of the minus sign in the mean relationship, and the expected inverse relation between WBC and mean survival time, θ_1 is expected to be positive.

Consider an informative prior distribution that assumes the two parameters to be independent *a priori*, takes $\log(\theta_0)$ to be normally distributed with mean $\log(52)$ and standard deviation $\log(2)$, and takes θ_1 to be exponentially distributed with mean $\mu = 5$. This prior is designed to represent an opinion that mean survival time at $WBC = 10,000$ should be around one year, but that guess could easily be off by a factor of two either way. The percentage change in the mean for a 1% change in WBC should be on the order of 1% to 10% or so. Examine the posterior distribution for this prior, and compare your results to the results for the vague prior used above.

Exercise 2.27

Construct and examine a posterior distribution for the parameters of the gamma model based on the aircraft data of Section 2.8.2.

Chapter 3

Programming in Lisp

In the preceding chapter we used a number of built-in Lisp and Lisp-Stat functions to perform some interesting computations. Some of the expressions we constructed were rather complicated. When you find yourself typing the same expression several times, perhaps with slight variations in the data you use, it is natural to want to introduce some shorthand forms for these expressions – you will want to define your own functions. Function definition is the basic operation in Lisp programming. Section 2.7 has already given a brief introduction to this topic, and it is now time to explore it more thoroughly. After a brief review of how to define a Lisp function, we will examine some of the techniques needed to develop more powerful functions: conditional evaluation, recursion and iteration, local variables, functional data, mapping, and assignment.

In addition to introducing the tools for defining functions, this chapter also presents some of the programming techniques and principles that are often used in Lisp programming. In particular, for most of this chapter I will use a *functional* programming style, avoiding the use of assignment to change the values of variables until the end of the chapter. The development of this chapter is based heavily on the first two chapters of Abelson and Sussman [1]. To allow us to concentrate on the programming process itself, this chapter will use only the basic forms of data we have already seen: numbers, strings, and lists. Lisp and Lisp-Stat offer a number of additional data types, and functions for operating on these data types, that can be used as building blocks in developing your own functions. I will introduce some of them in the following chapters.

3.1 Writing Simple Functions

Lisp functions are defined using the special form `defun`.¹ The basic syntax for `defun` is

```
(defun <name> <parameters> <body>)
```

where `<name>` is the symbol used to refer to the function, `<parameters>` is a list of symbols used to name the formal parameters of the function, and `<body>` consists of one or more expressions. When a function is called, the expressions in the body are evaluated in sequence, and the value obtained by evaluating the last expression is returned as the result of the function. We will not use functions with more than one expression in their bodies until later. None of the arguments to `defun` should be quoted since `defun` does not evaluate its arguments.

As an example, here is a function that computes the sum of squares of a data set:

```
(defun sum-of-squares (x) (sum (* x x)))
```

We can apply this new function to a list of numbers, just like any of the functions used in Chapter 2:

```
> (sum-of-squares (list 1 2 3))
14
```

This definition makes use of the vectorized arithmetic facility provided in Lisp-Stat. The function `sum` computes the sum of the elements of its argument.

A function can be defined to take several arguments. For example, a function to compute an inner product can be defined as

```
(defun my-inner-product (x y) (sum (* x y)))
```

I have called this function `my-inner-product`, instead of just `inner-product`, in order not to lose the definition of Lisp-Stat's own `inner-product` function, which is a bit more elaborate than the one defined here.

Functions defined using `defun` can be used to define other functions. For example, using `sum-of-squares`, we can define a function for computing the Euclidean distance between two data lists:

```
(defun distance (x y) (sqrt (sum-of-squares (- x y))))
```

Using `defun`, we can develop functions as shorthand notation for complex expressions. We are, however, still limited in the power of the expressions

¹In Common Lisp `defun` is a macro rather than a true special form, but again I will ignore this distinction to keep the discussion simple.

we can use. Suppose we would like to convert the definition of the absolute value given by

$$|x| = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

into a Lisp function. To write this function, we need some *predicates* to compare a number to zero, and a *conditional evaluation* construct to evaluate different expressions according to the results returned by the predicates.

3.2 Predicates and Logical Expressions

Predicates are functions used to determine if a condition is true or not. They return `nil` for false and some non-`nil` value, often the symbol `t`, for true. For comparing numbers, we can use the predicates `<`, `>`, `=`, and other comparison predicates. All three of these predicates take two or more arguments. The predicate `<` returns true if all its arguments are in increasing order:

```
> (< 1 2)
T
> (< 2 1)
NIL
> (< 1 2 3)
T
> (< 1 3 2)
NIL
```

The predicate `>` is similar. The predicate `=` returns true if all its arguments are equal:

```
> (= 1 2)
NIL
> (= 1 1)
T
> (= 1 1 1)
T
```

In Lisp-Stat the comparison predicates are vectorized. Noncompound arguments to `>` and `<` must be real numbers; for `=` they may be real or complex numbers. Otherwise, an error is signaled.

The special forms `and` and `or` and the function `not` can be used to build up more complex predicates in terms of simpler ones. As an example, we can define a function to test whether a number is in the interval $(3, 5]$ as

```
(defun in-range (x) (and (< 3 x) (<= x 5)))
```

The special form `and` takes two or more arguments, and evaluates them one at a time until all are found to be true or one is found that is false. Once an argument is found that evaluates to false, no further arguments are evaluated.

To test whether a number is *not* in the interval (3, 5], we can use the special form `or`:

```
(defun not-in-range (x) (or (>= 3 x) (> x 5)))
```

Like `and`, the special form `or` takes two or more arguments, and evaluates them one at a time until one is found to be true, or all are found to be false.

Another way to define `not-in-range` is to use the function `in-range` together with the function `not`:

```
(defun not-in-range (x) (not (in-range x)))
```

Exercise 3.1

Define a predicate that tests whether its argument is a non-negative real number. The predicate should not produce an error if called with a complex argument. The predicates `numberp` and `complexp` can be used to test whether a Lisp item is a number of any kind, or a complex number, respectively.

3.3 Conditional Evaluation

The basic Lisp conditional evaluation construct is `cond`. Using `cond` and the comparison predicates just introduced, we can define a function for computing the absolute value of a number as

```
(defun my-abs (x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

The general form of a `cond` expression is

```
(cond ((p 1) (e 1))
      ((p 2) (e 2))
      ...
      ((p n) (e n)))
```

The lists $(\langle p \rangle \langle e \rangle)$, etc., are called *cond clauses*. The first expressions in each of the `cond` clauses, $\langle p_1 \rangle, \dots, \langle p_n \rangle$, are *predicate expressions* that should evaluate to `nil` for false or any non-`nil` value for true. `cond` works through the clauses one at a time, evaluating predicate expressions until one evaluates to true. If one of the predicates is true, say, $\langle p_i \rangle$, then the corresponding consequent expression $\langle e_i \rangle$ is evaluated, and the result is returned as the result of the `cond` expression. If none of the predicates is true, `nil` is returned.

In the definition of `my-abs` I have used full predicate expressions for each of the three cond clauses. Since the last cond clause is a default, I could have used the symbol `t` as the predicate expression. The definition of `my-abs` would then look like

```
(defun my-abs (x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        (t (- x))))
```

Since the predicate of the final clause is always true, it will be used if none of the preceding clauses apply.

Cond clauses can have several consequent expressions. These expressions are evaluated in sequence, and the result of the final evaluation is returned. The expressions before the final one are only useful for the side effects they produce. For example, we can use the functions `print` and `terpri` to modify our definition of `my-abs` to print a message indicating which cond clause is being used. `print` sends a *newline* followed by a printed representation of its argument and a space to the interpreter; `terpri` sends the interpreter a *newline*. The new definition is

```
(defun my-abs (x)
  (cond ((> x 0) (print 'first-clause) (terpri) x)
        ((= x 0) (print 'second-clause) (terpri) 0)
        ((< x 0) (print 'third-clause) (terpri) (- x))))
```

This provides some information about the evaluation process when the function is used:

```
> (my-abs 3)
```

FIRST-CLAUSE

3

```
> (my-abs -3)
```

THIRD-CLAUSE

3

This use of `print` is of course rather silly in this particular example, but it can be a useful debugging tool in more complicated functions.

In addition to `cond`, Lisp offers several other conditional evaluation constructs. The most important of these is `if`. The special form `if` takes three expressions, a *predicate expression*, a *consequent*, and an *alternative*. The predicate expression is evaluated first. If it is true, then the consequent is evaluated, and its result is returned as the result of the `if` expression. Otherwise, the alternative expression is evaluated and its result returned. If the optional alternative expression is not supplied, it defaults to `nil`. Using `if`, we can define `my-abs` as

```
(defun my-abs (x) (if (> x 0) x (- x)))
```

An if expression of the form

```
(if <predicate> <consequent> <alternative>)
```

is equivalent to the cond expression

```
(cond <(<predicate> <consequent>)>
      <(<t> <alternative>)>))
```

Exercise 3.2

Use if to define a Lisp function to compute

$$f(x) = \begin{cases} \frac{\sin(x)}{x} & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$$

The predicate for testing whether numbers are not equal is `/=`. Why is it important in this problem that if is a special form, not a function?

3.4 Iteration and Recursion

Lisp is a recursive language. As a result, at this point we already have enough tools to write functions for performing numerical computations that require specialized iteration constructs in other languages. Lisp does provide several iteration tools, and we will look at some of them below, but it is useful to examine first how much can be accomplished using only recursion.

As an example,² let's look at an algorithm for calculating the square root of a number x . The algorithm starts with an initial guess y for \sqrt{x} , and calculates an improved guess as

$$\frac{1}{2}(y + \frac{x}{y})$$

This process is repeated until the square of the current guess is close enough to x . For example, for computing $\sqrt{3}$ starting with an initial guess of 1, we

²Adapted from Abelson and Sussman [1, Section 1.1.7].

would use the steps

Current Guess	Improved Guess
1	$\frac{1}{2}(1 + \frac{3}{1}) = 2$
2	$\frac{1}{2}(2 + \frac{3}{2}) = 1.75$
1.75	$\frac{1}{2}(1.75 + \frac{3}{1.75}) = 1.73214$
1.73214	$\frac{1}{2}(1.73214 + \frac{3}{1.73214}) = 1.73205$
...	...

This algorithm, which dates back to the first century A.D., is a special case of Newton's method for finding roots of an equation. We can express this algorithm in Lisp by defining a function `sqrt-iter` that takes an initial guess and the number x , and computes the approximate square root:

```
(defun sqrt-iter (guess x)
  (if (good-enough-p guess x)
      guess
      (sqrt-iter (improve guess x) x)))
```

If the current guess is satisfactory, it is returned. Otherwise, the computation is repeated with an improved guess.

This function requires two additional functions, `improve` for calculating a new guess, and `good-enough-p` for testing whether the current guess is close enough.³ The improvement function simply encodes the step given above, and can be defined as

```
(defun improve (guess x) (mean (list guess (/ x guess))))
```

The convergence test requires the choice of a convergence criterion and a cutoff value. A simple definition is

```
(defun good-enough-p (guess x)
  (< (abs (- (* guess guess) x)) .001))
```

The choice of .001 as the cutoff value is of course arbitrary.

Using `sqrt-iter`, we can now define a square root function as

```
(defun my-sqrt (x) (sqrt-iter 1 x))
```

and try it on an example:

```
> (my-sqrt 9)
3.00009
```

³A convention used by many Lisp programmers is to give predicate functions names ending in `p` or `-p`.

The definition of the function `sqrt-iter` is recursive – that is, it is defined in terms of itself. But there is a difference between the *computational process* generated by this function and the process generated, for example, by the following recursive function for evaluating the factorial of a positive integer:

```
(defun factorial (n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Here the factorial is defined as 1 for $n = 1$; for $n > 1$ it is computed as $n(n - 1)!$. When this factorial function is applied to an argument, it builds up a series of deferred computations until it finally reaches the base case and can complete the multiplication. The amount of space required to keep track of these deferred computations is linear in the number of steps in the process. Such a computational process is called *linearly recursive*.

In contrast, the computational process generated by `sqrt-iter` does not need to keep track of any deferred computations. All information about the current state of the process is contained in a single *state variable*, the current guess. A computational process that can be described in terms of a fixed number of state variables is called an *iterative process*.⁴

It is possible to define an iterative version of the factorial function as well. To do this, we can use a counter and a product as the state variables. At each step the product is multiplied by the counter, and the counter is incremented. This continues until the counter exceeds n :

```
(defun fact-iter (product counter n)
  (if (> counter n)
      product
      (fact-iter (* counter product) (+ counter 1) n)))
```

We can now define `factorial` using `fact-iter` with the product and counter started at 1:

```
(defun factorial (n) (fact-iter 1 1 n))
```

The two helper functions `sqrt-iter` and `fact-iter` are very similar in structure. Since defining a function that generates an iterative computational

⁴Even though the process generated by `sqrt-iter` is iterative when viewed at the Lisp level, for many older Lisp systems the overall process is linearly recursive. The reason is that these systems keep track of the sequence of calls of `sqrt-iter`. In fact, this is not necessary, and a technique called *tail recursion* can be used to make the entire computational process iterative, using only a fixed amount of space. Many newer Lisp systems recognize tail recursions and are called *tail recursive*. One way to tell if a system is in fact tail recursive is to write an infinite loop as `(defun f () (f))`. In a tail recursive system the process generated by calling this function will run indefinitely. If a system is not tail recursive, it will attempt to keep track of each call to `f` and will eventually run out of space (and possibly crash the Lisp system).

process often requires defining a helper function with this structure, Lisp provides a special form that simplifies this process, and avoids the need to come up with a name for the helper function.⁵ The special form is called `do`. A simplified version of the syntax for a `do` expression is⁶

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

There are two arguments: a list of triples followed by a list with the termination test expression and a result expression. Each of the triples consist of a name for a state variable, an initialization expression, and an update expression. `do` starts by evaluating the initial expressions, then binds the state variables to these values. Next, it evaluates the termination expression using the iteration variables. If the termination expression evaluates to true, the result expression is evaluated and the result is returned. Otherwise, the update expressions are evaluated, the variables are bound to their new values, the termination test is computed, and so on.

Using `do` we can write `my-sqrt` as

```
(defun my-sqrt (x)
  (do ((guess 1 (improve guess x)))
      ((good-enough-p guess x) guess)))
```

The factorial function becomes

```
(defun factorial (n)
  (do ((counter 1 (+ 1 counter))
        (product 1 (* counter product)))
      ((> counter n) product)))
```

Comparing these definitions to the original definitions in terms of `sqrt-iter` and `fact-iter` shows that each piece of the original definitions fits into a slot in the `do` construct. The argument expressions in the initial call to `fact-iter` become the initialization expressions. The argument expressions in the recursive call to `fact-iter` become the update expressions. The test in the `if` expression of `fact-iter` is the test in the `do` expression, and the consequent expression in the `if` becomes the result for the `do`.

The `do` construct is very powerful, but it can be somewhat intimidating. It is probably easiest to think of it as a simple replacement for helper functions with the structure of `sqrt-iter` or `fact-iter`. An important point is that the initialization expressions and update expressions are evaluated in parallel,

⁵Using these constructs may also be more efficient, in particular in systems that are not tail recursive.

⁶`do` allows some additional variations that are described in Section 4.3.2.

not sequentially. As a result, the order in which the triples are specified does not matter. This is analogous to the way arguments to a helper function are computed before the function is called.

Exercise 3.3

The `good-enough-p` test used here will not work well for very small or very large numbers (why?). Write a modified version that will work reasonably in these cases.

Exercise 3.4

Modify `my-sqrt` to keep track of the number of iterations and stop when the iteration count gets too large. For now, just return `nil` if the iteration count is exceeded before the convergence criterion is satisfied.

3.5 Environments

The variables and functions defined by `def` and `defun` are *global* variables and functions. Once they are defined, they remain in existence until they are redefined or undefined, or you end the session with the interpreter. We have also seen one other type of variable: the parameters of a function are used as *local* variables within the body of the function. Lisp provides several other methods for constructing local variables, and also allows you to construct local functions. To be able to use these methods effectively, we need to introduce a few new ideas and terms.

3.5.1 Some Terminology

Let's start with a very simple example. Suppose we have a global variable `x` defined using the expression

```
(def x 1)
```

and then define a function `f` as

```
(defun f (x) (+ x 1))
```

When we apply `f` to an argument, say, 2, the variable `x` in the body of `f` will refer to the argument with which `f` is called:

```
> (f 2)  
3
```

The pairing of a variable with a value is called a *binding*. In this example, the variable `x` has a global binding to the value 1. When the function is called with the argument 2, a local binding of `x` is created, and the body of

the function is evaluated using that local binding. The local binding is said to *shadow* the global binding. A collection of variable bindings in effect at a particular time is called an *environment*. The set of expressions to which a particular binding applies is called the *scope* of the binding. In this example, the body of the function **f** is evaluated in the environment consisting of the local binding of **x** to the parameter value, and the scope of this local binding is the body of **f**.

Now let's change this example a bit. Define another global variable **a** as

```
(def a 10)
```

and redefine **f** as

```
(defun f (x) (+ x a))
```

The body of **f** now refers to two variables, **x** and **a**. The variable **x** is bound to the function's parameter, and is therefore called a *bound variable*. The variable **a** is not bound to any parameter, and is therefore called a *free variable*. It is not too hard to predict what will happen if the function **f** is applied to the argument 2:

```
> (f 2)
12
```

The value for the free variable **a** is taken from the *global* or *null* environment.

But suppose **f** is called from within a function, for example, the function **g** defined as

```
(defun g (a) (f 2))
```

What is the value of the expression **(g 20)**? To answer this question, we need to know the rules Lisp uses for determining the values of free variables. There are several possibilities. One approach is to look up the values of free variables in the environment in which the function is called. This rule is called *dynamic scoping*. In our example, when **f** is called, the variable **a** has a local binding with value 20, which would give a result of 22 as the value of the expression **(g 20)**. Another approach is to look up the values of free variables in the environment in which the function was originally defined. This is called *static scoping* or *lexical scoping*. Since the function **f** was defined in the global environment, this approach would look up the value of **a** in the global environment, where it is bound to 10, resulting in the value 12 for the expression **(g 20)**.

Common Lisp uses the static scoping rule,⁷ so

⁷Actually, Common Lisp allows you to specify that certain variables are to be treated as dynamically scoped; these variables are called *special variables*. Many older Lisp dialects use dynamic scoping.

```
> (g 20)
12
```

When a Common Lisp function is applied to a set of arguments, a new environment is set up consisting of the environment in which the function was defined together with the parameters of the function bound to the values of the arguments. The expressions in the body of the function are then evaluated in this environment.

Exercise 3.5

Suppose we define the global variables *x* and *y* as

```
(def x 3)
```

and

```
(def y 7)
```

and the functions *f*, *g*, and *h* as

```
(defun f (x) (+ x y))
```

```
(defun g (y) (* x y))
```

and

```
(defun h (x y) (+ (f y) (g x)))
```

What are the values returned by the following expressions?

- a) (f 10)
- b) (g 5)
- c) (h 4 13)
- d) (h (f 2) (g 6))

3.5.2 Local Variables

Defining a function is one way to set up a local environment for evaluating a set of expressions, the body of the function. Another way is to use the special form *let* introduced briefly above in Section 2.7. The general syntax of a *let* expression is

```
(let ((<var 1> <e 1>)
      ...
      (<var n> <e n>))
  <body>)
```

When a `let` expression is evaluated, the value expressions $\langle e_1 \rangle, \dots, \langle e_n \rangle$ are evaluated first. Then an environment consisting of the environment surrounding the `let` expression together with the variables $\langle var_1 \rangle, \dots, \langle var_n \rangle$ bound to the results of the expressions $\langle e_1 \rangle, \dots, \langle e_n \rangle$ is set up. Finally, the expressions in $\langle body \rangle$ are evaluated in sequence in this new environment, and the old environment is restored. The result returned by the `let` expression is the result of evaluating the last expression in $\langle body \rangle$.

The special form `let` is particularly useful for setting up local variables to help simplify an expression. As a simple example, suppose we would like to define a function that takes two lists of numbers, representing real vectors, and returns the projection of one onto the other. If $\langle x, y \rangle$ represents the inner product of x and y , then the projection of y onto x is given by

$$\frac{\langle x, y \rangle}{\langle x, x \rangle} x$$

To compute this projection, we can define the function `project` by

```
(defun project (y x)
  (* (/ (sum (* x y)) (sum (* x x))) x))
```

We can now find the projection of the vector represented by (1 3) onto the vector represented by (1 1):

```
> (project '(1 3) '(1 1))
(2 2)
```

The body of `project` is not too complicated as it is, but we can simplify it further by representing the two inner products as local variables:

```
(defun project (y x)
  (let ((ip-xy (sum (* x y)))
        (ip-xx (sum (* x x))))
    (* (/ ip-xy ip-xx) x)))
```

The local variables `ip-xy` and `ip-xx` are set up to represent the inner products $\langle x, y \rangle$ and $\langle x, x \rangle$, and are used in the body of the `let` expression to compute the projection. The expression is closer to the mathematical definition given above and is thus easier to check.

An important point about `let` is that the bindings it constructs are set up in parallel. The expressions for the values of the new local variables are evaluated in the surrounding environment, and then the bindings are set up. To illustrate this point, let's look at a simple, artificial example,

```
(defun f (x)
  (let ((x (/ x 2))
        (y (+ x 1)))
    (* x y)))
```

and the result of calling this function with the argument 4:

```
> (f 4)
10
```

At first this result might seem a bit surprising; you might have expected the result to be 6. But it makes sense if you work through the evaluation rules given here. The variable *x* in *both* value expressions $(/ \mathbf{x} 2)$ and $(+ \mathbf{x} 1)$ refers to the variable *x* in the surrounding environment, the variable corresponding to the parameter of the function *f*. When the expression $(f 4)$ is evaluated, the value of this variable is 4. Once these two expressions have been evaluated, a new environment is set up with *x* bound to 2 and *y* bound to 5. This new binding for *x*, which remains in effect throughout the body of the *let* expression, shadows the binding from the surrounding environment. The value of the *let* expression is the value of $(* \mathbf{x} \mathbf{y})$ with these bindings for *x* and *y*; thus the value is 10.

It is sometimes useful to set up local variables sequentially, defining first one variable, and then a second one in terms of the first. In the definition of *project*, for example, we might want to define a variable to represent the coefficient of *x* in terms of the inner product variables. Because of the parallel assignment of bindings by *let*, this cannot be done with a single *let* expression, but it could be done using two nested ones:

```
(defun project (y x)
  (let ((ip-xy (sum (* x y)))
        (ip-xx (sum (* x x))))
    (let ((coef (/ ip-xy ip-xx)))
      (* coef x))))
```

This is a common problem, and Lisp provides a simpler approach. The special form *let** works like *let*, except that it adds bindings to the environment one at a time, and evaluates each value expression in an environment that includes all bindings constructed so far. Using *let**, we can write *project* as

```
(defun project (y x)
  (let* ((ip-xy (sum (* x y)))
        (ip-xx (sum (* x x)))
        (coef (/ ip-xy ip-xx)))
    (* coef x)))
```

If we replace *let* by *let** in the little artificial example used above,

```
(defun f (x)
  (let* ((x (/ x 2))
        (y (+ x 1)))
    (* x y)))
```

then the result would indeed be 6:

```
> (f 4)
6
```

3.5.3 Local Functions

In addition to variable bindings, environments also contain function bindings. Until now we have always defined global function bindings by using `defun`. The `flet` special form can be used to set up local function definitions that are only visible within the body of the `flet` expression. This allows you to define a function in terms of simple helper functions without having to worry about name conflicts with other globally defined functions.

The general form of an `flet` expression is

```
(flet ((name 1) (parameters 1) (body 1))
      ...
      ((name n) (parameters n) (body n)))
  (body))
```

The symbols $\langle \text{name } 1 \rangle, \dots, \langle \text{name } n \rangle$ are the names for the local functions, the lists $\langle \text{parameters } 1 \rangle, \dots, \langle \text{parameters } n \rangle$ are the parameter lists, and $\langle \text{body } 1 \rangle, \dots, \langle \text{body } n \rangle$ are the expressions or sequences of expressions making up the bodies of the functions.

As an example, we can look at another way of writing our `project` function. Instead of using local variables to represent the inner products, we can use a local inner product function called `ip`:

```
(defun project (y x)
  (flet ((ip (x y) (sum (* x y))))
    (* (/ (ip x y) (ip x x)) x)))
```

Like `let`, `flet` constructs its bindings in parallel in the surrounding environment. This implies that none of the functions defined in a particular `flet` can refer to each other, or to themselves. To define a second local function, `coef`, in terms of `ip` we would have to use a second `flet`, as in

```
(defun project (y x)
  (flet ((ip (x y) (sum (* x y))))
    (flet ((coef (x y) (/ (ip x y) (ip x x))))
      (* (coef x y) x))))
```

Again, Lisp provides a simpler alternative, the special form `labels`. Like `let*`, `labels` constructs its bindings sequentially allowing each function to refer to previously defined functions, or to itself. Thus `labels` allows you to define recursive local functions. For our `project` function we can use `labels` as

```
(defun project (y x)
  (labels ((ip (x y) (sum (* x y)))
           (coef (x y) (/ (ip x y) (ip x x))))
    (* (coef x y) x)))
```

3.6 Functions and Expressions as Data

One of the great strengths of Lisp is its ability to use functions as data and to construct new customized functions. This section introduces several functions and techniques that take advantage of this ability.

3.6.1 Anonymous Functions

In Section 2.7 we drew a plot of the function $f(x) = 2x + x^2$ over the range $[-2, 3]$ by first defining a function `f` as

```
(defun f (x) (+ (* 2 x) (^ x 2)))
```

and then evaluating the expression⁸

```
(plot-function #'f -2 3)
```

Once we have our plot, we have no further use for the function `f`. It would be nice if we could avoid having to define it formally and think up a name for it. The same problem exists in mathematics: in order to describe the function I wanted to plot, I referred to it as the “the function $f(x) = 2x + x^2$.” To deal with this problem, logicians have developed the *lambda calculus* [18], which allows you to use an expression like

$$\lambda(x)(2x + x^2)$$

to refer to “the function that returns $2x + x^2$ for the argument x .” Lisp has adopted this approach as well, allowing a function to be described by a *lambda expression*, a list consisting of the symbol `lambda`, a list of parameters, and one or more expressions making up the body of the function. A lambda expression for our function would look like

```
(lambda (x) (+ (* 2 x) (^ x 2)))
```

Since functions described by lambda expressions have no names, they are sometimes called *anonymous functions*.

A lambda expression can be used in place of a symbol as the first element of an expression given to the interpreter:

⁸Recall that the expression `#'f`, short for `(function f)`, is needed to extract the function definition of the symbol `f`.

```
> ((lambda (x) (+ (* 2 x) (- x 2))) 1)
3
```

A lambda expression can also be passed as an argument to a function like `plot-function`. This is usually done by first combining it with the current environment, to be used to determine the values of free variables. Such a combination is called a *function closure*, or simply a *closure*. A closure is constructed using the `function` special form, or its #' abbreviation. Thus to plot our function, we would use the expression⁹

```
(plot-function #'(lambda (x) (+ (* 2 x) (- x 2))) -2 3)
```

The ability to connect a function definition with an environment in a function closure is an extremely powerful programming tool. To be able to take full advantage of this idea, however, we need to examine how to define functions that use functions passed to them as arguments.

3.6.2 Using Function Arguments

Suppose we want to approximate an integral as¹⁰

$$\int_a^b f(x)dx \approx \left[f(a + \frac{h}{2}) + f(a + h + \frac{h}{2}) + f(a + 2h + \frac{h}{2}) + \dots \right] h$$

One approach is to define an integration function using the `do` construct introduced in Section 3.4 as

```
(defun integral (a b h)
  (do ((integral 0 (+ integral (* h (f x)))))
       ((x (+ a (/ h 2)) (+ x h)))
    ((> x b) integral)))
```

This definition assumes that there is a globally defined function `f` that evaluates the function to be integrated. For example, to integrate x^2 over the interval $[0, 1]$, we can define `f` as

```
(defun f (x) (^ x 2))
```

and then use `integral` to compute the integral:

⁹In some Lisp systems you may be able to use the expression

```
(plot-function (lambda (x) (+ (* 2 x) (^ x 2))) -2 3)
```

In these systems `lambda` is a macro that creates function closures. In the definition of Common Lisp, however, lambda expressions cannot be evaluated. This approach will therefore not work in Lisp systems that follow the Common Lisp standard. The `function` special form or #' should thus always be used to create closures.

¹⁰Adapted from Abelson and Sussman [1, Section 1.3.1].

```
> (integral 0 1 .01)
0.333325
```

It would be better to design `integral` to take the function to be integrated as an argument. To be able to do this, we need to know how to make use of such a function argument. At first it might seem that we could define `integral` as

```
(defun integral (f a b h)
  (do ((integral 0 (+ integral (* h (f x)))))
       (x (+ a (/ h 2)) (+ x h)))
      ((> x b) integral)))
```

and integrate our function using the expression

```
(integral #'(lambda (x) (^ x 2)) 0 1 .01)
```

Unfortunately, this will not work. (Try it and see what happens.) The reason is that the local variable `f` has our function as its *value*, not as its *function definition*.

Instead, we can use the function `funcall`. This function takes a function argument, and as many arguments as the function argument requires, and applies the function to these arguments. Here are a few examples:

```
> (funcall #'(lambda (x) (^ x 2)) 2)
4
> (funcall #'^+ 1 2)
3
```

Using `funcall`, we can define `integral` as

```
(defun integral (f a b h)
  (do ((integral 0 (+ integral (* h (funcall f x)))))
       (x (+ a (/ h 2)) (+ x h)))
      ((> x b) integral)))
```

This definition will work the way we want it to:

```
> (integral #'(lambda (x) (^ x 2)) 0 1 .01)
0.333325
```

The function `funcall` is useful when you know in advance how many arguments the function argument will take. If you do not know, or if the function can take a variable number of arguments, then you can use the function `apply`. This function takes a function argument and a list of arguments for the function, applies the function, and returns the result. Here are a few examples:

```
> (apply #'+ (list 1 2))  
3  
> (apply #'+ (list 1 2 3))  
6
```

It is possible to insert a number of additional arguments for the function argument between the function argument and the list. These arguments will be passed to the function in the order in which they are given, ahead of the arguments in the list. For example,

```
> (apply #'+ 1 2 (list 3 4 5))  
15
```

This technique will prove to be useful in later chapters.

There are some limitations to `funcall` and `apply`. They can only be used with functions, not special forms or macros. Furthermore, most Lisp systems place an upper limit on the number of arguments that can be given to a function. Although this limit may be large in some systems, the Common Lisp specification only requires that it be at least 50. This means that it is not a good idea, for example, to define a function for summing the elements in a list as

```
(defun my-sum (x) (apply #'+ x))
```

This definition will work for small lists but not for large ones. `apply` and `funcall` can also be given symbols as their function arguments. The function definitions for these symbols are determined in the global environment.

Exercise 3.6

Write a function to approximate an integral by Simpson's rule. Use a few examples to compare your function to the function defined here.

3.6.3 Returning Functions as Results

Now that we have seen how to use function arguments, we can return to the use of function closures. In Section 3.5 we defined a function `project` to compute the projection of y on x . Another way to think about the mathematical projection problem is to think in terms of the *projection operator* that takes any vector y into its projection on x . We can denote this operator by P_x , or simply by P , and write the projection of y onto x as Py . The operator is viewed as a function of a single argument. The space onto which vectors are projected, the space spanned by x , is somehow "built into" P .

We can model this projection operator in Lisp using a function closure. First, we construct a function `make-projection`, which takes a list x representing a mathematical vector and returns a function closure that computes the projection of its argument onto x :

```
(defun make-projection (x)
  (flet ((ip (x y) (sum (* x y))))
    #'(lambda (y) (* x (/ (ip x y) (ip x x))))))
```

The lambda expression used to construct the result contains a free variable `x`. In the environment in which the lambda expression is converted to a closure, `x` refers to the argument used in the call to `make-projection`. Thus the function closure returned by `make-projection` “remembers” the vector onto which it is to project. As an example, we can define the projection operator onto the constant vector in four dimensions as

```
(def p (make-projection '(1 1 1 1)))
```

The projection operator is now the *value* of the symbol `p`, so we have to apply it to an argument using `funcall` or `apply`:

```
> (funcall p '(1 2 3 4))
(2.5 2.5 2.5 2.5)
> (funcall p '(-1 1 1 -1))
(0 0 0 0)
```

Instead of using a lambda expression to construct our result, we could also have used a local function. To allow this local function to use the inner product function `ip`, we have to use either a nested pair of `flet` expressions or a `labels` expression, as in

```
(defun make-projection (x)
  (labels ((ip (x y) (sum (* x y)))
           (proj (y) (* x (/ (ip x y) (ip x x)))))
    #'proj))
```

Exercise 3.7

Write a function that takes a list of data, assumed to be a random sample from a $N(\mu, \sigma^2)$ population, and returns the log likelihood function for the data.

Exercise 3.8

Write a function that takes two functions of one argument and returns their composition. The composition $f \circ g$ of f and g is defined as

$$(f \circ g)(x) = f(g(x)).$$

3.6.4 Expressions as Data

Instead of using function arguments, you may at times find it more convenient to use expressions. A compound Lisp expression is simply a Lisp list. Its elements can be extracted by using either the `select` function or the functions `first`, `second`, ..., `tenth`:

```
> (def expr '(+ 2 3))
EXPR
> (first expr)
+
> (second expr)
2
```

Another useful function for examining expressions is `rest`. This function takes a list and returns a list of all but the first element:

```
> (rest expr)
(2 3)
```

The function `eval` can be used to evaluate an expression:

```
> (eval expr)
5
```

The function `eval` does its evaluation in the global environment. Thus if your expression contains any variables, the global values of these variables will be used:

```
> (def x 3)
3
> (let ((x 5)) (eval 'x))
3
```

If you want to replace a variable in an expression with a particular value before evaluating the expression, you can use the function `subst`:

```
> (def expr2 '(+ x 3))
EXPR2
> (eval (subst 2 'x expr2))
5
```

Actually, `subst` replaces all occurrences of its second argument with its first argument, without regard for syntax. This can result in meaningless expressions:

```
> (subst 2 'x '(let ((x 3)) x))
(LET ((2 3)) 2)
```

An alternative is to construct a `let` expression around your expression and pass it to `eval`:

```
> (list 'let '((x 2)) expr2)
(LET ((X 2)) (+ X 3))
> (eval (list 'let '((x 2)) expr2))
5
```

This process of having to fill in a template by creating a list in which all but a few expressions are quoted is fairly common. Once again Lisp provides a shorthand notation. A backquote ` in front of a list causes all elements of the list to be quoted, except those preceded by a comma:

```
> `'(let ((x 2)) ,expr2)
(LET ((X 2)) (+ X 3))
> (eval `'(let ((x 2)) ,expr2))
5
```

The commas need not appear at the top level of the backquoted list but can be contained in sublists:

```
> `'(let ((x ,(- 3 1))) ,expr2)
(LET ((X 2)) (+ X 3))
```

As an example, we can construct a simple function for plotting the result of a vectorized expression against the values of a variable. If we call the function `plot-expr`, then

```
(plot-expr '(+ (* 2 x) (^ x 2)) 'x -2 3)
```

should produce a plot of $2x+x^2$ for x in the range $[-2, 3]$. Using the backquote mechanism, we can define this function as

```
(defun plot-expr (expr var low high)
  (flet ((f (x) (eval `'(let ((,var ,x)) ,expr))))
    (plot-function #'f low high)))
```

One advantage of using an expression argument instead of a function is that `plot-expr` now has access to the expression and the variable name. They could be used to construct meaningful axis labels for the plot.

3.7 Mapping

Often it is useful to be able to apply a function elementwise to a list. This process is called *mapping*. The `mapcar` function takes a function and a list of arguments, and returns the list of results of applying the function to each element:

```
> (def x (mapcar #'normal-rand '(2 3 2)))
X
> x
((0.2739724 3.535817) (-0.1106453 1.217806 1.05023)
 (0.7826753 0.9595545))
> (mapcar #'mean x)
(1.904895 0.71913 0.8711149)
```

`mapcar` can take several lists as arguments. The function to be mapped must take as many arguments as there are lists. The function is called with the first elements, then the second elements, etc.:

```
> (mapcar #'+ '(1 2 3) '(4 5 6))
(5 7 9)
```

If the list arguments are not all the same length, then the evaluation stops once the shortest list has been used up.

Another mapping function available in Lisp-Stat is `map-elements`. This function uses the Lisp-Stat distinction between *compound data* and *simple data*. Compound data are lists, vectors, arrays, and compound data objects (to be introduced later). A data item that is not compound is considered to be simple. The predicate `compound-data-p` can be used to test whether an item is compound.

The function `map-elements` allows you to map a function on arguments that can be a combination of simple and compound items. If any of the arguments are compound, any simple items are treated as constants of the appropriate size. For example,

```
> (map-elements #'+ 1 '(1 2 3) '(4 5 6))
(6 8 10)
```

All vectorized arithmetic functions in Lisp-Stat implicitly use a recursive call to `map-elements`. The definition of the `+` function is similar to

```
(defun vec+ (x y)
  (if (or (compound-data-p x) (compound-data-p y))
      (map-elements #'vec+ x y)
      (+ x y)))
```

Some examples are

```
> (vec+ 1 2)
3
> (vec+ 1 '(2 3))
(3 4)
> (vec+ '(1 2) '(3 4))
(4 6)
> (vec+ '(1 2) '(3 (4 5)))
(4 (6 7))
```

Unlike `mapcar`, the function `map-elements` expects all its compound arguments to have the same number of elements:

```
> (vec+ '(1 2) '(3 4 5))
error: arguments not all the same length
```

In fact, compound arguments should all be the same shape; thus arrays should have the same dimensions.

You usually do not need to use `map-elements` to define your own vectorized functions unless you require conditional computation. For example, suppose you want to define a vectorized version of

$$f(x) = \begin{cases} \log(x) & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Lisp-Stat provides a function `if-else` that is called as

```
(if-else <x> <y> <z>)
```

If `<x>`, `<y>`, and `<z>` are all lists of the same length, `if-else` returns a list of that length. The i -th element of the result is either the i -th element of `<y>` or of `<z>`, depending on whether the i -th element of `<x>` is non-nil or nil. It might appear that a simple way to write our function would be to use `if-else`:

```
(defun f (x) (if-else (> x 0) (log x) 0))
```

Unfortunately, this will not work. Since `if-else` is a function, all its arguments are evaluated before the function is called. Therefore the attempt to call `log` with a zero argument will generate an error before `if-else` has a chance to examine the result of the comparison. We must use a more complex definition, such as

```
(defun f (x)
  (if (compound-data-p x)
    (map-elements #'f x)
    (if (> x 0) (log x) 0)))
```

Exercise 3.9

Write a function that takes a list of lists of numbers, viewed as a list of samples, and returns a list of the within groups and between groups sums of squares.

3.8 Assignment and Destructive Modification

In procedural programming languages like FORTRAN or C, the assignment statement is the most basic element of programming. For example, in the following C definition of the factorial function, the assignment used to update the local variable `prod` is the key step in accumulating the result:

```
int factorial(n)
    int n;
{
    int count;
    int prod = 1;

    for (count = 0; count < n; count = count + 1)
        prod = prod * (count + 1);
    return(prod);
}
```

At this point we cannot write a Lisp translation of this program. We can set up local variables like `prod`, but we have no way to change their values, to assign new values to them. Perhaps more important, we have had no need for an assignment mechanism.

Everything we have done so far in this chapter has been accomplished using a *functional* or *applicative* programming style. More complex functions have been constructed as compositions of simpler functions. Local variables have only been used as definitions to simplify expressions. The meanings of our functions would not be changed if we replaced local variables created in a `let` expression, for example, by their definitions, though the function definitions would become less clear.¹¹ In contrast, the C program for the factorial uses the local variable `prod` as a storage location. Replacing `prod` everywhere by its initial value, 1, would produce nonsense.

A programming style that uses local variables only for definitions and does not change their values by assignment is called *referentially transparent*. There are in fact good reasons to use such a style whenever possible. In a nutshell, it is much harder to construct a mathematical proof that a program is correct if the program uses assignment than if it does not. Abelson and Sussman [1] discuss this issue at length.

Assignment does, however, have some important applications. For example, we might want to construct a software representation of a window on a computer screen. The window has various characteristics, like its size and location, that can be recorded as local state *variables*. As a user interacts with the window, it may be moved or resized. For our representation to remain current, we need to be able to change these state variables, to assign new values to them.

The basic tool for assignment in Lisp is the special form `setf`. `setf` can be used to change the value of a global variable or a local variable. For example,

¹¹In cases where the value of a local variable is used several times, replacing it by the expression used to compute its value will cause that expression to be evaluated several times. This will slow down the computation. But as long as the expressions do not produce any side effects, the final result of the `let` expression will not be affected.

```
> (setf x 3)
3
> x
3
```

and

```
> (let ((x 1))
  (setf x 2)
  x)
2
```

The first argument to `setf` is not evaluated, so the symbol `x` does not need to be quoted. Unlike `setf`, `def` only affects global bindings.

An example of a problem in which we need to be able to modify a state variable is a random number generator. A linear congruential generator is specified by a seed X_0 , a multiplier A , and a modulus M , all positive integers. It computes a sequence X_1, X_2, \dots , of pseudorandom integers by the rule

$$X_{i+1} = A * X_i \bmod M$$

An approximately uniform pseudorandom sequence U_1, U_2, \dots , is then produced as

$$U_i = X_i / M$$

We can implement such a generator using a function closure that contains the multiplier, the modulus, and the current value of X . The current value of X is the state variable that is updated each time a new number is obtained. The function `make-generator` constructs such a closure:

```
(defun make-generator (a m x0)
  (let ((x x0))
    #'(lambda ()
      (setf x (rem (* a x) m))
      (/ x m))))
```

The function `rem` calculates the remainder of its first argument upon division by its second argument. The body of the lambda expression in this definition contains two expressions. The first is an assignment that is used to produce the side effect of changing the value of the local variable `x`. The second expression produces the result that is returned when the generator is used. The lambda expression takes no arguments.

We can construct and try out a particular generator using a multiplier of $A = 7^5$, a modulus of $M = 2^{31} - 1$, and seed of 12345:¹²

¹²I have used floating point numbers for the parameters since this is more efficient in many Lisp systems and avoids overflow in XLISP-STAT.

```
> (def g (make-generator (^ 7.0 5) (- (^ 2.0 31) 1) 12345))
G
> (funcall g)
0.09661653
> (funcall g)
0.8339946
> (funcall g)
0.9477025
```

Since the function returned by `make-generator` does not require any arguments, `funcall` needs no arguments other than the generator `g`. The local state variable `x` in the environment in which the function closure `g` was created is updated each time `g` is called. As a result, each call to `g` returns a different value.

Assignment can be used to write programs in a procedural style. For example, we can now translate the C version of the factorial function given at the beginning of this section into Lisp:

```
(defun factorial (n)
  (let ((prod 1))
    (dotimes (count n)
      (setf prod (* prod (+ count 1))))
    prod))
```

The `dotimes` construct was introduced briefly in Section 2.5.6. It repeats its body `n` times with `count` bound to $0, \dots, n - 1$.

As we saw in Section 2.4.5, `setf` can also be used to *destructively modify* a list by changing some of its elements. For example, if `x` is constructed as

```
(setf x (list 1 2 3))
```

we can change its second element, the element with index 1, using the expression

```
(setf (select x 1) 'a)
```

The expression `(select x 1)` in the `setf` expression is called a *place form*, or a *generalized variable*. A number of other place forms are available, and it is also possible to define new place forms, or *setf methods*. This will be discussed further in the next chapter.

It is worth emphasizing once again that Lisp variables are merely names for items and that destructive modification can have unexpected side effects. In particular, if `x` is constructed by

```
(setf x '(1 2 3))
```

and `y` is defined using

```
(setf y x)
```

then modifying the value of `x` will modify the value of `y` since these symbols are merely two different names used to refer to the same Lisp item:

```
> (setf (select x 1) 'a)
A
> x
(1 A 3)
> y
(1 A 3)
```

You can use the `copy-list` function to make a copy of `x` before modifying it.

3.9 Equality

Many functions need to determine when two Lisp items are to be considered the same. This turns out to be a surprisingly subtle issue, in particular after the introduction of assignment. Whether two things are to be considered equal depends on what is to be done with them. The strings “fred” and “Fred” may be considered different since they contain different characters. They may also be viewed as the same since they both spell out the name *Fred*. When you type the name of a symbol into the Lisp interpreter, the case of the characters is ignored. For this purpose the two strings are the same. If you compute a code for the strings using the ASCII codes of their characters, you get different results for the two strings. For this purpose the strings are different.

As another example, suppose you have two lists that both print as

`(A B)`

Are they equal? If you are only going to extract their elements, then both will produce the same results, and they may therefore be considered the same. On the other hand, with the introduction of `setf` we are able to physically modify the contents of a list. Suppose we do this to one of our lists. Will this affect the other one as well? This depends on whether they have the same location in computer memory.

To deal with this situation, Lisp provides four different equality predicates of varying levels of stringency. The most stringent is `eq`. Two items are `eq` if and only if they occupy the same location in memory. This is the test we would want to use to make sure our two lists are really different before we modify one of them.

When the interpreter translates a string of characters into a symbol it ensures that two symbols with the same name are `eq`.

A closely related predicate is `eql`. The only difference between `eq` and `eql` is that `eql` considers numbers of the same type and value, characters¹³ with

¹³The Lisp character data type will be introduced in Section 4.4.

the same value and case, and strings with the same characters in identical cases to be `eql`. Depending on the Lisp implementation, they may or may not be `eq`. Thus

```
(eq 1 1)
```

may return `t` for some implementations and `nil` for others, but

```
(eql 1 1)
```

always returns `t`. On the other hand,

```
> (eq (list 'a 'b) (list 'a 'b))
NIL
> (eql (list 'a 'b) (list 'a 'b))
NIL
> (eq 1 1.0)
NIL
> (eql 1 1.0)
NIL
```

The two calls to `list` return different lists, and the integer 1 and the floating point number 1.0 are of different types.

The predicates `equal` and `equalp` are used to determine if two items *look* the same. The expression `(equal x y)` returns `t` if `(eql x y)` returns `t` or if `x` and `y` are lists of the same length with all elements `equal`. The `equalp` predicate is slightly weaker still. It considers two numbers to be `equalp` if they are numerically equal, regardless of type, and two strings are `equalp` if they have the same characters, regardless of case. Thus

```
> (equal (list 'a 'b) (list 'a 'b))
T
> (equalp (list 'a 'b) (list 'a 'b))
T
> (equal 1 1.0)
NIL
> (equalp 1 1.0)
T
> (equal "fred" "Fred")
NIL
> (equalp "fred" "Fred")
T
```

Several functions, such as the function `subst` introduced in Section 3.6.4, need to test for equality among elements of lists. By default these functions use the `eql` test. A keyword argument can be used to override this default (see Section 4.4).

3.10 Some Examples

This chapter has introduced a large number of new ideas and techniques. Before proceeding further, it may be helpful to look at a few more extensive examples that draw on several of the techniques presented here. The first example uses Newton's method to find the root of a function of one variable. The second shows one approach to constructing a symbolic differentiator.¹⁴

3.10.1 Newton's Method for Finding Roots

Newton's method for finding a root of a differentiable function f takes a guess y and computes a (hopefully) improved guess as

$$y - \frac{f(y)}{Df(y)}$$

where Df denotes the derivative of f . The basic iteration can be developed along the same lines as the square root iteration in Section 3.4 by using a recursive definition of the form

```
(defun newton-search (f df guess)
  (if (good-enough-p guess f df)
      guess
      (newton-search f df (improve guess f df))))
```

or by using do:

```
(defun newton-search (f df guess)
  (do ((guess guess (improve guess f df)))
      ((good-enough-p guess f df) guess)))
```

The functions `improve` and `good-enough-p` could be defined as

```
(defun improve (guess f df)
  (- guess (/ (funcall f guess) (funcall df guess))))
```

and

```
(defun good-enough-p (guess f df)
  (< (abs (funcall f guess)) .001))
```

As a check, we can use `newton-search` to find π as the root of $\sin(x)$ near 3:

```
> (newton-search #'sin #'cos 3)
3.14255
```

¹⁴Both examples have been adapted from examples in Abelson and Sussman [1].

This definition of `newton-search` has some drawbacks. In particular, the definitions of `improve` and `good-enough-p` may interfere with other definitions, such as the ones which we set up for the square root problem. To avoid this difficulty, we can use `flet` to set up a *block structure*:

```
(defun newton-search (f df guess)
  (flet ((improve (guess f df)
              (- guess
                 (/ (funcall f guess)
                     (funcall df guess))))
         (good-enough-p (guess f df)
                      (< (abs (funcall f guess)) .001)))
         (do ((guess guess (improve guess f df))
              ((good-enough-p guess f df) guess))))
    ...))
```

The functions `improve` and `good-enough-p` are now only visible within the body of `newton-search`.

Having moved `improve` and `good-enough-p` into `newton-search`, we can simplify them a little. Since `f` and `df` are available in the environment in which `improve` and `good-enough-p` are defined, we do not need so pass them as arguments:

```
(defun newton-search (f df guess)
  (flet ((improve (guess)
              (- guess
                 (/ (funcall f guess)
                     (funcall df guess))))
         (good-enough-p (guess)
                      (< (abs (funcall f guess)) .001)))
         (do ((guess guess (improve guess)))
              ((good-enough-p guess) guess))))
    ...))
```

Finding derivatives (correctly) is often a problem. Since numerical derivatives are often accurate enough for Newton's method, we could rewrite the function `newton-search` to use numerical derivatives. But this would mean that we could not take advantage of exact derivatives when we do have them available. An alternative is to construct a function that produces a numerical derivative function:

```
(defun make-derivative (f h)
  #'(lambda (x)
      (let ((fx+ (funcall f (+ x h)))
            (fx- (funcall f (- x h)))
            (2h (* 2 h)))
        (/ (- fx+ fx-) 2h))))
```

The result returned by `make-derivative` is a function closure that remembers both the function `f` and the step size `h` to use in computing the numerical

derivative. When this function is called with an argument x , it approximates the derivative of f at x using a symmetric difference quotient. We can now use `make-derivative` to supply the derivative argument to `newton-search`:

```
> (newton-search #'sin (make-derivative #'sin .001) 3)
3.14255
```

We can also use `newton-search` and `make-derivative` to locate the maximum of a function by finding the root of its derivative. As an example, suppose we want to find the maximum likelihood estimator of the exponent in a gamma distribution with its scale parameter known to be 1. We can generate a sample from this distribution with exponent $\alpha = 4.5$, say, and assign it to a variable x using

```
(def x (gamma-rand 30 4.5))
```

The log likelihood can be written as

$$(\alpha - 1)s - n \log \Gamma(\alpha)$$

with $s = \sum \log X_i$, the sufficient statistic, and n the sample size.

A function for evaluating this log likelihood can be constructed as

```
(def f (let ((s (sum (log x)))
            (n (length x)))
      #'(lambda (a)
          (- (* (- a 1) s) (* n (log-gamma a))))))
```

The function is now the value of the variable f . The `let` statement surrounding the lambda expression is used to create an environment in which the variables s and n , representing the sufficient statistic and the sample size, are bound to the values appropriate for our sample. The process of generating this function closure is analogous to the mathematical process of suppressing the dependence of the log likelihood on the data once the sample has been obtained.

To use `newton-search` to find the maximum likelihood estimate of α , we need the first and second derivatives of the log likelihood function. These can be obtained as

```
(def df (make-derivative f .001))
```

and

```
(def ddf (make-derivative df .001))
```

We also need an initial guess for the maximum likelihood estimator. Since the scale parameter in the gamma distribution is 1, the method of moments estimate of α is the sample mean,

```
> (mean x)  
4.426856
```

Now we can search for the maximum likelihood estimator:

```
> (newton-search df ddf (mean x))  
4.481326
```

Exercise 3.10

Suppose the scale parameter in the gamma distribution needs to be estimated as well. One way to estimate two parameters is to fix the first one and estimate the second, then fix the second and estimate the first, and continue this cycle until the result converges. For a fixed value of α the maximum likelihood estimator of the scale parameter is available in closed form. Use this and the function `newton-search` to develop a function for computing the maximum likelihood estimates of both parameters of a gamma distribution.

3.10.2 Symbolic Differentiation

Symbolic differentiation applies the rules of differential calculus to expressions and produces expressions representing the derivatives. It may seem surprising at first, but it is in fact quite simple to develop a computer program that applies these rules automatically. The hardest part is deciding how to represent the data used by such a program: the expressions to be differentiated and the expressions representing the derivatives.

To keep things simple, let's start the way a calculus class might start, by considering only the rules for differentiating constants, variables, sums, and products.

Abstract Expressions

The data to be used by a differentiation function are expressions. There are many different ways in which these could be represented on a computer, but the details of the representation matter little to the basic task the differentiator has to perform. To reflect this fact, it is useful to think briefly about exactly what features of expression are really needed to develop the differentiation function, and to develop a set of functions that capture these features. This set of functions, called an *abstract data representation*, can then be used to write our differentiation function. Later, we can experiment with different ways of developing this data representation. But, by separating the *use* of our expressions from the internal details of their representation, we will have obtained a system that is easier to understand and to modify than one in which data representation and data use are intertwined. This programming strategy is called *data abstraction*.

What are the basic features of expressions that we need to represent in our functional representation? First, there are four types of expressions:

constants, variables, sums, and products. We need to be able to recognize whether a particular item is of one of these types. Let's assume that we can define four predicates for this purpose,

```
(constantp (e))
(variablep (e))
( sump (e))
(productp (e))
```

We will also need one more predicate to recognize whether two variables are the same or not,

```
(same-variable-p (v1) (v2))
```

Constants and variables are atomic expressions that cannot be decomposed or constructed from other expressions. In contrast, sums and products are compound expressions. We therefore need *accessor functions* to extract their components, and *constructor functions* to construct new sums and products. The components of a sum are the addend and the augend. Suppose we can access them using

```
(addend (e))
(augend (e))
```

and we can construct a new sum with

```
(make-sum (a1) (a2))
```

The components of a product, the multiplicand and multiplier, can be accessed with

```
(multiplicand (e))
(multiplier (e))
```

and a new product is constructed by

```
(make-product (m1) (m2))
```

These functions are all we need to describe the differentiation process:

```
(defun deriv (exp var)
  (cond
    ((constantp exp) 0)
    ((variablep exp) (if (same-variable-p exp var) 1 0))
    ((sump exp)
      (make-sum (deriv (addend exp) var)
                (deriv (augend exp) var)))
    ((productp exp)
      (make-sum (make-product (multiplier exp)
                             (deriv (multiplicand exp) var))
                (make-product (deriv (multiplier exp) var)
                             (multiplicand exp)))))
    (t (error "Can't differentiate this expression"))))
```

The function `error` is used to signal an error if no applicable differentiation rule is found. The string argument is the error message.

The function `deriv` represents a simple Lisp encoding of the first few differentiation rules covered in a calculus class. The rule for sums, for example, is a Lisp encoding of the rule

$$\frac{d(u + v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

The details of how expressions are implemented are not important, as long as they conform to the interface of the functional abstraction used here. Before we can use this function, though, we do need to develop at least one representation for expressions.

A Representation for Expressions

There are many ways to represent expressions, but one of the easiest is to use standard Lisp syntax, with constants represented by numbers and variables by symbols.

Let's start by defining the accessor and constructor functions for sums and products. For sums we have

```
(defun addend (e) (second e))
```

```
(defun augend (e) (third e))
```

and

```
(defun make-sum (a1 a2) (list '+ a1 a2))
```

For products we have

```
(defun multiplier (e) (second e))
```

```
(defun multiplicand (e) (third e))
```

and

```
(defun make-product (m1 m2) (list '* m1 m2))
```

The predicates for testing whether an item is a constant or a variable are quite simple:

```
(defun constantp (e) (numberp e))
```

and

```
(defun variablep (e) (symbolp e))
```

Testing whether two variables are the same is also easy:

```
(defun same-variable-p (v1 v2)
  (and (variablep v1) (variablep v2) (eq v1 v2)))
```

We can use the `eq` predicate since two symbols are the same if and only if they are `eq`.

The predicates for testing for a sum or a product are just a little more complicated. We need to check whether the item is a list of three elements with the leading element equal to `+` or `*`. Thus for sums

```
(defun sump (e)
  (and (listp e) (= (length e) 3) (eq (first e) '+)))
```

and for products

```
(defun productp (e)
  (and (listp e) (= (length e) 3) (eq (first e) '*)))
```

Now we can try our derivative function on some examples:

```
> (deriv '(+ x 3) 'x)
(+ 1 0)
> (deriv '(* x y) 'x)
(+ (* X 0) (* 1 Y))
> (deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* X Y) (+ 1 0))
  (* (+ (* X 0) (* 1 Y))
     (+ X 3)))
```

The results are correct, but they are more complicated than they need to be. Simplifying an expression is unfortunately a much more difficult task than differentiation, primarily because it is not easy to state what it means for one expression to be simpler than another. We can, however, improve our differentiator somewhat by making our functions for constructing sums and products a little more intelligent. In the definition of `make-sum` we can check if both arguments are numbers and, if so, return their sum. If one of the arguments is zero, we can simply return the other argument:

```
(defun make-sum (a1 a2)
  (cond
    ((and (numberp a1) (numberp a2)) (+ a1 a2))
    ((numberp a1) (if (= a1 0) a2 (list '+ a1 a2)))
    ((numberp a2) (if (= a2 0) a1 (list '+ a1 a2)))
    (t (list '+ a1 a2))))
```

Similarly, for `make-product` we can return the product if both are numbers, zero if one is zero, and the other argument if one of the arguments is equal to one:

```
(defun make-product (m1 m2)
  (cond
    ((and (numberp m1) (numberp m2)) (* m1 m2))
    ((numberp m1)
     (cond ((= m1 0) 0)
           ((= m1 1) m2)
           (t (list '* m1 m2))))
    ((numberp m2)
     (cond ((= m2 0) 0)
           ((= m2 1) m1)
           (t (list '* m1 m2))))
    (t (list '* m1 m2))))
```

Now the results are a little more reasonable, though still not perfect:

```
> (deriv '(+ x 3) 'x)
1
> (deriv '(* x y) 'x)
Y
> (deriv '(* (* x y) (+ x 3)) 'x)
(+ (* X Y) (* Y (+ X 3)))
```

Exercise 3.11

The Lisp procedures `+` and `*` take one, two, or more arguments. Modify the data representation to accommodate these cases. (Hint: For `+` you only need to modify the definitions of `sump` and `augend`.)

Exercise 3.12

Add rules for handling differences and quotients.

Exercise 3.13

Add a rule for handling the differentiation rule

$$\frac{d(u^n)}{dx} = n u^{n-1} \left(\frac{du}{dx} \right)$$

Recall that powers can be computed in Lisp-Stat with the function `^`.

Adding Unary Functions

Suppose we want to add the `exp` function. Instead of adding an explicit rule for this function to `deriv`, it is probably better to add a rule representing the chain rule, and to handle the particular function to be differentiated in the data representation. To add the chain rule, we need a predicate for recognizing a unary function call,

```
(unary-p (e))
```

We also need accessor functions for determining the function name and the function argument,

```
(unary-function (e))
(unary-argument (e))
```

and we need to be able to construct an expression for the derivative of the function evaluated at an argument:

```
(make-unary-deriv (f) (x))
```

We can then modify `deriv` as

```
(defun deriv (exp var)
  (cond
    ...
    ((unary-p exp)
     (make-product (make-unary-deriv (unary-function exp)
                                      (unary-argument exp))
                  (deriv (unary-argument exp) var)))
    (t (error "Can't differentiate this expression"))))
```

The function `make-unary-deriv` can also be defined in terms of our abstract representation of expressions. One approach is to use a `case` construct, as in

```
(defun make-unary-deriv (fcn arg)
  (case fcn
    (exp (make-unary 'exp arg))
    (sin (make-unary 'cos arg))
    (cos (make-product -1 (make-unary 'sin arg))))
    (t (error "Can't differentiate this expression"))))
```

`case` takes an expression that evaluates to a symbol, the *case selector*, followed by a series of *case clauses*. Each clause starts with a symbol or a list of symbols, and `case` proceeds down the list of clauses until the selector matches one of the symbols. When it finds a match, the remaining expressions in the matched clause are evaluated and the result of the last expression is returned. If no match is found, `nil` is returned. The symbol `t` is special: it matches any selector.

This definition of `make-unary-deriv` requires one additional constructor,

```
(make-unary (f) (x))
```

for constructing a unary function call expression.

The predicate and accessor functions are easy to define for our current representation:

```
(defun unary-p (e)
  (and (listp e) (= (length e) 2)))

(defun unary-function (e) (first e))
```

and

```
(defun unary-argument (e) (second e))
```

This definition of the predicate `unary-p` is not perfect, but it will do for the present. The constructor `make-unary` is also easy:

```
(defun make-unary (fcn arg) (list fcn arg))
```

To make sure this new rule works, let's try a few examples:

```
> (deriv '(exp (* 3 x)) 'x)
(* (EXP (* 3 X)) 3)
> (deriv '(sin (* 3 x)) 'x)
(* (COS (* 3 X)) 3)
```

Using a Database of Rules

One aspect of this approach is not quite satisfactory. Adding functions is something you might want to do as you use `deriv`. Right now, this requires editing `make-unary-deriv`. An alternative is to set up a derivative database. Let's assume that we have a database containing rules for handling unary functions that we can query using a function `get-unary-rule`. The function `apply-unary-rule` can be used to apply a rule to an argument to produce a derivative expression evaluated at the argument. We can then write `make-unary-deriv` as

```
(defun make-unary-deriv (fcn arg)
  (apply-unary-rule (get-unary-rule fcn) arg))
```

To implement our database, we can use an *association list*. An association list is a list of lists. Each sublist starts with a symbol that serves as a key. The function `assoc` takes a key and an association list, and returns the first sublist matching the key, or `nil` if there is no match. A simple example is

```
> (def *mylist* '((x 1) (y "Hello") (abc a w (1 2 3))))
*MYLIST*
> (assoc 'x *mylist*)
(X 1)
> (assoc 'y *mylist*)
(Y "Hello")
> (assoc 'abc *mylist*)
(ABC A W (1 2 3))
> (assoc 'z *mylist*)
NIL
```

Let's use a global variable `*derivatives*` to hold our derivative database. Initially our database is empty, so we set this variable to `nil`:

```
(def *derivatives* nil)
```

To be able to add to this database, we can use the function `cons` to add an element to the front of a list. For example,

```
> (cons 'a '(b c))
(A B C)
```

Using `cons`, we can define a function `add-unary-rule` for adding rules to the database as

```
(defun add-unary-rule (f rule)
  (setf *derivatives* (cons (list f rule) *derivatives*)))
```

The retrieval function can be written as

```
(defun get-unary-rule (f)
  (let ((rule (assoc f *derivatives*)))
    (if rule
        rule
        (error "Can't differentiate this expression"))))
```

Now we can decide what to use for the rules. A simple choice is a function that takes a single argument, the derivative argument, and returns the derivative expression. Then the `apply-unary-rule` function is simply

```
(defun apply-unary-rule (entry arg)
  (funcall (second entry) arg))
```

and we can add a few rules to our database by using

```
(add-unary-rule 'exp #'(lambda (x) (make-unary 'exp x)))
```

and

```
(add-unary-rule 'sin #'(lambda (x) (make-unary 'cos x)))
```

Now we can look at some examples:

```
> (deriv '(exp x) 'x)
(EXP X)
> (deriv '(exp (* -1 (* x x))) 'x)
(* (EXP (* -1 (* X X))) (* -1 (+ X X)))
> (deriv '(sin (* 3 x)) 'x)
(* (COS (* 3 X)) 3)
> (deriv '(* (cos (* 3 x))) 'x)
error: Can't differentiate this expression
```

Our system cannot handle the final expression because it does not yet know how to differentiate the cosine. But once we add a rule for the cosine function as

```
(add-unary-rule 'cos
  #'(lambda (x)
      (make-product -1 (make-unary 'sin x))))
```

it can differentiate this expression as well:

```
> (deriv (deriv '(sin (* 3 x)) 'x) 'x)
(* (* (* -1 (SIN (* 3 X))) 3) 3)
```

Using a database of actions with the appropriate action determined by the data is called *data-directed programming*. The process of selecting an action for a piece of data is called *dispatching*.

Exercise 3.14

Write a function `print-infix` that takes an expression returned by `deriv` and prints it in fully parenthesized infix notation. For example, the result of

```
(print-infix '(+ x (* y z)))
```

should be

```
(X + (Y * Z))
```

You may find the functions `princ` and `terpri` useful. The function `princ` prints single Lisp expressions without *escape characters*. Escape characters are things like double quotes used around strings. Thus strings will be printed without quotes. `princ` returns its argument and does not print a *newline*. The function `terpri` prints a *newline*. You may also want to take advantage of the fact that function bodies and cond clauses can consist of several expressions.

Chapter 4

Additional Lisp Features

The previous chapter introduced the basics of Lisp programming. The emphasis in that chapter was on presenting programming techniques that are useful for writing Lisp functions. To make the most effective use of these techniques, it is helpful to be aware of the functions and data types provided by Lisp and Lisp-Stat. This chapter is intended to give an overview of these functions and data types. On first reading, you may wish to skim this chapter briefly, and then return to it as a reference when necessary.

4.1 Input/Output

The Common Lisp programming language contains an extensive set of tools for reading and writing files, formating output, and customizing the way input is read. A full discussion of these features would fill several chapters. Therefore I will only present those aspects I have found most useful for statistical programming. More extensive discussions can be found in several books on Common Lisp [31,56,59,67].

4.1.1 The Lisp Reader

The Lisp reader is responsible for converting characters typed by a user or read from a file into Lisp items. When a number is read, the reader is responsible for recognizing it as an integer or a floating point number and converting it to the appropriate internal representation. In Common Lisp there are several choices for the floating point type, specified by the symbols `short-float`, `single-float`, `double-float`, and `long-float`. The type used for reading a floating point number entered as 2.0, say, is controlled by the value of the global variable

`*read-default-float-format*`

which should be one of the four symbols listed above. It is also possible to specify explicitly that a number is to be interpreted as a double float, for example, by entering it as `2.d0`.¹

The reader is also responsible for interpreting some characters, or sequences of characters, in a special way. Such a character sequence, and the code used to interpret it, is called a *read macro*. We have already used several read macros: the quote character `'`, the backquote character `'`, the comma character `,`, and the pair `#'`. To understand how they work, we can enter quoted expressions containing these read macros into the interpreter and see what it returns:

```
> (quote 'x)
(QQUOTE X)
> (quote 'x)
(BACKQUOTE X)
> (quote ,x)
(COMMA X)
> (quote #'x)
(FUNCTION X)
```

When the reader sees a quote character `'`, for example, it reads the next Lisp expression, and returns a list consisting of the symbol `quote` and the expression.

Other read macros are the double quote character `"`, which reads the following characters up to the next `"` and forms them into a string, and the backslash character `\`, which is used to escape any special meaning associated with the following character. A backslash can be used, for example, to produce a string containing a double quote. We will encounter several other read macros below.

It is possible to define your own read macros. By doing this, you can customize the Lisp reader to provide a lexical analyzer for any language syntax you like.

4.1.2 Basic Printing Functions

Several functions are available for printing to the standard output or to a file. The most basic functions are `print`, `prin1`, `princ`, and `terpri`. `print` and `prin1` both print their argument in a form that is intended to be readable by the Lisp reader. If at all possible, the reader should be able to read anything printed by these functions. The difference between `print` and `prin1` is that the output of `print` is preceded by a *newline* and followed by a space. `princ` is like `prin1`, except it omits all *escape characters*. In particular, strings are printed without surrounding quotes. The function `terpri` outputs a *newline*.

¹The exact nature of these floating point types varies from one Lisp implementation to another. On many systems several of these types are identical. In XLISP-STAT all four floating point types are identical and correspond to the C double precision type.

To illustrate these functions, let's look at a function `printer` defined as

```
(defun printer ()
  (print "Hello")
  (prin1 "this")
  (princ " is a")
  (print "string")
  (terpri))
```

It produces the following output:

```
> (printer)
"Hello" "this" is a
"string"
NIL
```

The final `nil` is the result returned by `terpri`, and thus by `printer`.

4.1.3 Format

If you need more output control, you can use the `format` function to print directly to the standard output or to build up an output string. Let's look at building up a string for the moment. `format` has many variations, but I will only discuss a few that should be sufficient for most purposes.²

`format` is similar to the `sprintf` function in C. For forming a string, a `format` command looks like this:

```
(format nil <control-string> <arg-1> ... <arg-n>)
```

The control string contains *format directives* that begin with a tilde, ~. These are to be filled in using the remaining arguments.

The simplest format directive is ~a. It is replaced by the next argument, processed as it would be printed by `princ`. Here is an example:

```
> (format nil "Hello, ~a, how are you?" "Fred")
"Hello, Fred, how are you?"
```

The ~s directive is like ~a, except it uses `print`-style formatting. Using ~s, the greeting to Fred would look like

```
> (format nil "Hello, ~s, how are you?" "Fred")
"Hello, \"Fred\", how are you?"
```

The interpreter prints the result using `print`. It places a backslash before the embedded quotes to indicate that they are to be taken literally and do not indicate the end of the string.

²They are also the only ones currently supported by XLISP-STAT.

You can specify a field width in `~a` or `~s` directives by inserting an integer between the `~` and the `a` or `s`. The argument is then printed using a field of at least the specified length, longer if needed. The argument is left justified in the field:

```
> (format nil "Hello, ~10a, how are you?" "Fred")
"Hello, Fred      , how are you?"
```

If you do not know in advance how large a field you need, you can replace the integer by the letter `v`, in lower or upper case. `format` then expects to find an integer as its next argument, to be used in place of `v`:

```
> (format nil "Hello, ~va, how are you?" 10 "Fred")
"Hello, Fred      , how are you?"
```

To print numbers, you can use the `~d` directive for integers, and `~f`, `~e`, or `~g` for floating point numbers. The `~f` directive uses decimal notation, `~e` uses exponential notation, and `~g` uses `~e` or `~f`, whichever is shorter. You can specify field widths for these directives; numbers are right justified within their fields. You can specify the number of digits after the decimal point for the three floating point directives by adding another integer preceded by a comma. For example, to print in decimal notation using a field width of 10 and 3 digits after the decimal point, you would use the directive `~10,3f`. You can omit the field length by using just `~,3f`, and you can replace one or both numbers by the letter `v` and include the appropriate integer as an argument. Here are some examples:

```
> (format nil "The value is: ~,3g" 1.23456)
"The value is: 1.235"
> (format nil "~10,2e, ~v,vf" 123.456 10 3 56.789)
" 1.23e+002,      56.789"
```

Two more format directives are `~~` to insert a `~` character and `~%` to insert a *newline*. Finally, the directive consisting of a tilde `~` followed by a *newline* causes `format` to ignore the *newline* and any following white space up to the first non white space character. This directive is useful for continuing long format strings onto a second line:

```
> (format nil "A format string ~
written on two lines")
"A format string written on two lines"
```

Using `format` to build up a string is useful for constructing labels for plots. As an example, in Section 3.6.4 I defined a function `plot-expr` to plot the value of an expression for a range of values of a particular variable. We can now modify this function to construct variable labels for the plot by using the expression and the variable symbol:

```
(defun plot Expr (Var Low High)
  (flet ((f (X) (eval `(let ((,Var ,X)) ,Expr)))
         (S (X) (format nil "~s" X)))
    (plot-function #'f Low High
                  :labels (list (S Var) (S Expr)))))
```

The local function `S` converts its argument to a string using the `~s` format directive. The `:labels` keyword argument to `plot-function` takes a list of strings to use as the axis labels.

`format` does not have to be used only to build up strings. If the first argument of `format` is `t` instead of `nil`, `format` prints to the standard output and returns `nil`. For example, you can use `format` to print a small table as

```
> (format t
           "A Small Table~%~10,3g ~10,3g~%~10,3g ~10,3g~%"
           1.2 3.1 4.7 5.3)
A Small Table
 1.2      3.1
 4.7      5.3
NIL
```

4.1.4 Files and Streams

Lisp I/O functions read from and write to *streams*. The default stream for output is the stream bound to the global variable `*standard-output*`; the default input stream is `*standard-input*`. `print` can be told to use a different stream by giving it a stream as a second argument. `format` can also be instructed to use a stream by giving it a stream as its first argument instead of `t` or `nil`.

File Streams

Streams can be formed in several ways. One way is to open a file with the `open` function. `open` takes a file name as its argument and returns a stream. By default, the stream is an input stream. To open a file for output, you can give the `open` command an additional keyword argument `:direction`. Its value should be another keyword: `:input` for an input file, which is the default, and `:output` for an output file. Here is how you would write a few simple expressions to a file named "`myfile`". If a file by this name already exists, it is overwritten. Otherwise, a new file is created:

```
> (setf f (open "myfile" :direction :output))
#<File-Stream: #34a1a2>
> (print '(+ 1 2 3) f)
(+ 1 2 3)
> (format f "~%~s~%" 'Hello)
NIL
```

```
> (close f)
NIL
```

The function `close` closes a file stream. Since most systems limit the number of files that can be open at the same time, it is important to remember to close files once they are no longer needed.

The contents of the file just created looks like this:

```
(+ 1 2 3)
HELLO
```

We can use the `read` function to read the contents of this file back into Lisp:

```
> (setf f (open "myfile"))
#<File-Stream: #33f44e>
> (read f)
(+ 1 2 3)
> (read f)
HELLO
```

`read` reads in one expression at a time from the specified stream.

If the stream is empty, or the end of the file has been reached, `read` signals an error:

```
> (read f)
error: end of file on read
```

To be able to detect the end of a file without causing an error, we can give `read` a second argument with value `nil`. We can also give `read` a third argument, a Lisp item to return when an end of file condition occurs. If this third argument is not supplied, it defaults to `nil`:

```
> (read f nil)
NIL
> (read f nil '*eof*)
*EOF*
```

The pattern of opening a file, assigning its stream to a variable, performing some operations, and then closing the stream is quite common. Lisp therefore provides the macro `with-open-file` to simplify this process. This macro is called as

```
(with-open-file ((stream) (open args)) (body))
```

The parameter `(stream)` is a symbol used to refer to the stream, `(open args)` are the arguments to the `open` function, and `(body)` represents the expressions to be evaluated using the stream. To write our file using `with-open-file`, we would use

```
(with-open-file (f "myfile" :direction :output)
  (print '(+ 1 2 3) f)
  (format f "~%~s~%" 'Hello))
```

The `with-open-file` macro closes the file stream before returning. In particular, it ensures that the file stream will be closed even if an error occurs while evaluating one of the expressions in its body.

Two functions that are occasionally useful in dealing with file I/O are `force-output` and `file-position`. `force-output` takes an optional stream argument and flushes the output buffer for the stream. If no argument is supplied, the stream defaults to the standard output. `file-position` takes a stream and an optional integer argument. The stream should be attached to a file. Called with only the stream argument, `file-position` returns the current position of the file pointer for the file stream. When given an integer as a second argument, it sets the file pointer to the position specified by that integer. This can be used to implement file rewinding and random access file I/O.

`format`, `print` and `read` are high-level I/O functions that deal with Lisp items. The functions `read-char` and `write-char` are available if you need character level I/O.

String Streams

Streams can also be constructed to build up and decompose strings. Suppose you have obtained a string containing a Lisp expression, like "`(+ 1 2)`". Such a string might be obtained from a text field in a dialog window. You can read the expression from the string using the macro `with-input-from-string`. This macro is called as

```
(with-input-from-string ((stream) (string)) (body))
```

It creates a *string input stream* using the string `<string>`, binds it to the symbol `<stream>`, and evaluates the expressions in `<body>` with this binding. The result returned is the result of evaluating the last expression in `<body>`. You can extract the expression from our string using

```
> (with-input-from-string (s "(+ 1 2)") (read s))
(+ 1 2)
```

The macro `with-output-to-string` allows you to build up a string using `print` and other output functions. It is called as

```
(with-output-to-string ((stream)) (body))
```

It creates a *string output stream*, binds it to the symbol `<stream>`, evaluates the `<body>` expressions with this binding, and returns the string contained in the stream after all expressions have been processed:

```
> (with-output-to-string (s) (prin1 '(+ 1 2) s))
"(+ 1 2)"
```

4.2 Defining More Flexible Functions

Several of the built-in functions we encountered so far take optional arguments, or keyword arguments, or allow an arbitrary number of arguments to be used. You can incorporate these features in your own functions as well.

4.2.1 Keyword Arguments

You can define a function with keyword arguments by placing `&key` in its argument list. All arguments following `&key` are expected to be supplied as keyword arguments. As an example, suppose you would like a version of the recursive factorial function that prints the current value of the argument as the recursion proceeds. This can be implemented using a keyword argument:

```
(defun factorial (n &key print)
  (if print (format t "n = ~d~%" n))
  (if (= n 0) 1 (* (factorial (- n 1) :print print) n)))
```

Without the keyword argument, `factorial` simply computes the factorial and returns it,

```
> (factorial 3)
6
```

but with a non-nil value for the argument it prints the additional information:

```
> (factorial 3 :print t)
n = 3
n = 2
n = 1
n = 0
6
```

A keyword argument is optional. If it is not supplied, it defaults to `nil`. You can give it an alternative default value by giving a list of the argument symbol and an expression for its default value instead of just a symbol. If we define `factorial` as

```
(defun factorial (n &key (print t))
  (if print (format t "n = ~d~%" n))
  (if (= n 0) 1 (* (factorial (- n 1) :print print) n)))
```

then the default is to print the arguments, and printing has to be turned off by supplying the keyword argument `:print` with value `nil`:

```
> (factorial 3)
n = 3
n = 2
```

```
n = 1
n = 0
6
> (factorial 3 :print nil)
6
```

Even if you are happy with the default `nil`, you may sometimes like to be able to distinguish between an omitted keyword value and one that was supplied with value `nil`. This can be done by adding another symbol to the list of the keyword's symbol and its default expression. This symbol, called a *supplied-p argument*, is set to `t` if the keyword is supplied and to `nil` if it is not. As a simple example, let's define

```
(defun is-it-there (&key (mykey nil is-it)) is-it)
```

and try this function with and without the keyword argument:

```
> (is-it-there :mykey nil)
T
> (is-it-there)
NIL
```

By default, the keyword used to supply a keyword argument is just the argument symbol preceded by a colon. At times it may be useful to use a long keyword, but to be able to refer to the argument using a short symbol in the function body. We can supply an alternate keyword-parameter pair by replacing the argument symbol in the form including a default argument by a list of a keyword and a symbol. In our factorial example, to use the keyword `:print` but call the argument `pr`, we could use

```
(defun factorial (n &key ((:print pr) t))
  (if pr (format t "n = ~d~%" n))
  (if (= n 0) 1 (* (factorial (- n 1) :print pr) n)))
```

The environment in which default expressions for keyword arguments are evaluated consists of the environment in which the function is defined, together with bindings for all arguments in the function argument list preceding the argument being initialized. The initialization process can be thought of as implicitly using a `let*` construct.

A function can be defined to take more than one keyword argument. Keyword arguments may be supplied in any order, following the required arguments.

4.2.2 Optional Arguments

Instead of using a keyword in our factorial example we could have used an optional argument. The approach is similar: the symbol `&optional` is used in the argument list instead of `&key`:

```
(defun factorial (n &optional print)
  (if print (format t "n = ~d~%" n))
  (if (= n 0) 1 (* (factorial (- n 1) print) n)))
```

Without the optional argument, `factorial` just returns its result:

```
> (factorial 3)
6
```

With a non-`nil` value for the argument, it prints the argument for each call:

```
> (factorial 3 t)
n = 3
n = 2
n = 1
n = 0
6
```

Like keyword arguments, optional arguments default to `nil` but can be given an alternate default value. The method is the same as used for keyword arguments. To have the value of `print` default to `t`, we would use

```
(defun factorial (n &optional (print t))
  (if print (format t "n = ~d~%" n))
  (if (= n 0) 1 (* (factorial (- n 1) print) n)))
```

Supplied-p arguments also work as for keyword arguments.

Unlike keyword arguments, if a function takes more than one optional argument, then the optional arguments must be supplied in the order in which they are specified in the definition. If both optional and keyword arguments are used in a function, then the optional arguments must precede the keyword arguments in the definition. In using a function that takes both optional and keyword arguments, you must specify all optional arguments if you are going to give any keyword arguments.

4.2.3 Variable Number of Arguments

The simple vector addition function `vec+` defined in Section 3.7 only allows two arguments, while the built-in function `+` allows arbitrarily many arguments. We can modify the definition of `vec+` to allow an arbitrary number of arguments by using the `&rest` symbol in the argument list. If this symbol is included in an argument list of a function, then the remaining arguments in a call to the function will be formed into a list and bound to the symbol following the `&rest` symbol. Here is a modified version of `vec+:`

```
(defun vec+ (&rest args)
  (if (some #'compound-data-p args)
      (apply #'map-elements #'vec+ args)
      (apply #'+ args)))
```

In the body of the function the variable `args` refers to the list of all the arguments used in the call to `vec+`. The function `some` takes a predicate function and one or more lists, and maps the predicate down the list or lists until it returns a non-`nil` result or runs out of elements. This definition also uses the more elaborate version of `apply` described in Section 3.6.

Optional, rest, and keyword arguments may all appear in a single argument list. If more than one of them appears, then they must appear in the order `&optional`, `&rest`, `&key`.

4.3 Control Structure

4.3.1 Conditional Evaluation

We have already made extensive use of the conditional evaluation constructs `cond` and `if`. Another conditional evaluation construct, introduced briefly in Section 3.10.2, is `case`. `case` can be used to select actions according to a discrete set of values returned by an expression. The general form of a `case` expression is

```
(case <key expr>
  ((<key 1> <expr 1>)
   ...
   (<key n> <expr n>))
```

The `<key expr>` is evaluated and compared to the `<key i>` expressions one at a time. The `<key i>` expressions may be numbers, symbols, characters, or lists of numbers, symbols, or characters. If the value of the `<key expr>` matches a key or an element of a key list, then the corresponding consequent expression is evaluated, and its result is returned as the result of the `case` expression. Like `cond` clauses, `case` clauses may contain a series of consequent expressions. These are evaluated one at a time, and the result of the last evaluation is returned. The final `case` clause may start with the symbol `t`; in this case the final clause will be used if none of the other clauses applies. Without a default clause, the `case` expression returns `nil` if none of the clauses is matched. As a simple example, an expression like

```
(case choice
  ((ok delete) (delete-file))
  (cancel (restore-file)))
```

might be used to respond to a choice made by a user selecting an item from a menu or pressing buttons in a dialog window.

Occasionally it is useful to be able to evaluate a series of expressions if a predicate is or is not true. This can be accomplished using `cond`, but Lisp also provides two simpler alternatives called `when` and `unless`. The expressions

```
(when <p> <expr 1> ... <expr n>)
```

and

```
(unless ⟨p⟩ ⟨expr 1⟩ ... ⟨expr n⟩)
```

are equivalent to

```
(cond ⟨⟨p⟩ ⟨expr 1⟩ ... ⟨expr n⟩⟩)
```

and

```
(cond ⟨⟨p⟩ nil)
      (t ⟨expr 1⟩ ... ⟨expr n⟩))
```

respectively.

4.3.2 Looping

The most flexible Lisp iteration construct currently available is `do`, introduced in Section 3.4.³ `do` actually has a few more options than shown so far. In particular, it allows a series of result expressions to follow the termination test, and it also allows a series of expressions to be given as the body of the loop. The body is executed on each iteration using the current bindings of the `do` variables. The most general form of a `do` expression is

```
(do ((⟨name 1⟩ ⟨initial 1⟩ ⟨update 1⟩)
     ...
     ⟨⟨name n⟩ ⟨initial n⟩ ⟨update n⟩⟩)
    (⟨test⟩ ⟨result 1⟩ ... ⟨result k⟩)
    ⟨expr 1⟩
    ...
    ⟨expr m⟩)
```

The additional result and body expressions are useful for producing side effects such as assignments or printing. For example, we could modify the definition of `my-sqrt` in Section 3.4 to

```
(defun my-sqrt (x &key print)
  (do ((guess 1 (improve guess x)))
    ((good-enough-p guess x)
     (if print (format t "Final guess: ~g~%" guess))
     guess)
    (if print (format t "Current guess: ~g~%" guess))))
```

in order to provide some information on the iteration process:

³At the time of writing, a more powerful looping system is under consideration for incorporation into the Common Lisp standard [56].

```
> (my-sqrt 3 :print t)
Current guess: 1
Current guess: 2
Current guess: 1.75
Final guess: 1.732143
1.732143
```

The variable bindings set up by `do` are set up in parallel, like the bindings for local variables set up by `let`. The macro `do*` is like `do`, except that it constructs its bindings sequentially. It is thus analogous to `let*`.

Several simpler iteration constructs are available as well. Two such constructs are `dotimes` and `dolist`, introduced briefly in Section 2.5.6. The macro `dotimes` is called as

```
(dotimes ((var) (count) (result)) (body))
```

When a `dotimes` expression is evaluated, the `(count)` expression is evaluated first. It should evaluate to an integer. Then the expressions in `(body)` are evaluated once for each integer from zero (inclusive) to the value of `(count)` (exclusive), in order, with the variable `(var)` bound to the integer. If the result of evaluating `(count)` is zero or negative, the expressions in `(body)` are not evaluated. Finally, the `(result)` expression is evaluated, and the result is returned. The `(result)` expression is optional; if it is not supplied, the value of the `dotimes` expression defaults to `nil`.

Using the `result` expression in `dotimes`, we can modify the translation of the C version of the factorial function given in Section 3.8 as

```
(defun factorial (n)
  (let ((n-fac 1))
    (dotimes (i n n-fac)
      (setf n-fac (* n-fac (+ i 1))))))
```

The `dolist` macro is similar to `dotimes`. It is called as

```
(dolist ((var) (list) (result)) (body))
```

and repeats its body once for each element of `(list)`, with the variable `(var)` bound to each successive element. Using `dolist`, you can write a simple function for adding up the elements of a list as

```
(defun sum-list (x)
  (let ((sum 0))
    (dolist (i x sum)
      (setf sum (+ sum i)))))
```

A very simple iteration construct is `loop`. The general form is

```
(loop (body))
```

It repeats indefinitely, executing each expression in *<body>* in sequence each time through the loop. To stop the loop, you can include a `return` expression in the body. `return` takes an optional argument that is returned as the value of the loop expression; the default is `nil`. Yet another version of factorial:

```
(defun factorial (n)
  (let ((i 1)
        (n-fac 1))
    (loop (if (> i n) (return n-fac))
          (setf n-fac (* i n-fac))
          (setf i (+ i 1)))))
```

`return` can be used within `dotimes`, `dolist`, `do`, and `do*` as well. If one of these loops is terminated by a call to `return`, then the standard return expressions are not evaluated.

4.4 Basic Lisp Data and Functions

Up to now, the only Lisp data types we have used extensively are numbers, strings, symbols and lists. This section reviews these data types and introduces several new ones, in particular vectors and multi-dimensional arrays.

4.4.1 Numbers

Lisp provides several different types of numbers, including integers, floating point numbers, and complex numbers. Common Lisp also includes a rational number data type.⁴ The predicate `numberp` returns `t` for any Lisp number. The predicates `integerp` and `floatp` can be used to recognize integers and floating point numbers. The predicate `complexp` identifies complex numbers.

Conversion among Real Number Types

The functions `floor`, `ceiling`, `truncate`, and `round` can be used to convert floating point numbers or rationals to nearby integers. When applied to integers, they simply return their arguments. All four functions are vectorized in Lisp-Stat. `floor` always rounds down, while `truncate` rounds toward zero.

It is possible to coerce integers and rationals to floating point numbers using the function `float`. If `float` is applied to a floating point number, then that number is simply returned. In Common Lisp, if `float` is applied to an integer or a rational, then, by default, a `single-float` floating point number is returned. To coerce to a floating point number of a different type, you can give `float` a second argument, a number of the desired type. Thus the expression

⁴At present XLISP-STAT does not support rational numbers.

```
(float 1 0.d0)
```

returns a **double-float**. In Lisp-Stat the **float** function is vectorized, so

```
(float (list 1 2 3) 0.d0)
```

returns a list of **double-float** numbers.

In Common Lisp, integers and rationals are coerced to floating point numbers of type **single-float** when given as arguments to the transcendental functions such as **log** and **sin**. If this floating point type does not offer enough precision on your Lisp system, you should explicitly coerce your data to double precision numbers before using these functions.⁵

Complex Numbers

Complex numbers are printed using a representation of the form

```
#C(<realpart> <imagpart>)
```

with **<realpart>** representing the real part of the number and **<imagpart>** the imaginary part. For example,

```
> (sqrt -1)
#C(0 1)
```

To construct a complex number directly, you can either use the function **complex**, as in

```
> (complex 3 7)
#C(3 7)
```

or you can enter the printed representation,

```
> #c(3 7)
#C(3 7)
```

In Lisp-Stat the function **complex** is vectorized.

There is a subtle difference between these two approaches. The function **complex** is an ordinary function. Thus we can call it with expressions as its arguments:

```
> (complex (- 2 3) 1)
#C(-1 1)
```

⁵A Lisp-Stat implementation may provide a mechanism for automatically coercing rationals to double precision numbers before applying the transcendental functions. Such a mechanism is not needed in XLISP-STAT, since all floating point types are identical to the double precision type.

In contrast, the pair of characters `#C`, or `#c`, form a read macro. This read macro instructs the Lisp reader to take the next expression, which must be a list of two real numbers, and construct a complex number using those numbers as the real and imaginary parts, respectively. The list is taken literally, without evaluating it or its arguments. As a result, the real and imaginary parts have to be given as numbers, not as expressions that evaluate to numbers. Trying to give an expression instead of a number produces an error:

```
> #c((- 2 3) 1)
error: not a number - (- 2 3)
```

We will encounter a few other read macros in this chapter. They all share this feature of not evaluating their arguments.

Like reals, complex numbers can be integers, rationals, or floating point numbers. If either of the two arguments to `complex` is a floating point number, the other is converted to a floating point number, and the result is a complex floating point number. If both arguments are integers, the result is a complex integer. Complex integers always have a nonzero imaginary part. Complex integers with an imaginary part of zero are always converted to real integers. Complex floating point numbers, on the other hand, may have an imaginary part that is zero:

```
> #c(1 0)
1
> #c(1.0 0.0)
#C(1 0)
```

The components of a complex number's cartesian or polar representation can be extracted using the functions `realpart`, `imagpart`, `phase`, and `abs`:

```
> (realpart #c(3 4))
3
> (imagpart #c(3 4))
4
> (phase #c(3 4))
0.9272952
> (abs #c(3 4))
5
```

These functions are vectorized in Lisp-Stat.

4.4.2 Strings and Characters

We have already used strings, written as sequences of characters enclosed in double quotes, in a number of examples. Characters are the elements making up a string. They are treated as a special kind of data, distinct from

single element strings or numerical character codes. To see how a character is printed by Lisp, we can use the function `char` to extract a character from a string. As is usual in Lisp, indexing starts at zero. Thus the characters in the string "Hello" are numbered 0, 1, 2, 3, and 4:

```
> (char "Hello" 0)
#\H
> (char "Hello" 1)
#\e
```

The characters `#\` form a read macro for reading and printing characters, much like the one used above for complex numbers. Here are some more examples:

```
> #\a
#\a
> #\3
#\3
> #\:
#\:
> #\newline
#\newline
```

Standard printable characters, like letters, digits, or punctuation marks, are represented by `#\` followed by the appropriate character. Special characters, like the *newline* character, often have names more than one character long. Some systems have additional nonstandard characters not found on other systems. In XLISP-STAT on the Apple Macintosh, for example, the *apple* character is represented by `#\Apple`.

A character can be converted to a single character string using the `string` function:

```
> (string #\H)
"H"
```

Strings and characters can be recognized using the predicates `stringp` and `characterp`, respectively.

4.4.3 Symbols

Unlike identifiers in most other languages, Lisp symbols are real physical objects taking up a certain amount of space in computer memory. A symbol has four parts:

- a print name
- a value cell
- a function cell

- a property list

Since I will not use property lists in the remainder of this book, I will not discuss them further.

The value cell of a symbol contains the value of the global variable named by the symbol. If the value cell does not contain a value, the symbol is said to be *unbound*, and an error is signaled when an attempt is made to retrieve the value. The predicate `boundp` can be used to determine if a symbol has a global value. The function cell of a symbol contains the symbol's global function definition, if it has one. The predicate `fboundp` determines whether a symbol has a global function definition.

The contents of a symbol's cells can be extracted by using the functions `symbol-name`, `symbol-value`, and `symbol-function`:

```
> (symbol-name 'pi)
"PI"
> (symbol-value 'pi)
3.141593
> (symbol-function 'pi)
error: unbound function - PI
```

The symbol `pi` does not have a function definition, so an error is signaled when its function cell is accessed.

The accessor functions `symbol-name`, `symbol-value`, and `symbol-function` can also be used as generalized variables with `setf`. For example, we can set the value of a symbol,

```
> (setf (symbol-value 'x) 2)
2
> x
2
```

or the function definition of a symbol:

```
> (setf (symbol-function 'f) #'(lambda (x) (+ x 1)))
#<Closure: #2b20c0>
> (f 1)
2
```

In using symbols as labels for variables or functions, we often think of them as being equivalent to their print names. For most purposes this is reasonable. When the reader needs a symbol for a particular print name, it only constructs a new symbol if one by that name does not exist yet. As a result, symbols with identical print names should be `eq`. It is, however, possible to use `setf` and `symbol-name` to change a symbol's print name. This may cause the reader to get confused and lead to unpredictable results. It should be avoided.

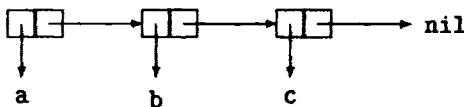


Figure 4.1: Diagram of cons cells for the list (a b c).

4.4.4 Lists

Lists are the basic tool for constructing complex data structures from more elementary data. While their structure is sequential, they can also be used to represent unordered collections, or sets, of items.

Basic List Structure

The most basic list is the empty list `nil`, also called the *null list*. It can be written as `nil` or as `()`. It evaluates to itself:

```
> nil
NIL
> ()
NIL
```

A nonempty list is built up of a series of constructor cells, or *cons cells*. A cons cell can be viewed as containing two fields. One field points to an element of a list. The other points to the rest of the list, or rather to the next cons cell in the rest of the list, or to `nil`. Figure 4.1 shows a diagram of the cons cells making up a list of the form (a b c).

The contents of the two cons cell fields can be extracted using the functions `first` and `rest`⁶:

```
> (first '(a b c))
A
> (rest '(a b c))
(B C)
> (rest (rest '(a b c)))
(C)
```

⁶The first element and the rest of a list are also called the list's *car* and *cdr*, respectively. They can be extracted using the functions `car` and `cdr` instead of `first` and `rest`. These strange names have their origin in the first implementation of Lisp on an IBM 704 in which the first element of a list was the Contents of the Address Register and the rest was the Contents of the Decrement Register.

```
> (rest (rest (rest '(a b c))))
NIL
```

The function `cons` constructs a new cons cell with fields pointing to its two arguments:

```
> (cons 'a '(b c))
(A B C)
> (cons 'C nil)
(C)
> (cons 'a (cons 'b (cons 'c nil)))
(A B C)
```

`cons` only constructs one new cons cell; it does not copy the list in its second argument. If we construct a list `x` as

```
(setf x '(a b c))
```

and construct a second list `y` as

```
(setf y (cons 'd (rest x)))
```

then the lists `x` and `y` only differ in their first cons cells. Their rests consist of identical cons cells and are therefore `eq`:

```
> (eq x y)
NIL
> (eq (rest x) (rest y))
T
```

Thus destructively changing the first element of `x` will not affect `y`, but changing the second or third elements of `x` will change the corresponding elements of `y`.

The function `append`, introduced in Section 2.4.4, reuses the cons cells in its final argument. That is, the final three cons cells of the result returned by

```
(append '(1 2) y)
```

are identical to the cons cells making up `y`:

```
> (append '(1 2) y)
(1 2 D B C)
> (eq y (rest (rest (append '(1 2) y))))
T
```

The main advantage in reusing cons cells as done by `append` is that it conserves memory. A drawback is that destructive modification of lists can have unexpected side effects.

Lists as Sets

Several functions are available for using lists to represent sets of items. The function `member` tests whether an element is in a list or not:

```
> (member 'b '(a b c))
(B C)
> (member 'd '(a b c))
NIL
```

If the element is found, then, instead of returning the symbol `t`, `member` returns the remainder of the list, starting with the first appearance of the element in question. Since Lisp interprets any result other than `nil` as *true*, this does not cause any problems and it is often useful.

`adjoin` adds an element to the front of a list, unless the element is already in the list:

```
> (adjoin 'd '(a b c))
(D A B C)
> (adjoin 'd '(d a b c))
(D A B C)
```

`union` and `intersection` return lists of the union and intersection of the elements in their two arguments, respectively:

```
> (union '(a b c) '(c d))
(D A B C)
> (intersection '(a b c) '(c d))
(C)
```

As long as neither of the arguments contains any duplicates, the results will not contain any duplicates.

`set-difference` returns a list of the elements of its first argument not contained in its second argument:

```
> (set-difference '(a b c) '(c d))
(B A)
```

None of the functions `union`, `intersection`, or `set-difference` are guaranteed to preserve the order in which the elements appear in their arguments.

All of these functions need to be able to determine whether two elements of a list are to be considered equal or not. By default, they use the predicate `eql` to test for equality, but you can specify an alternate test using the keyword `:test`. For example,

```
(member 1.0 '(1 2 3))
```

is false since the integer 1 and the floating point number 1.0 are not `eql`. On the other hand,

```
(member 1.0 '(1 2 3) :test #'equalp)
```

is true since 1 and 1.0 are `equalp`. You can supply any function of two arguments as the test predicate.

Several other functions requiring the comparison of Lisp items also allow an alternate predicate to be specified with the `:test` keyword. Two examples are the functions `subst` and `remove`.

4.4.5 Vectors

So far, we have represented numerical data sets as lists. We can also represent them as Lisp vectors. A vector can be constructed by using an expression like

```
> '#(1 2 3)
 #(1 2 3)
```

The characters `#(` are the read macro indicating the start of a vector. The following expressions, up to the matching `)`, are read in, without evaluation, and formed into a vector by the reader. The quote is needed since it is not meaningful to evaluate a vector.⁷

You can also use the `vector` function to make a vector of its arguments:

```
> (vector 1 2 3)
 #(1 2 3)
```

The function `length` returns the length of a vector. Elements can be extracted using `select` or `elt`. The predicate `vectorp` can be used to test if a Lisp item is a vector or not:

```
> (setf x (vector 1 'a #\b))
 #(1 A #\b)
 > (length x)
 3
 > (elt x 1)
 A
 > (select x 1)
 A
 > (vectorp x)
 T
```

Unlike `select`, `elt` does not allow a subvector to be extracted by giving it a list or vector of indices. `elt` may, however, be more efficient than `select` in many implementations. Thus you may want to use `elt` in code that is to be optimized for speed.

⁷In some Lisp systems vectors may evaluate to themselves, but in others attempting to evaluate a vector produces an error.

elt and **select** can both be used as generalized variables with **setf**.

It is not clear whether it is better to represent data sets as lists or vectors. Vectors do offer some advantages over lists. In particular, the time needed to access a particular element of a vector is independent of the position the element. In contrast, elements of a list can only be accessed by working through the cons cells of the list until the element is found. A function that needs to access elements of a data set in random order will therefore run faster if it represents its data set as a vector. On the other hand, Lisp provides a few more functions for dealing with lists than it does for vectors. In some Lisp systems lists may also make more efficient use of memory than vectors.

4.4.6 Sequences

Common Lisp has a number of functions that can be applied to lists, vectors, or strings.⁸ The function **length** is one example:

```
> (length '(1 2 3))
3
> (length '#(a b c d))
4
> (length "Hello")
5
```

The term *sequence* is used to describe the union of these three data types.

The function **elt** can be used to extract an element of a sequence:

```
> (elt '(1 2 3) 1)
2
> (elt '#(a b c d) 2)
C
> (elt "Hello" 4)
#\o
```

You can coerce a sequence of one type to another using the function **coerce**. For example,

```
> (coerce '(1 2 3) 'vector)
#(1 2 3)
```

If the first argument given to **coerce** is already of the specified type, it may or may not be copied, depending on the Lisp system. If you want to be sure you have made a copy of a sequence, you can use the function **copy-seq**. This function only copies the sequence, not the elements. The original sequence and the copy will be elementwise **eq**. If the result type symbol for **coerce** is

⁸In XLISP-STAT, several of these functions only work for lists, or only for lists and vectors.

string but the elements of the sequence are not all characters, **coerce** will signal an error.

concatenate takes a result type symbol and an arbitrary number of sequences as arguments. It returns a new sequence containing all elements of the argument sequences in order:

```
> (concatenate 'list '(1 2 3) #(4 5))
(1 2 3 4 5)
> (concatenate 'vector '(1 2 3) #(4 5))
#(1 2 3 4 5)
> (concatenate 'string '(#\a #\b #\c) #(#\d #\e) "fgh")
"abcdefgh"
```

Unlike **append**, **concatenate** will not reuse parts of lists – all sequences are copied. Again, if the result type symbol is **string** but the elements of the sequence are not all characters, **concatenate** will signal an error.

The function **map** can be used to map a function across lists or vectors. Its first argument must be a symbol specifying the type of the result:

```
> (map 'list #'+ (list 1 2 3) #(4 5 6))
(5 7 9)
> (map 'vector #'+ (list 1 2 3) #(4 5 6))
#(5 7 9)
```

The arguments following the function may be lists, vectors, or strings, in any combination.

The function **some** applies a predicate elementwise to one or more sequences until the predicate is satisfied or the shortest sequence is exhausted:

```
> (some #'< '(1 2 3) '(1 2 4))
T
> (some #'> '(1 2 3) '(1 2 4))
NIL
```

The function **every** is analogous.

Another useful sequence function is **reduce**. This function takes a binary function and a sequence as arguments, and combines the elements of the sequence using the binary function. For example, you can add up the elements of a sequence using

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

You can give **reduce** an initial value by using the **:initial-value** keyword. Using **reduce** with this keyword, you can reverse the elements in a list:

```
> (reduce #'(lambda (x y) (cons y x))
          '(1 2 3)
          :initial-value nil)
(3 2 1)
```

The first argument to the reducing function is the reduction obtained so far; the second argument is the next element in the sequence.

The function **reverse** returns a list of the elements of its argument in reverse order:

```
> (reverse '(a b c))  
(C B A)
```

The function **remove** takes an element and a sequence, and returns a new sequence consisting of all elements not equal to the first argument, in the order in which they appeared in the original sequence. **remove-duplicates** takes a sequence and returns a new sequence with all duplicates removed. If an element occurs more than once in the argument sequence, then all occurrences but the last are removed. The default equality test used by these functions is **eql**; an alternative can be supplied using the **:test** keyword.

The function **remove-if** takes a single argument predicate and a sequence, and returns a new sequence with all elements satisfying the predicate removed. For example,

```
> (remove-if #'(lambda (x) (<= x 0)) '(2 -1 3 0))  
(2 3)
```

remove-if-not is analogous; it merely reverses the result of the predicate.

Functions with names beginning with **remove** are nondestructive. They produce new sequences without modifying or reusing their arguments. Analogous functions with names starting with **delete** are destructive. They may modify or reuse their arguments and may thus be more efficient in their use of memory.

The function **find** takes an element and a sequence, and returns the first element of the sequence that matches its first argument, if there is one, and **nil** otherwise:

```
> (find 'b '(a b c))  
B  
> (find 'd '(a b c))  
NIL
```

position takes the same arguments and returns the index of the first element in the sequence matching its first argument, or **nil** if there is no match:

```
> (position 'b '(a b c))  
1  
> (position 'd '(a b c))  
NIL
```

Both **find** and **position** use **eql** by default but can be given an alternate equality predicate with the **:test** keyword.

Common Lisp provides a powerful sorting function called **sort**. This function requires two arguments, a sequence and a predicate. The predicate

should be a function that takes two arguments and returns a non-nil result only if its first argument is to be considered strictly less than its second argument. The `sort` function should be used with caution because it destructively sorts the sequence. That is, for efficiency reasons it will reuse parts of the argument sequence in the sorting algorithm, destroying the sequence in the process. Thus if you want to preserve the argument, you must copy it before passing it to the `sort` function. In contrast, the `sort-data` function introduced in Section 2.7.3 is not destructive.

4.4.7 Arrays

Vectors are one-dimensional arrays. Lisp also supports multi-dimensional arrays.

Array Construction

The `make-array` function is used to construct an array. It requires one argument, a list of the dimensions of the array. For example,

```
> (make-array '(2 3))
#2A((NIL NIL NIL) (NIL NIL NIL))
```

The result is a two-dimensional array with 2 rows and 3 columns, and all elements equal to `nil`. It is printed using a # followed by the rank, the letter A, and a list of the rows. A three-dimensional array can be created as

```
> (make-array '(2 3 4))
#3A(((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL))
     ((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)))
```

In the printed result the elements are split by the first dimension, then the second, then the third.

A vector can also be constructed using the `make-array` function:

```
> (make-array '(3))
#(NIL NIL NIL)
```

The dimension list for a vector only has one element. For convenience it can be given to `make-array` as a number:

```
> (make-array 3)
#(NIL NIL NIL)
```

`make-array` takes several keyword arguments. The `:initial-element` keyword can be used to construct an array with an initial element other than `nil`:

```
> (make-array '(2 3) :initial-element 1)
#2A((1 1 1) (1 1 1))
```

The keyword `:initial-contents` lets you specify the contents of a nonconstant array using a list like the one used in the printed representation:

```
> (make-array '(3 2) :initial-contents '((1 2)(3 4)(5 6)))
#2A((1 2) (3 4) (5 6))
```

Another keyword argument for `make-array` is `:displaced-to`. If this keyword is supplied, its argument must be an array with the same total number of elements as specified in the dimension list. The new array that is returned is a so-called *displaced*, or *shared*, array. It does not have its own data but shares its data with the array supplied as the argument. The difference between the arrays is the dimensions used in accessing the elements, printing the arrays, and so on. Here is an example:

```
> (setf x (make-array '(3 2)
                      :initial-contents '((1 2)(3 4)(5 6))))
#2A((1 2) (3 4) (5 6))
> (make-array '(2 3) :displaced-to x)
#2A((1 2 3) (4 5 6))
```

To understand this result, you need to know that the elements of an array are stored internally in row-major order.

The `:displaced-to` keyword is most useful for obtaining access to an array's data as a vector:

```
> (make-array 6 :displaced-to x)
#(1 2 3 4 5 6)
```

This can be used, for example, with the `map` function to construct a matrix addition function.

The printed form of multi-dimensional arrays can be read in by the reader. Thus you can construct an array by typing in the array as it would be printed:

```
> '#2a((1 2) (3 4) (5 6))
#2A((1 2) (3 4) (5 6))
```

Again a quote is needed, since it is not meaningful to evaluate an array. As with the vector read macro, the elements are not evaluated when an array is constructed in this way.

Array Access

The `aref` function can be used to extract elements of an array. Using the array `x` constructed above, we can extract the element in row 1 and column 0:

```
> x
#2A((1 2) (3 4) (5 6))
> (aref x 1 0)
3
```

`setf` can be used with `aref` to modify the elements of an array:

```
> (setf (aref x 1 0) 'a)
A
> x
#2A((1 2) (A 4) (5 6))
```

The Lisp-Stat function `select` can also be used to access array elements. `select` can be used like `aref`:

```
> x
#2A((1 2) (A 4) (5 6))
> (select x 1 0)
A
> (setf (select x 1 0) 'b)
B
```

`select` can also be given lists or vectors of integers as one or more of its index arguments to extract a subarray:

```
> (select x '(0 1) 1)
#2A((2) (4))
> (select x 1 '(0 1))
#2A((B 4))
> (select x '(0 1) '(0 1))
#2A((1 2) (B 4))
```

Subarrays can be modified by using `select` as a generalized variable with `setf`:

```
> (setf (select x '(0 1) 1) '#2a((x)(y)))
#2A((X) (Y))
> x
#2A((1 X) (B Y) (5 6))
```

The symbols in `#2a((x)(y))` are not quoted since the array read macro does not evaluate the array arguments as they are read in.

`aref` does not allow lists of indices but may be more efficient on some Lisp systems.

Additional Array Information

The `array-rank` function returns the rank of an array:

```
> (array-rank '#(1 2 3))
1
> (array-rank '#2a((1 2 3)(4 5 6)))
2
> (array-rank (make-array '(2 3 4)))
3
```

`array-dimension` returns the size of the specified dimension:

```
> (array-dimension '#(1 2 3) 0)
3
> (array-dimension '#2a((1 2 3)(4 5 6)) 0)
2
> (array-dimension '#2a((1 2 3)(4 5 6)) 1)
3
```

The entire dimension list can be obtained with `array-dimensions`:

```
> (array-dimensions '#(1 2 3))
(3)
> (array-dimensions '#2a((1 2 3)(4 5 6)))
(2 3)
```

`array-total-size` returns the number of elements in the array, and `array-in-bounds-p` determines if a set of indices is valid for an array:

```
> (array-total-size '#2a((1 2 3)(4 5 6)))
6
> (array-in-bounds-p '#2a((1 2 3)(4 5 6)) 2 1)
NIL
> (array-in-bounds-p '#2a((1 2 3)(4 5 6)) 1 2)
T
```

`array-row-major-index` takes an array and a valid set of indices, and returns the position of the specified element in the row-major ordering of the elements:

```
> (array-row-major-index '#2a((1 2 3)(4 5 6)) 1 2)
5
```

The element with indices 1 and 2 is the third element in the second row, the last element in the row-major ordering. Thus its row-major index is 5.

4.4.8 Other Data Types

In addition to the data types discussed in this section Lisp-Stat also has a data type called an *object* that is used for object-oriented programming. Objects are the subject of Chapter 6. Common Lisp also includes a *structure* type that is similar to C structures or Pascal records.

4.5 Odds and Ends

This section contains a collection of miscellaneous topics that did not fit conveniently in any of the previous sections.

4.5.1 Errors

When an error occurs during a calculation, Lisp enters a debugger, described below in Section 4.5.3, or resets the system and returns you to the interpreter. Lisp provides tools for signaling errors from your own functions, and for ensuring that certain cleanup operations are performed even if an error occurs in evaluating an expression.

Signaling Errors

Many built-in functions print reasonably informative error messages if, say, an argument is not of the proper type. You can use the function `error` to have your functions print such messages. `error` takes a format string argument, followed by any additional arguments required by the format directives. The error string should not contain a statement indicating that an error has occurred; the system will supply one. Here are some examples:

```
> (error "An error has occurred")
error: An error has occurred

> (error "Bad argument type - `a" 3)
error: Bad argument type - 3
```

Unwind-Protect

With the possibility of errors, it may be important to make sure that certain actions are taken before the system is reset after an error. For example, if an error occurs in handling an action in a dialog window, you may want to make sure the dialog is removed from the screen. `unwind-protect` is used to ensure that certain cleanup actions occur. The general form of a call to `unwind-protect` is

```
(unwind-protect <protected expr>
  (<cleanup expr 1>
   ...
   <cleanup expr n>))
```

`unwind-protect` evaluates `<protected expr>`, and then `<cleanup expr 1>`, ..., `<cleanup expr n>`. The cleanup expressions are executed even if a reset is called for after an error during the evaluation of `<protected expr>`. The cleanup expressions themselves are not protected further. The result returned by the `unwind-protect` expression is the result returned by the expression `<protected expr>`. As an example, an expression for handling a dialog and ensuring that the dialog is closed even if an error occurs might be written as

```
(unwind-protect (do-dialog) (close-dialog))
```

4.5.2 Code-Writing Support

Comments, Indentation, and Documentation

Comments in Lisp are preceded by a semicolon. All text on a line following a semicolon is ignored by the Lisp reader. Different numbers of semicolons are traditionally used for different levels of comments: four for headings, three for function definitions, two for headings within functions, and one for comments added to a line of code.

Comments are of course often used to remove sections of code temporarily. Having to add a semicolon to the beginning of each line is a nuisance, so a multiline commenting system exists: all text appearing between a `|#` and a `|#` is ignored, no matter how many lines it is on.

Lisp code, with its many parentheses, can be very hard to read if it is not indented in a way that reflects its structure. Good Lisp editors, like the *emacs* editor, provide facilities for automatically indenting a function according to certain conventions for indentation. The XLISP-STAT interpreter and editor windows on the Macintosh also include a simple indentation facility: hitting the *tab* key in a line indents the line to a reasonable level.

Lisp includes a facility for adding documentation to code as it is defined. If the first expression in the body of a `defun` is a string, and if there is more than one expression in the body, then that string is installed as the function documentation of the symbol used as the function name in the `defun`. These documentation strings can be retrieved using the `documentation` function or the Lisp-Stat `help` and `help*` functions. For example, to extract the function documentation for the symbol `mean`,

```
> (documentation 'mean 'function)
"Args: (x)
Returns the mean of the elements x. Vector reducing."
```

Variable and type documentation strings are extracted similarly:

```
> (documentation 'pi 'variable)
"The floating-point number that is approximately equal to the
ratio of the circumference of a circle to its diameter."
> (documentation 'complex 'type)
"A complex number"
```

Modules

The functions `require` and `provide` are useful for breaking up function definitions into multiple files. The function `provide` adds its argument, a string, to the list `*modules*` if it is not already there. The function `require` checks if its argument is already in the list `*modules*`. If it is, it does nothing. Otherwise, it attempts to load its argument. `require` can be given a second

argument to use as the name of the file to be loaded. If this second argument is not provided, a name is constructed from the module's name.⁹

To implement a system of modules using separate files, place a **provide** expression at the beginning of a file, followed by **require** expressions for each of the modules the file depends on. Loading the main file of interest should then bring in all other files, without loading any files required by more than one module twice.

Adapting to System Features

Common Lisp provides a mechanism for detecting whether a system has particular features, and for evaluating expressions only when certain features, or combinations of features, are present.

The global variable ***features*** contains a list of features for a particular system. In XLISP-STAT on a Macintosh with a color monitor the value of this variable might be

```
> *features*
(WINDOWS DIALOGS COLOR MACINTOSH XLISP)
```

The read macros **#+** and **#-** can be used to evaluate expressions if certain combinations of features are present or are absent, respectively. For example, if the reader is given

```
#+color (add-color)
```

then the expression **(add-color)** will only be evaluated if the symbol **color** is in the features list. In

```
#-color (fake-color)
```

the expression **(fake-color)** will only be evaluated if the symbol **color** is not in the features list.

The symbols following the **#+** or **#-** characters can be replaced by logical expressions in features. For example,

```
#+(and windows (not color)) (fake-color)
```

would ensure that **(fake-color)** is only evaluated on a system that has the feature **windows** but does not have the feature **color**.

⁹In XLISP-STAT **require** will first try to load the file by the name given as its first or second argument. If this fails, it searches in the startup folder on the Macintosh or the XLISP-STAT library directory on UNIX systems. Finally, it concatenates the argument onto the end of the string in the global variable ***default-path*** and tries to load a file of the resulting name.

4.5.3 Debugging Tools

Three debugging tools are available in Lisp: the break loop, the trace facility, and the stepper. All Lisp systems compliant with the Common Lisp standard provide at least these three facilities, but the details of how they work and what output they produce differ from system to system. In this section I will describe the versions of these facilities provided in XLISP-STAT.

The Break Loop

A break loop can be entered by executing the `break` function. Placing a `(break)` expression inside a function's code, as in

```
(defun f (x) (break) x)
```

allows you to examine variables in the environment of the function. To continue from a break, execute the `continue` function:

```
> (f 3)
break: **BREAK**
if continued: return from BREAK

1> x
3
1> (continue)
[ continue from break loop ]
3
```

The number in the prompt is the level of the break loop; if another break is caused from within a break loop, this level will be incremented. `break` also accepts an optional format string, followed by any additional arguments required by the format directives; these are used to print a message on entering the break loop.

The system will automatically enter a break loop when an error occurs if the global variable `*breakenable*` is not `nil`. If `*tracenable*` is also not `nil`, a traceback of the calls leading to the break is printed. The number of levels is controlled by the `*tracelimit*` variable. If `*tracenable*` is `nil` you can still get a call stack trace by calling the `baktrace` (*sic*) function.

The functions `debug` and `nodebug` provide a convenient way of switching `*breakenable*` from `nil` to `t` and back.

If a break loop was entered as a result of an error it may not be possible to use `continue`. For noncontinuable errors you can return to the top level by calling the `top-level` function. On the Macintosh version of XLISP-STAT you can also choose the **Top-Level** item in the **Command** menu, or its command key equivalent.

If you are several breaks deep after several errors, you can pop back one at a time using the `clean-up` function.

Here is an example session with breaks enabled:

```

> (defun f (x) (/ x 0))
F
> (debug)
T
> (f 3)
error: illegal zero argument

1> (baktrace)
Function: #<Subr-BAKTRACE: #273d7a>
Function: #<Subr-/: #28c152>
Arguments:
  3
  0
Function: #<Closure-F: #37733e>
Arguments:
  3
NIL
1> (f 3)
error: illegal zero argument

2> (clean-up)
[ back to previous break level ]
1> x
3
1> (top-level)
[ back to top level ]

```

Tracing

To track the use of a particular function without interrupting it with break loops, you can *trace* it. To trace a function *f*, evaluate the expression

```
(trace f)
```

To stop tracing, evaluate

```
(untrace f)
```

Calling *untrace* with no arguments untraces all functions currently being traced. *trace* and *untrace* are macros that don't require their arguments to be quoted.

As an example, let's trace the execution of a recursive *factorial* function defined as

```
(defun factorial (n)
  (if (= n 0) 1 (* (factorial (- n 1)) n)))
```

```

> (trace factorial)
(FACTORIAL)
> (factorial 6)
Entering: FACTORIAL, Argument list: (6)
  Entering: FACTORIAL, Argument list: (5)
    Entering: FACTORIAL, Argument list: (4)
      Entering: FACTORIAL, Argument list: (3)
        Entering: FACTORIAL, Argument list: (2)
          Entering: FACTORIAL, Argument list: (1)
            Entering: FACTORIAL, Argument list: (0)
              Exiting: FACTORIAL, Value: 1
              Exiting: FACTORIAL, Value: 1
              Exiting: FACTORIAL, Value: 2
              Exiting: FACTORIAL, Value: 6
              Exiting: FACTORIAL, Value: 24
              Exiting: FACTORIAL, Value: 120
Exiting: FACTORIAL, Value: 720
720
> (untrace factorial)
NIL
> (factorial 6)
720

```

The Stepper

The stepper allows you to go through the evaluation of an expression one step at a time. You can step through the evaluation of an expression by evaluating

`(step expr)`

At each step the stepper waits for a response. The possible responses are

```

:b - break
:h - help (this message)
:n - next
:s - skip
:e - evaluate

```

In XLISP-STAT on the Macintosh `step` opens a dialog box with buttons for constructing the responses. Here is a simple example of a stepper session:

```

(step '(+ 1 2 (* 3 (/ 2 4))))
Form: (+ 1 2 (* 3 (/ 2 4))) ? :n
Form: 1
Value: 1
Form: 2

```

```

Value: 2
Form: (* 3 (/ 2 4)) ? :n
  Form: 3
  Value: 3
  Form: (/ 2 4) ? :n
    Form: 2
    Value: 2
    Form: 4
    Value: 4
    Value: 0.5
  Value: 1.5
Value: 4.5
4.5

```

4.5.4 Timing

Several macros and functions are available for timing evaluation and determining the current state of the system clock. The macro `time` takes an expression, evaluates it, prints the time required by the evaluation, and returns the result of the expression:

```

> (time (mean (iseq 1 1000)))
The evaluation took 0.97 seconds
500.5

```

The argument to `time` does not need to be quoted.

The functions `get-internal-run-time` and `get-internal-real-time` return the total run time and the total elapsed time from some starting point. The starting point and the measurement units are system dependent. The number of internal time units per second is the value of the global variable `internal-time-units-per-second`.

4.5.5 Defsetf

A number of accessor functions can be used as generalized variables with `setf`. You can define *setf methods* for accessor functions of your own using `defsetf`. `defsetf` can be used in several ways. The simplest form is

```
(defsetf <accessor> (<access-fcn>))
```

The parameter `<access-fcn>` is a symbol naming a function that takes as its arguments the arguments to `<accessor>` followed by a new value for the location specified by the `<accessor>` arguments. The `<access-fcn>` should return the value argument as its value.

As an example, in Section 3.10.2 I introduced a functional abstraction for an expression representing a sum. This abstraction included a constructor function, `make-sum`, and two accessor functions, `addend` and `augend`. To be

able to change the addend in a sum, we might like to be able to use an expression like

```
(setf (addend mysum) 3)
```

Assuming that sums are represented as Lisp expressions, a function for modifying the addend can be defined as

```
(defun set-addend (sum new)
  (setf (select sum 1) new)
  new)
```

and this function can then be installed as the `setf` method for `addend` by

```
(defsetf addend set-addend)
```

Now we can use `setf` to change the addend in a sum:

```
> (setf s (make-sum 1 3))
(+ 1 3)
> (setf (addend s) 2)
2
> s
(+ 2 3)
```

4.5.6 Special Variables

By default, Common Lisp variables are lexically scoped. It is possible to have Lisp treat variables named by a particular symbol as dynamically scoped variables. Such variables are called *special variables*. A number of predefined system constants are special variables.

Special global variables can be set up using `defvar`, `defparameter`, and `defconstant`.¹⁰ `defvar` sets up a global variable and initializes it to `nil` or gives it a value specified with a second optional argument, unless it already has a value. If the variable already has a value, the value is left unchanged and the value expression, if supplied, is not evaluated. `defvar` can also be given a string as a third argument that will be installed as a variable documentation string. This string can be retrieved by the `documentation` function and the functions `help` and `help*`:

```
> (defvar x (list 1 2 3) "A simple data set")
X
```

`defparameter` is like `defvar`, except it requires a value and it always assigns that value to the variable, whether the variable already has a value or not. `defconstant` is like `defparameter`, except it marks the variable as a constant that cannot be changed:

¹⁰These functions do exist in XLISP-STAT but do not at present create special variables; the variables they create are statically scoped.

```
> (defconstant e (exp 1) "Base of the natural logarithm")
E
> (setf e 3)
error: can't assign to a constant
```

The variable `pi` is a constant, `t` is a constant with value `t`, and every keyword (symbol starting with a colon) is a constant with itself as its value.

Chapter 5

Statistical Functions

A number of the statistical functions included in Lisp-Stat have already been introduced in the preceding chapters. Most of these functions deal with numerical data sets, represented as Lisp-Stat compound data items. The first section of this chapter introduces some additional functions for examining compound data and reviews the vectorized arithmetic system. The following sections describe some additional functions for handling data, for calculations involving standard probability distributions, and for numerical linear algebra computations. Many of these functions are patterned after similar functions in the *S* system [6,7]. The chapter concludes with two examples illustrating the use of some of the tools presented.

5.1 Compound Data

5.1.1 Compound Data Properties

Compound data include lists, vectors, arrays, and compound data objects (to be introduced in the next chapter). Items that are not compound are called simple. The predicate `compound-data-p` can be used to recognize compound data items.

The major characteristic of compound data items is that they contain a sequence of items. This sequence can be extracted with the function `compound-data-seq`. For lists and vectors this function simply returns its argument. For arrays it returns a displaced array with the elements in row-major order. Elements of compound data items can themselves be compound. The function `element-seq` works through a compound data item recursively and returns a sequence of its simple elements:

```
> (compound-data-seq '(a (b c)))
(A (B C))
> (element-seq '(a (b c)))
```

(A B C)

If all elements of a compound item are simple, then `element-seq` is equivalent to `compound-data-seq`.

The function `count-elements` returns the number of simple items in a compound data item, the length of the result of `element-seq`:

```
> (count-elements '(a (b c)))
3
```

5.1.2 Vectorized Arithmetic

We have already discussed vectorized arithmetic briefly at several points in the preceding chapters. This subsection discusses some of the issues and conventions related to this topic.

Most statistical operations involve operations on collections of numbers. To simplify writing functions for performing these operations, Lisp-Stat redefines the basic Lisp arithmetic functions `+`, `-`, `*`, `/`, `log`, etc., to take compound data as arguments and return the compound data item obtained by applying these functions to each element.

Lisp provides a number of mapping functions for applying a function elementwise to lists or vectors of items. These functions cannot be used directly for implementing vectorized arithmetic for two reasons. First, in the expression

```
(+ '(1 2 3) 4)
```

the intent is to add the number 4 to each element of the list (1 2 3). Since the argument 4 is not a list, functions like `mapcar` are not directly applicable. We would like a constant to be treated like a constant list or array of the appropriate length or dimensions, without having to construct it explicitly.

For the second reason, consider the expression

```
(+ '(1 2 3) '#(4 5 6))
```

The intent is to add two sequences elementwise and return the result. But one of the sequences is a list and the other is a vector. What should the result be in this case? The `map` function requires that the type of the result sequence be specified explicitly. Since this would be inconvenient, we need to adopt a convention for determining the type of the result when the arguments are of different types. The convention used in Lisp-Stat is to take any structural information needed (type, length, dimensions, etc.) from the first compound data item in the argument list. The other compound arguments must match the shape of the first compound argument. Sequences are considered to be the same shape if they are the same length, regardless of type. Multi-dimensional arrays are the same shape if their dimensions are identical.

These rules for handling constants and determining result structure are used in the definition of the `map-elements` function introduced in Section 3.7.

One more decision has to be made in implementing vectorized arithmetic functions: how is an expression like

```
(+ '(1 (2 3)) 4)
```

to be evaluated? There are two choices. If we only allow one level of vectorization, then this expression should result in an error. On the other hand, we can *recursively vectorize* the addition function, producing the result

```
(5 (6 7))
```

for this expression. It is not clear which choice is best. I have decided to use recursive vectorization.

5.2 Data-Handling Functions

5.2.1 Basic Operations

The functions `repeat`, `iseq`, and `rseq` for generating systematic data have already been introduced in Chapter 2. A related function is `difference`. Applied to a sequence of numbers, it returns the sequence of differences between the elements:

```
> (difference '(1 3 6 10))
(2 3 4)
> (difference #'(1 3 6 10))
#(2 3 4)
```

The function `split-list` takes a list and an integer length n , and splits the list into a list of lists, each of length n :

```
> (split-list '(1 2 3 4 5 6) 3)
((1 2 3) (4 5 6))
```

An error is signaled if the length of the list is not a multiple of n .

The function `cumsum` takes a sequence and returns the sequence of the cumulative sums:

```
> (cumsum '(1 2 3 4))
(1 3 6 10)
> (cumsum #'(1 2 3 4))
#(1 3 6 10)
```

Cumulative sums can also be computed using `accumulate`. This function takes a binary function and a sequence, and returns a sequence of the results of cumulatively applying the binary function to the elements of the sequence. Thus cumulative sums can be computed as

```
> (accumulate #'+ '(1 2 3 4))
(1 3 6 10)
```

and cumulative products as

```
> (accumulate #'* '(1 2 3 4))
(1 2 6 24)
```

Several functions compute simple summaries for compound data items. In addition to functions like `mean` and `standard-deviation` introduced in Chapter 2, these include `count-elements`, `sum`, `prod`, `max`, and `min`. These functions are called *vector reducing* since they reduce compound data items to single numbers.

The functions `min` and `max` can take several arguments, but they always return the overall minimum or maximum over all arguments. The functions `pmin` and `pmax` are vectorized functions that can be used to calculate the parallel minimum or maximum of several arguments. For example,

```
> (pmin '(1 2 3 4 5) '(5 4 3 2 1))
(1 2 3 2 1)
```

Finally, the function `covariance-matrix` takes sequences or matrices, forms them into a list of columns, and returns the sample covariance matrix for the columns. All columns should be of equal length.

5.2.2 Sorting Functions

The function `sort-data` takes a compound data item, and returns a sequence of the sorted elements. The argument is not modified. The elements should all be real numbers or strings. String sorting is case sensitive.

`order` applies `element-seq` to its argument, and returns a list of the indices of the smallest, second smallest, etc., elements of the element sequence. `rank` takes a compound data item, and returns a compound data item of the same shape with its entries replaced by their ranks:

```
> (rank '(14 10 12 11))
(3 0 2 1)
> (rank '#2a((14 10) (12 11)))
#2A((3 0) (2 1))
```

The lowest rank is zero, and ties are broken arbitrarily.

The function `quantile` takes a compound data item x and a number (or sequence of numbers) p , with $0 \leq p \leq 1$, and returns the p -th quantile (or sequence of quantiles) of x :

```
> (def x (uniform-rand 20))
X
> (quantile x .5)
```

```
0.4537602
> (quantile x '(.2 .8))
(0.1661055 0.7847313)
```

The functions `median`, `interquartile-range`, and `fivnum` are defined in terms of `quantile`. `fivnum` returns the five-number summary of the minimum, first, second, and third quartiles and the maximum of its argument.

5.2.3 Interpolation and Smoothing

The function `spline` takes two sequences x and y of equal length, and returns a list of the x and y values of a natural cubic spline interpolation for the points specified by the two sequences. The values in the x argument must be strictly increasing. By default, the interpolation is computed at 30 equally spaced points in the range of the x values. This can be changed using the keyword `:xvals`. If this keyword is supplied with an integer n , then the result uses n equally spaced points instead of 30. Thus the expression

```
(let ((x (rseq 0 pi 10)))
  (spline x (sin x) :xvals 50))
```

returns an interpolation of the sine function using 50 points. If the `:xvals` keyword is supplied with a sequence, then this sequence is used as the x values at which to evaluate the spline. So our 50-point interpolation can also be constructed as

```
(let ((x (rseq 0 pi 10)))
  (spline x (sin x) :xvals (rseq 0 pi 50)))
```

The `lowess` function applies the LOWESS smoothing algorithm [19] to two sequences of x and y values. It returns a list of the x sequence and the smoothed y sequence. So the expressions

```
(setf p (plot-points x y))
(send p :add-lines (lowess x y))
```

construct a scatterplot of the data in variables x and y and add a LOWESS smooth to the plot.

Several parameters of the LOWESS algorithm can be controlled by keyword arguments to the `lowess` function. The fraction of the data used to estimate the fit at each point is specified by the keyword `:f`; the default is 0.25. The number of robust iterations can be given using the keyword `:steps`; the default is 2. Finally, the fit at points close together is determined by linear interpolation. The cutoff can be changed with the keyword `:delta`; the default is 2% of the range of the x values.

The function `kernel-smooth` provides an alternative smoothing function. It takes x and y sequences, and returns a set list of x values and smoothed y values. By default, the x values of the result are 30 equally spaced points

in the range of the input x values. The keyword :**xvals** can be used as for the **spline** function to specify an alternate number of points or an alternate sequence of x values for the result. Four types of kernels are supported, specified by the symbols G for Gaussian, T for triangular, U for uniform, and B for bisquare. The default kernel is the bisquare, but you can specify an alternative using the :**type** keyword. The width of the kernel can be specified with the :**width** keyword. Thus the expression

```
(send p :add-lines (kernel-smooth x y :type 'g))
```

would add a Gaussian kernel smooth to the plot constructed above.

A kernel density estimator is available as well. **kernel-dens** requires a single sequence of x values, and returns a list of x values and estimated density values. The expression

```
(plot-lines (kernel-dens (normal-rand 50)))
```

produces a plot of a kernel density estimate for a sample of 50 standard normal random variables. **kernel-dens** accepts the same keyword arguments as **kernel-smooth**.

5.3 Probability Distributions

Lisp-Stat includes functions for evaluating the density, cumulative distribution function, and quantile function of several continuous univariate distributions, including the normal, Cauchy, beta, gamma, χ^2 , t , and F distributions.

Similar functions are available for computing the probability mass function, cumulative distribution function and quantile function of the binomial and Poisson distributions. The quantile functions for these discrete distributions are defined as left continuous inverses of the cumulative distribution functions. The probability mass functions are only defined for integer arguments. Finally, there is a function for evaluating the cumulative distribution function of the bivariate normal distribution. The available functions are listed in the following table; all of these functions are vectorized.

normal-dens	normal-cdf	normal-quant
cauchy-dens	cauchy-cdf	cauchy-quant
beta-dens	beta-cdf	beta-quant
gamma-dens	gamma-cdf	gamma-quant
chisq-dens	chisq-cdf	chisq-quant
t-dens	t-cdf	t-quant
f-dens	f-cdf	f-quant
binomial-dens	binomial-cdf	binomial-quant
poisson-dens	poisson-cdf	poisson-quant
bivnorm-cdf		

The functions for the normal and Cauchy distributions assume standardized distributions and thus require only one argument, the value at which to calculate the density or the cumulative distribution function, or the proportion, between zero and one, at which to evaluate the quantile function. The functions for the gamma, χ^2 , t and Poisson distributions require an additional parameter: the gamma exponent, the degrees of freedom for the χ^2 and t distributions, and the Poisson mean. The gamma distribution is assumed to have scale parameter equal to one. A few examples are

```
> (chisq-cdf 7.5 8)
0.5162326
> (t-quant .975 3)
3.182418
> (poisson-pmf 3 2.5)
0.213763
```

The functions for the beta, F , and binomial distributions require two additional parameters: the two exponents for the beta distribution, the numerator and denominator degrees of freedom for the F distribution, and the sample size and success probability for the binomial distribution. Again here are a few examples:

```
> (beta-cdf .3 5.2 6.3)
0.148392
> (f-quant .9 3 7)
3.074072
> (binomial-pmf 3 5 .5)
0.3125
```

The function `bivnorm-cdf` assumes that both margins are standard normal and takes three arguments, the x and y arguments for the cumulative distribution function and the correlation coefficient:

```
> (bivnorm-cdf .3 .2 .6)
0.4551159
```

The random number generators for these distributions described in Section 2.4.1 are implemented using the Common Lisp random number generator. The Common Lisp random number interface is intended to be portable across a variety of systems. The particular generator used may vary from one Lisp implementation to another. The seed of the generator can be saved and restored, and a new seed can be generated, usually using the system clock. But it is not possible to specify an arbitrary seed for the generator because an appropriate choice would depend on the generator implementation, the machine word length, etc.

The current value of the state is held in the global variable `*random-state*`. The function `make-random-state` can be used to set and save the

state. It takes an optional argument. If the argument is `nil` or is omitted, then `make-random-state` returns a copy of the current value of `*random-state*`. If the argument is a random state object, a copy of it is returned. Finally, if the argument is the symbol `t`, a new “randomly” initialized state object is produced and returned. New seeds are often produced using the system clock, but the exact mechanism may vary from one Lisp system to another.

A random state object has a printed representation that can be read back in using the `read` function. You can thus save the current state to a file and use it to repeat a simulation with the same stream of pseudorandom numbers. For example, after writing the current state to a file using the expression

```
(with-open-file (s "currentstate" :direction :output)
  (print *random-state* s))
```

an expression like

```
(with-open-file (s "currentstate")
  (setf *random-state* (read s))
  (run-simulation))
```

would repeat a simulation with the same set of random numbers each time it is evaluated.

5.4 Array and Linear Algebra Functions

5.4.1 Basic Matrix and Array Functions

Matrices are two-dimensional arrays. They play a major role in statistical computations, so Lisp-Stat provides several additional functions to aid in manipulating matrices. The predicate `matrixp` returns `t` for a matrix and `nil` otherwise.

It is sometimes useful to be able to paste together a set of lists or vectors into the rows or columns of a matrix. This can be done using `bind-rows` or `bind-columns`:

```
> (bind-rows '#(1 2 3) '(4 5 6))
#2A((1 2 3) (4 5 6))
> (bind-columns '#(1 2 3) '(4 5 6))
#2A((1 4) (2 5) (3 6))
```

The arguments to `bind-rows` and `bind-columns` may also be matrices:

```
> (bind-rows '#2a((1 2 3)(4 5 6)) '(7 8 9))
#2A((1 2 3) (4 5 6) (7 8 9))
```

This can be exploited to write a function that borders a matrix with two vectors and a constant:

```
(defun border-matrix (a b c d)
  (bind-rows (bind-columns a b)
    (concatenate 'list c (list d))))
```

Then, for example,

```
> (border-matrix '#2a((1 2)(3 4)) '(5 6) '(7 8) 9)
#2A((1 2 5) (3 4 6) (7 8 9))
```

To be able to compute column means, for example, you need to be able to break a matrix up into its columns. `column-list` does this:

```
> (column-list '#2a((1 2 3)(4 5 6)))
(#(1 4) #(2 5) #(3 6))
```

A `column-means` function can now be defined using `mapcar`:

```
(defun column-means (x)
  (mapcar #'mean (column-list x)))
```

The function `row-list` works analogously.

Four other functions are worth mentioning. `transpose` takes a matrix argument and returns the transpose of its argument. `transpose` can also be used on a list of lists of equal length, representing the rows of a matrix. It then returns the list of lists representing the rows of the transpose. The function `identity-matrix` takes an integer argument k and returns a k by k identity matrix. The function `diagonal` can be used in two ways. Given a matrix argument, it returns a list of the diagonal entries of the matrix (the matrix need not be square):

```
> (diagonal '#2a((1 2)(3 4)))
(1 4)
```

Given a sequence argument, `diagonal` returns a square matrix with the elements of the sequence along the diagonal and zeros off the diagonal:

```
> (diagonal '(a b c))
#2A((A 0 0) (0 B 0) (0 0 C))
```

Finally, `print-matrix` can be used to print a nicer representation of a matrix than the printer usually produces:

```
> (print-matrix #2a((1 2 3)(4 5 6)(7 8 9)))
#2a(
  (1 2 3)
  (4 5 6)
  (7 8 9)
)
NIL
```

Like `print`, the function `print-matrix` can be given an optional stream argument.

Lisp-Stat also provides a function `permute-array` that takes an array and a list, and permutes the array according to the permutation specified in the list:

```
> (permute-array '#2a((1 2 3)(4 5 6)) '(1 0))
#2A((1 4) (2 5) (3 6))
> (permute-array '#3a(((1 2 3)(4 5 6))((7 8 9)(10 11 12)))
               '(2 0 1))
#3A(((1 4) (7 10)) ((2 5) (8 11)) ((3 6) (9 12)))
```

5.4.2 Matrix Multiplication

The function `matmult` takes two matrices and multiplies them. The elements of the matrices must be numbers.

```
> (matmult '#2a((1 2 3)(4 5 6)) '#2a((1 2)(3 4)(5 6)))
#2A((22 28) (49 64))
```

One of the arguments can be a sequence – a list or a vector. If this is the case, the sequence is treated as a row vector if it is the first argument and a column vector if it is the second. The result is a sequence of the same type as the sequence argument:

```
> (matmult '#(1 2) '#2a((1 2 3)(4 5 6)))
 #(9 12 15)
> (matmult '#2a((1 2 3)(4 5 6)) '(1 2 3))
(14 32)
```

If both arguments to `matmult` are sequences of the same length, a number is returned – the inner product. The inner product can also be computed with the `inner-product` function:

```
> (matmult '(1 2 3) '#(4 5 6))
32
> (inner-product '(1 2 3) '#(4 5 6))
32
```

`matmult` can be called with more than two arguments. In this case, the first two arguments are multiplied together, the result is multiplied on the right by the third argument, and so on.

The function `outer-product` takes two sequences and returns their outer product:

```
> (outer-product '#(1 2) '(3 4 5))
#2A((3 4 5) (6 8 10))
```

outer-product also provides a generalized outer product. It takes a function of two arguments as an optional argument. If this function is supplied, the (i, j) element of the result is obtained by applying the function to element i of the first sequence and element j of the second sequence:

```
> (outer-product #'(1 2) '(3 4 5) #''+)
#2A((4 5 6) (5 6 7))
```

The function **cross-product** takes a single argument **x** and returns
`(matmult (transpose x) x).`

5.4.3 Linear Algebra Functions

Lisp-Stat provides a number of functions for linear algebra computations.¹ A function used by several other functions is **lu-decomp**. **lu-decomp** takes a square matrix of numbers, performs an *LU* decomposition with partial pivoting, and returns a list of four elements. The first element is a square matrix containing the *LU* decomposition of a row-wise permutation of the matrix determined by the pivoting process. The lower triangle, excluding the diagonal, is the *L* part; the diagonal elements of *L* are all equal to one. The upper triangle of the result matrix, including the diagonal, is the *U* part. The second element of the result list is a vector of integers describing the permutation used in the pivoting process. The third element is a number that is equal to ± 1 , positive if the number of row interchanges used is even, negative if it is odd. The last element of the result matrix is a flag that is **t** if the input matrix is numerically singular, **nil** if it is nonsingular:

```
> (lu-decomp '#2a((1 2)(3 4)))
(#2A((3 4) (0.3333333 0.6666667)) #(1 1) -1 NIL)
```

lu-solve takes two arguments. The first is the result returned by **lu-decomp** for a matrix *A*. The second argument is a list or vector that is taken as the right-hand side *b* of the equation $Ax = b$. **lu-solve** solves the equation using back-solving with the *LU* decomposition provided, and returns the solution:

```
> (lu-solve (lu-decomp '#2a((1 2)(3 4))) '(5 6))
(-4 4.5)
> (matmult '#2a((1 2)(3 4)) '(-4 4.5))
(5 6)
```

An error is signaled if the matrix is found to be numerically singular during the back-solving process.

¹In XLISP-STAT most of the algorithms used for these computations are taken from Press, Flannery, Teukolsky, and Vetterling [50].

The functions `determinant`, `inverse`, and `solve` use the *LU* decomposition to process their matrix argument. `determinant` takes a matrix and returns its determinant. `inverse` returns the inverse of its argument, or signals an error if the matrix is numerically singular:

```
> (inverse '#2a((1 2)(3 4)))
#2A((-2 1) (1.5 -0.5))
> (matmult '#2a((1 2)(3 4)) '#2A((-2 1) (1.5 -0.5)))
#2A((1 0) (0 1))
```

`solve` takes a matrix *A* and a second argument *b*, and returns the solution to $Ax = b$. *b* can be a sequence or a matrix:

```
> (solve '#2a((1 2)(3 4)) '(5 6))
(-4 4.5)
> (solve '#2a((1 2)(3 4)) (identity-matrix 2))
#2A((-2 1) (1.5 -0.5))
```

Arguments to `lu-decomp`, `lu-solve`, `determinant`, `inverse`, and `solve` may contain real numbers or complex numbers.²

`chol-decomp` takes a square symmetric matrix *A*, and computes the Cholesky decomposition of $A + D$, with *D* a nonnegative diagonal matrix that is added to *A* if necessary to ensure that $A + D$ is numerically strictly positive definite. The result returned by `chol-decomp` is a list of the lower triangular matrix *L* that satisfies $LL^T = A + D$ and the maximum of the elements of *D*. If *A* is strictly positive definite, this second element of the result is zero:

```
> (setf x (chol-decomp '#2a((2 1)(1 2))))
(#2A((1.414214 0) (0.7071068 1.224745)) 0)
> (matmult (first x) (transpose (first x)))
#2A((2 1) (1 2))
```

`qr-decomp` takes an *m* by *n* matrix *A*, with $m \geq n$, and computes a *QR* decomposition. The result returned is a list consisting of an *m* by *n* matrix *Q* with orthonormal columns and an upper triangular matrix *R* such that $QR = A$:

```
> (setf x (qr-decomp '#2a((1 2)(3 4)(5 6))))
(#2A((-0.1690309 0.8970852) (-0.5070926 0.2760262)
(-0.8451543 -0.3450328)) #2A((-5.91608 -7.437357)
(0 0.8280787)))
> (matmult (first x) (second x))
#2A((1 2) (3 4) (5 6))
```

²At the time of writing, in XLISP-STAT the remaining functions described in this section only take real arguments.

We can use **print-matrix** to make the components of the result more readable:

```
> (print-matrix (first x))
#2a(
  (-0.1690309  0.8970852)
  (-0.5070926  0.2760262)
  (-0.8451543 -0.3450328)
)
NIL
> (print-matrix (second x))
#2a(
  ( -5.91608 -7.437357)
  (          0 0.8280787)
)
NIL
```

If **qr-decomp** is given the :pivot keyword with value **t**, it permutes the columns of the matrix to ensure the absolute values of the diagonal entries of the matrix R are nonincreasing. In this case, the result includes a third element, a list of the indices of the columns in the order in which they were used.

sv-decomp takes an m by n matrix A , with $m \geq n$, and computes its singular value decomposition. It returns a list of the form $(U W V FLAG)$, where U and V are matrices whose columns are the left and right singular vectors, and W is a vector of the singular values of A in decreasing order. **FLAG** is **t** if the algorithm converged and is **nil** otherwise. Thus if **FLAG** is **t** and D is the diagonal matrix with W on the diagonal and zeros off the diagonal, then $UDV^T = A$:

```
> (setf x (sv-decomp '#2a((1 2)(3 4)(5 6))))
(#2A((0.229848 0.883461) (0.524745 0.240782)
(0.819642 -0.401896)) #(9.52552 0.514301)
#2A((0.619629 -0.784894) (0.784894 0.619629)) T)
> (matmult (first x)
            (diagonal (second x))
            (transpose (third x)))
#2A((1 2) (3 4) (5 6))
```

The functions **eigenvalues** and **eigenvectors** return a vector of the eigenvalues, in decreasing order, and a list of the corresponding eigenvectors, respectively, of a square symmetric matrix. **eigen** returns a list whose two elements are a vector of eigenvalues and a list of eigenvectors of a square symmetric matrix. The function **backsolve** takes an upper triangular matrix A and a sequence b , and returns the solution to $Ax = b$. The function **rcondest** returns an estimate of the reciprocal of the condition number of an upper triangular matrix.

A final function that is particularly useful for constructing certain dynamic graphs is `make-rotation`. This function takes two sequences of equal length and an optional real number, representing an angle in radians, as its arguments. It returns the rotation matrix that rotates in the plane defined by the two sequences, keeping the orthogonal complement fixed, from the first sequence toward the second by the specified angle. For example,

```
(make-rotation '(1 0 0 0) '(1 1 1 1) (/ pi 10))
```

returns a matrix that rotates in the plane of the first coordinate axis and the diagonal in four-dimensional space by an angle of $\pi/10$. If `make-rotation` is not given an angle, it uses the smaller angle between the two sequences.

The Lisp-Stat linear algebra functions internally use floating point numbers of type `double-float` or `long-float`, depending on the Lisp system and the hardware. The global variable `machine-epsilon` contains the smallest floating point number `x` of the appropriate type for which

```
(not (= 1 (+ 1 x)))
```

returns true.

5.5 Regression Computation Functions

Regression computations can be carried out using the sweep operator as described in Weisberg [65, Problem 2.7]. Sweeping is performed on rows. That is, sweeping a matrix A on pivot k transforms it to a matrix B with elements given by

$$\begin{aligned} b_{kk} &= \frac{1}{a_{kk}} \\ b_{ik} &= \frac{a_{ik}}{a_{kk}} \quad i \neq k \\ b_{kj} &= -\frac{a_{kj}}{a_{kk}} \quad j \neq k \\ b_{ij} &= a_{ij} - \frac{a_{ik}a_{kj}}{a_{kk}} \quad i \neq k, j \neq k \end{aligned}$$

This definition of the sweep operator is commutative and self-inverting: the order in which two pivots are swept does not matter, and sweeping on the same pivot twice is equivalent to not sweeping on that pivot at all.

The function `make-sweep-matrix` takes a matrix `x`, a sequence `y`, and an optional sequence `w` of weights, and returns the (possibly weighted) corrected cross product matrix that has been swept on the constant row. That is, if A is the matrix consisting of a column of ones, the columns of `x`, and the column `y`, and W is the diagonal matrix with the elements of `w` on its diagonal, then `make-sweep-matrix` computes the cross product matrix $A^T W A$ and sweeps it on the constant row.

The function **sweep-operator** takes a matrix, a sequence of row indices to sweep, and an optional sequence of tolerances as arguments. It returns a list consisting of a copy of the matrix swept on (some of) the specified rows and a list of the indices of the rows actually swept. A row is only swept if the absolute value of its pivot is larger than the specified tolerance. The default tolerance is .000001.

If a cross product matrix has been swept on a set of pivots, then the corresponding elements of the bottom row of the swept matrix are the least squares coefficients for a model consisting of the variables indexed by the swept pivots. Thus a function that takes an X matrix, excluding the constant term, a y vector, and a sequence of weights and calculates the least squares coefficients can be defined as

```
(defun reg-coefs (x y weights)
  (flet ((as-list (x) (coerce (compound-data-seq x) 'list)))
    (let* ((m (make-sweep-matrix x y weights))
           (p (array-dimension x 1)))
      (as-list (select (first (sweep-operator m (iseq 1 p)))
                      (+ p 1)
                      (iseq 0 p))))))
```

The result returned by the **select** expression is a 1 by $p + 1$ matrix that is first converted to a vector using **compound-data-seq** and then to a list using **coerce**. This definition assumes that the model includes an intercept. For a model without an intercept, we would need to first sweep on index 0 to remove the constant term, and only select the entries with indices $1, \dots, p$ for the result.

5.6 Some Examples

This section gives two additional examples that make use of some of the tools introduced in this chapter. The first example is a slightly more elaborate version of the **make-projection** example used earlier, and the second is a function for computing robust regression coefficients for a variety of weight functions.

5.6.1 Constructing a Projection Operator

In Chapter 3 we saw several variations on a function that modeled a projection operator using a function closure. At the time we could only easily handle projections onto a one-dimensional space, but now we have several tools for handling higher-dimensional projections. One approach is to take a list of lists or vectors spanning a space, use them as the columns of a matrix X , and form the singular value decomposition UDV^T of X . The columns of U corresponding to positive singular values then form an orthonormal basis

for the column space of X . Thus if U_1 is the matrix of these columns, then the projection P_y of a vector y onto the column space of X can be written as

$$P_y = U_1 U_1^T y$$

In defining a new version of `make-projection`, we can allow the columns to be specified either as a list of vectors or as a matrix by using the expression

```
(if (matrixp cols) cols (apply #'bind-columns cols))
```

to convert to a matrix if necessary.

After computing the matrix U as `u` and the singular values as `s-vals`, we can compute a list of the columns for the nonzero singular values as

```
(select (column-list u) (which (> s-vals 0.0)))
```

These columns can then be formed into the matrix U_1 by using the function `bind-columns`.

In computing the final projection, we can avoid the need for transposing `u1` by multiplying it on the left by the list or vector `y` and using the result, which will be a list or vector as well, on the right of `u1`, as in the expression

```
(matmult u1 (matmult y u1))
```

Combining these steps, we have the following definition:

```
(defun make-projection (cols)
  (let* ((x (if (matrixp cols)
                 cols
                 (apply #'bind-columns cols)))
         (svd (sv-decomp x))
         (u (first svd))
         (s-vals (second svd))
         (basis (select (column-list u)
                        (which (> s-vals 0.0))))
         (u1 (apply #'bind-columns basis)))
    #'(lambda (y) (matmult u1 (matmult y u1))))
```

Exercise 5.1

Testing for zero singular values only allows for exact collinearity of some of the columns given as the arguments. It would be better to choose an appropriate tolerance, and include only those columns of U in the basis that correspond to singular values exceeding this tolerance. Modify the `make-projection` function to use such a tolerance. You should choose a default value, but allow it to be changed with a keyword argument.

5.6.2 Robust Regression

One approach to computing robust regression estimates for a linear model of the form $y = X\beta$ is to use an iteratively reweighted least squares algorithm in which weights are computed using a robust weight function w that assigns small weights to observations with large standardized residuals [21]. Specifically, given a current set of weights, we

- compute the weighted least squares estimate $\hat{\beta}$
- compute the residuals r
- compute a scale estimate \hat{s} as the median absolute residual divided by .6745
- calculate a new set of weights as $w(r/\hat{s})$

and this is repeated until a convergence criterion is satisfied. The scale estimate includes the division by .6745 to ensure that it estimates the population standard deviation for normally distributed data.

A number of different weight functions have been proposed in the literature, including the biweight function

$$w(u) = \begin{cases} (1 - (u/K)^2)^2 & |u| \leq K \\ 0 & |u| > K \end{cases}$$

the Cauchy weight function

$$w(u) = \frac{1}{1 + (u/K)^2}$$

and the Huber weight function

$$w(u) = \begin{cases} 1 & |u| \leq K \\ K/|u| & |u| > K \end{cases}$$

All three depend on a tuning constant K . Recommended default choices for K are 4.685 for the biweight, 2.385 for the Cauchy, and 1.345 for the Huber weight functions.

In implementing a robust regression function, let's begin by assuming the model contains an intercept along with the columns in a specified X matrix. The weighted least squares estimates for a given set of weights can then be computed as in the function `reg-coefs` defined in Section 5.5. If we obtain the coefficients as a list `beta`, we can compute the fitted values as

```
(+ (first beta) (matmult x (rest beta))))
```

The iteration can be controlled using a do loop like

```
(do ((last nil beta)
     (count 0 (+ count 1))
     (beta (reg-coefs weights) (improve-guess beta)))
    ((good-enough-p last beta count) beta)))
```

The iteration variable `last` is used to hold the previous value of the coefficients for comparison to the current guess; the initial value of `nil` is used to indicate that no previous value is available. The iteration count is monitored as well.

If the variable `wf` refers to the weight function, then the body of `improve-guess` can be written as

```
(let* ((resids (- y (fitvals beta)))
      (scale (/ (median (abs resids)) .6745))
      (wts (funcall wf (/ resids scale))))
      (reg-coefs wts))
```

Finally, `good-enough-p` should print a message if the iteration count has been exceeded, and check whether the change in the parameter estimates is sufficiently small. The test expression can be written as

```
(flet ((rel-err (x y) (mean (/ (abs (- x y)) (+ 1 (abs x)))))))
  (or (> count count-limit)
      (and last (< (rel-err beta last) tol))))
```

The `and` expression first checks that `last` is not `nil`, and then compares the average relative change in the coefficients to a tolerance `tol`. The addition of 1 in the change computation ensures that the change is measured as a relative error for large coefficients and an absolute error for small coefficients. Other choices are obviously possible.

We can now combine these elements into a single function, using a `labels` expression to define the functions for implementing the individual steps. Keyword arguments are used to specify initial weights, a tolerance for the convergence criterion, and a bound on the number of iterations:

```
(defun robust-coefs (x y wf &key
                      (weights (repeat 1 (length y)))
                      (tol .0001)
                      (count-limit 20))
  (let ((x (if (matrixp x) x (apply #'bind-columns x))))
    (labels ((as-list (x) (coerce (compound-data-seq x) 'list))
             (rel-err (x y)
              (mean (/ (abs (- x y)) (+ 1 (abs x))))))
             (reg-coefs (weights)
              (let* ((m (make-sweep-matrix x y weights))
                     (p (array-dimension x 1)))
                (as-list
```

```

(select (first (sweep-operator m (iseq i p)))
        (1+ p)
        (iseq 0 p)))))

(fitvals (beta)
  (+ (first beta) (matmult x (rest beta)))))

(improve-guess (beta)
  (let* ((resids (- y (fitvals beta)))
         (scale (/ (median (abs resids)) .6745))
         (wts (funcall wf (/ resids scale))))
    (reg-coefs wts)))

(good-enough-p (last beta count)
  (if (> count count-limit)
      (format t "Iteration limit exceeded~%")
      (or (> count count-limit)
          (and last (< (rel-err beta last) tol))))))

(do ((last nil beta)
      (count 0 (+ count 1))
      (beta (reg-coefs weights) (improve-guess beta)))
  ((good-enough-p last beta count) beta)))

```

To use this function, you need to specify a weight function. To make it easy to use one of the standard weight functions listed above, we can use the function defined as

```

(defun make-wf (name &optional
  (k (case name
    (biweight 4.685)
    (cauchy 2.385)
    (huber 1.345)))))

#'(lambda (r)
  (let ((u (abs (/ r k))))
    (case name
      (biweight (^ (- 1 (^ (pmin u 1) 2)) 2))
      (cauchy (/ 1 (+ 1 (^ u 2)))))
      (huber (/ 1 (pmax u 1)))))))

```

As an illustration we can try these functions on an example, the stack loss data from Brownlee [15, Section 13.12]. This data set, shown in Table 5.1, consists of 21 observations on the operation of a plant for the oxidation of ammonia to nitric acid. The three independent variables are *AIR*, the airflow through the plant, representing its rate of operation, *TEMP*, the temperature of cooling water circulated through coils in the absorption tower, and *CONC*, a linear coding of the concentration of acid circulating. The dependent variable is *LOSS*, 10 times the percentage of ammonia fed to the plant that escapes from the absorption tower unabsorbed.

The least squares fit can be computed as

AIR	TEMP	CONC	LOSS
80	27	89	42
80	27	88	37
75	25	90	37
62	24	87	28
62	22	87	18
62	23	87	18
62	24	93	19
62	24	93	20
58	23	87	15
58	18	80	14
58	18	89	14
58	17	88	13
58	18	82	11
58	19	93	12
50	18	89	8
50	18	86	7
50	19	72	8
50	19	79	8
50	20	80	9
56	20	82	15
70	20	91	15

Table 5.1: Stack loss data.

```
> (regression-model (list air temp conc) loss)
```

Least Squares Estimates:

Constant	-39.91967	(11.896)
Variable 0	0.7156402	(0.1348582)
Variable 1	1.295286	(0.3680243)
Variable 2	-0.1521225	(0.156294)

R Squared: 0.9135769

Sigma hat: 3.243364

Number of cases: 21

Degrees of freedom: 17

The coefficients of the robust regression using Huber's weight function are

```
> (robust-coefs (list air temp conc) loss (make-wf 'huber))
(-41.0267 0.829344 0.926232 -0.127856)
```

Having obtained these coefficients, we could go back and calculate the weights used in the final fit, and check whether there were any points that received particularly low weights. In this case the indices of points receiving weights of less than .5 are 0, 2, 3, and 20, points often identified as possible outliers in this data set.

Exercise 5.2

The **robust-coefs** function assumes that an intercept is to be included in the fit. Modify the definition to allow a keyword to be used to specify whether an intercept is to be included.

Exercise 5.3

The definition of **reg-coefs** used here ignores the possibility that the X matrix may be numerically rank deficient. **sweep-operator** does check for this possibility and includes a list of indices for the columns used in the fit in its result. Modify **reg-coefs** to allow for degeneracy in the X matrix. You may have to change the result returned to include information about the columns used in the fit.

Chapter 6

Object-Oriented Programming

Lisp-Stat supports object-oriented programming by providing an *object* data type and several functions for dealing with objects. In Chapter 2 we saw that Lisp-Stat plots and models are implemented as objects, and we used several messages to examine and modify these objects. The present chapter describes the Lisp-Stat object system in more detail. This system is designed specifically for use in interactive statistical work, and differs somewhat from other object systems used in the Lisp community. Other object systems are discussed briefly in Section 6.7.

6.1 Some Motivation

Suppose we would like to build an intelligent function for describing or plotting a data set. Initially, we might consider three types of data: single samples, multiple samples, and paired samples. A `describe-data` function could then be written as

```
(defun describe-data (data)
  (cond
    ((single-sample-p data) ...)
    ((multiple-sample-p data) ...)
    ((paired-sample-p data) ...)
    (t (error "don't know how to describe this data set"))))
```

You could define several other functions similarly, for example, a `plot-data` function.

Suppose we later decide that we would also like to be able to handle a fourth type of data set, a simple time series. To accommodate this new data type, we would have to edit the `describe-data` and `plot-data` functions.

This has a number of drawbacks. First, we need to have access to, and be able to understand, the original code for these functions. This is not a problem in this simple example, but it could create major difficulties in more complicated settings. Second, by editing the code with the intent of merely adding the ability to handle the new data type, we run the risk of breaking code for the existing data types.

An alternative strategy is to arrange for the data sets themselves to “know” how to describe themselves, in other words, to be able to access appropriate pieces of code for printing a description of themselves. The **describe-data** function would then simply ask a data set to find the appropriate piece of code and execute it. Adding a new type of data set would not require any modification of the **describe-data** function, or of code for handling existing data types; we would only have to write and install the appropriate code for the new data type. This is one of the main ideas behind object-oriented programming. The process of requesting that the appropriate piece of code be located and executed is called *message passing*; the piece of code itself is the *method* invoked by the message.

Another key notion in object-oriented programming is *inheritance*. A new kind of object may only differ in a few details from an existing object. For example, a time series may need its own description method, but might be able to use the same plotting method as a paired sample. To take advantage of this similarity, we can arrange objects in a hierarchy, with each object inheriting from other objects, its *ancestors*. The system for handling messages can then be instructed to use the method of an ancestor if an object does not have a method of its own. Then, as we develop new objects, we only need to provide a new method if no inherited method is available, or if the inherited method is not suitable. This ability to reuse code already developed can reduce the time needed to design new objects by an order of magnitude. It also facilitates code maintenance by ensuring that improvements to methods used by many objects are passed on automatically.

6.2 Objects and Messages

6.2.1 Basic Structure and Terminology

Lisp-Stat objects are a separate data type designed specifically for object-oriented programming. You can determine whether an item is an object by using the **objectp** predicate.

An object is a data structure that contains information specific to the object in named locations called *slots*. In this respect objects are similar to C or S structures. In addition to holding data, objects can respond to messages asking them to take certain actions. A message is sent to an object using the function **send** in an expression of the form

```
(send (object) (selector) (arg 1) ... (arg n))
```

The *(selector)* is a symbol, typically a keyword symbol, used to identify the message. As an example, suppose the variable *p* refers to a histogram object returned by the function *histogram* introduced in Chapter 2. Then the expression

```
(send p :add-points (normal-rand 20))
```

sends the histogram a message asking it to add a sample of 20 standard normal random variables to itself. The *message selector* for the message is *:add-points*; the *message argument* is the sample of normal variables generated by the expression *(normal-rand 20)*. The message consists of the selector and the arguments. The code used to respond to this message is the *method* for the message. The *send* function finds the appropriate method by *dispatching* on the object and the message selector.

Objects are organized in an *inheritance hierarchy*. At the top is the *root object*, the value of the global variable **object**. This object contains methods for a number of standard messages that apply to all objects. For example, the message *:own-slots* returns a list of the symbols naming the slots contained in an object. The slots in the root object are

```
> (send *object* :own-slots)
(DOCUMENTATION PROTO-NAME INSTANCE-SLOTS)
```

The message *:slot-value* takes one argument, a symbol naming a slot, and returns the value currently stored in the slot:

```
> (send *object* :slot-value 'proto-name)
*OBJECT*
> (send *object* :slot-value 'instance-slots)
NIL
```

The documentation slot is used to hold information for use by the *:help* message.

The root object is a *prototype* object, designed to serve as a template for constructing other objects, called *instances* of the prototype. The slot *proto-name* contains a symbol naming the prototype. The value of this slot is used in producing the printed representation of an object:

```
> *object*
#<Object: 820716, prototype = *OBJECT*>
```

Prototype objects are like any other objects, except that they contain some additional information needed to construct instances. In particular, to be used as a prototype an object should contain a slot named *instance-slots*. The value of this slot is a list of symbols, representing slots to be installed in instances of the prototype. The prototype should also contain a slot corresponding to each symbol in this list. Instances of a prototype differ from the prototype, and from one another, in the values contained in these slots.

When an object is sent a message, it checks first whether it has a method of its own for handling the message. If it does, it uses that method. Otherwise, it works its way through its ancestors until it finds a method. If no method is found, an error is signaled. The same procedure is used in determining a slot value: first the object checks its own slots, and then the slots of its ancestors. The order in which the ancestors are checked is called the object's *precedence list*. The precedence list always contains the object itself as the first element and the root object as the last element. In the simple cases considered in this section every object has only a single direct ancestor, or *parent*. In this case the precedence list consists of the object followed by its parent, the parent of its parent, and so on, up to the root object. More complicated cases will be discussed below in Section 6.6.

6.2.2 Constructing New Objects

New objects are usually constructed by sending a prototype object the :new message. The number of arguments required by this message depends on the prototype; for the root object it requires no arguments. As an example, we can construct an object to represent a data set with the expression

```
(setf x (send *object* :new))
```

Initially, an object created from the *object* prototype has no slots of its own:

```
> (send x :own-slots)
NIL
```

You can add slots to an object with the :add-slot message. This message requires one argument, a symbol naming the slot. You can specify the value to place in the slot with an additional optional argument. If no value is supplied, the value of the slot defaults to nil. Let's add a data slot containing some normal random numbers, and a title slot for holding a descriptive title to our data set:

```
> (send x :add-slot 'data (normal-rand 50))
(1.39981 -0.0601746 ...)
> (send x :add-slot 'title)
NIL
> (send x :own-slots)
(TITLE DATA)
```

The message :slot-value can be used to examine the values of an object's slots:

```
> (send x :slot-value 'data)
(1.39981 -0.0601746 ...)
> (send x :slot-value 'title)
NIL
```

The `:slot-value` message can also be given a second argument, to be inserted as the new value of the slot. We can use this approach to place a title in the `title` slot of our data set:

```
> (send x :slot-value 'title "a data set")
  "a data set"
> (send x :slot-value 'title)
  "a data set"
```

It is also possible to delete a slot from an object by sending it the message `:delete-slot` with one argument, the symbol naming the slot to be deleted.

In addition to its own slots, the object `x` also has access to the slots in the root object as inherited, or *shared*, slots:

```
> (send x :slot-value 'proto-name)
*OBJECT*
```

The message `:slot-names` returns a list of all slots accessible to an object, including both slots owned by the object and inherited slots:

```
> (send x :slot-names)
(TITLE DATA DOCUMENTATION PROTO-NAME INSTANCE-SLOTS)
```

The message `:has-slot` can be used to determine if an object has access to a slot of a particular name:

```
> (send x :has-slot 'title)
T
> (send x :has-slot 'x)
NIL
> (send x :has-slot 'proto-name)
T
```

By default, the method for this message checks the inherited slots in addition to the object's own slots. If the keyword `:own` is supplied with value `t`, then only the object's own slots are checked:

```
> (send x :has-slot 'title :own t)
T
> (send x :has-slot 'proto-name :own t)
NIL
```

Whether a slot is owned by an object or inherited from another object is only important if you want to delete the slot or change the slot's value. You can only delete a slot or change the value of a slot by sending the appropriate message to the slot's owner. Asking an object to change the value of an inherited slot results in an error. For example,

```
> (send x :slot-value 'proto-name 'new-name)
error: object does not own slot - PROTO-NAME
```

Exercise 6.1

Construct an object to represent a paired sample.

6.2.3 Defining New Methods

Initially, an object can respond to a number of standard messages like the ones we just used. The methods for these messages are inherited from the root object. As already outlined briefly in Section 2.7, you can define new methods for an object by using the macro `defmeth`. A `defmeth` expression looks like

```
(defmeth <object> <selector> <parameters> <body>)
```

The argument `<object>` should be an expression that evaluates to an object. The remaining arguments are not evaluated. `<parameters>` is a list of symbols naming the parameters of the method. This parameter list can include `&key`, `&optional`, and `&rest` arguments. If the body of the method starts with a string, that string is taken to be a documentation string and is installed for use by the `:help` message.

In writing a method for a message, you usually need to be able to refer to the object receiving the message. Since methods can be inherited, you cannot be certain of the identity of the receiving object at the time a method is written. To deal with this problem, the object system ensures that the variable `self` is bound to the receiving object when the body of a method is evaluated. As an example, we might give our data set a method for the `:describe` message by using the definition

```
(defmeth x :describe (&optional (stream t))
  (format stream "This is ~a~%"
         (send self :slot-value 'title))
  (format stream "The sample mean is ~g~%"
         (mean (send self :slot-value 'data))))
  (format stream "The sample standard deviation is ~g~%"
         (standard-deviation
          (send self :slot-value 'data))))
```

The variable `self` is used to extract the object's data and title. The new `:describe` message can now be used like any other message:

```
> (send x :describe)
This is a data set
The sample mean is 0.127521
The sample standard deviation is 1.0005
```

By default, this method prints to the standard output. The optional `stream` argument can be used to specify an alternative output stream.

The variable `self` can be viewed as an implicit first argument to a method. It can be used in default expressions for keyword and optional arguments. If a function closure is created within a method, the variable `self`, bound to the receiving object, is included in the environment of the closure.

There are several functions that can only be used within the body of a method. One such function is `slot-value`. This function takes one argument, a symbol naming a slot in the current object, and returns the value of this slot. Within a method body you can also use `slot-value` as a place form with `setf`. Using the `slot-value` function instead of the `:slot-value` message typically produces faster code; the `:slot-value` message is intended primarily for examining objects from the interpreter, where the `slot-value` function is not available. Using this function, we can define our `:describe` method as

```
(defmeth x :describe (&optional (stream t))
  (format stream "This is ~a~%" (slot-value 'title))
  (format stream "The sample mean is ~g~%"
    (mean (slot-value 'data)))
  (format stream "The sample standard deviation is ~g~%" 
    (standard-deviation (slot-value 'data))))
```

A drawback to this definition is that it ties us to one particular implementation of our data set by assuming explicitly that the data and title are stored in slots by these names. A more flexible approach, motivated by the principle of data abstraction, is to define accessor methods for obtaining and modifying the title and data as

```
(defmeth x :title (&optional (title nil set))
  (if set (setf (slot-value 'title) title))
  (slot-value 'title))
```

and

```
(defmeth x :data (&optional (data nil set))
  (if set (setf (slot-value 'data) data))
  (slot-value 'data))
```

We can then rewrite the `:describe` method using these accessors as

```
(defmeth x :describe (&optional (stream t))
  (let ((title (send self :title))
        (data (send self :data)))
    (format stream "This is ~a~%" title)
    (format stream "The sample mean is ~g~%" (mean data))
    (format stream "The sample standard deviation is ~g~%" 
      (standard-deviation data))))
```

If we later decide to modify the `:title` method, perhaps to provide a useful default title if the value of the slot is `nil` or to determine the title without using a slot of that name, we will not need to modify the `:describe` method, nor any other method that obtains the title for the data set by using the accessor message `:title`.

The message `:own-methods` returns a list of the messages for which an object has methods of its own:

```
> (send x :own-methods)
(:DESCRIBE :DATA :TITLE)
```

The `:method-selectors` message lists all messages for which methods are available in the object or its ancestors:

```
> (send x :method-selectors)
(:DESCRIBE :DATA :TITLE :METHOD-SELECTORS :SLOT-NAMES ...)
```

The message `:has-method` is analogous to the `:has-slot` message described above.

You can delete a method from an object by using the `:delete-method` message with one argument, the message selector symbol for the method to be deleted.

Exercise 6.2

Define a `:plot` method for the data set `x`.

Exercise 6.3

Modify the `:data` and `:title` methods to check for errors before storing new information in the `title` and `data` slots.

Exercise 6.4

The function `send` signals an error if it does not find a method for the message being sent. Write a function `send-check` that acts like `send` if a method is available, but simply returns `nil` without signaling an error if no method is available.

6.2.4 Printing Objects

The Lisp-Stat printing system prints an object by sending it the `:print` message. The method for this message inherited from the root object produces output that looks something like

```
#<Object: 2106610, prototype = *OBJECT*>
```

The fact that the printed result starts with #< indicates that this is a form the reader cannot understand. As an alternative, you can define your own :print method. To conform with the way this method is called, it should take an optional stream argument and print to the standard output if no stream is supplied. As an example, we can define a :print method for x as

```
(defmeth x :print (&optional (stream t))
  (format stream "#<-a>" (send self :title)))
```

Then asking the interpreter to print the object x produces

```
> x
#<a data set>
```

6.3 Prototypes and Inheritance

6.3.1 Constructing Prototypes and Instances

We could define additional data set objects by repeating the process of making an object, adding data and title slots, and defining a set of methods. But this would involve considerable duplication of effort. Instead, we can repeat the process once to construct a prototype data set, and then use that prototype as a template to generate additional data sets.

Prototypes are generated using the defproto macro. In its simplest form it is called as

```
(defproto <name> <instance slots>)
```

The first argument, *<name>*, should be a symbol; it is not evaluated. The second argument should be an expression that evaluates to a list of symbols. defproto takes the following actions:

- It constructs a new object, assigns the object to the global variable *<name>*, and installs a slot proto-name with value *<name>*. This slot is used by the default :print method to construct the printed representation of an object.
- It takes the list of symbols produced by the *<instance slots>* expression, and installs a slot named instance-slots with this list as its value in the new object. Then it installs in the object a slot with value equal to nil for each symbol in the instance-slots list.

The *instance-slots* list specifies the names of slots that are to be added to each object created from a prototype. Different instances created from a prototype typically differ only in the values of their instance slots. We can define a data set prototype as

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

Prototype objects can be used like any other objects. In particular, we can give a value to the `:title` slot of our data set prototype as

```
(send data-set-proto :slot-value 'title "a data set")
```

and we can give it `:describe`, `:data`, `:title`, and `:print` methods using expressions of the form

```
(defmeth data-set-proto :describe ...)
(defmeth data-set-proto :data ...)
(defmeth data-set-proto :title ...)
(defmeth data-set-proto :print ...)
```

But the main use for a prototype is as a template for constructing new objects.

An instance of a prototype is an object that inherits from the prototype and contains as its slots the instance slots specified for the prototype. An instance is constructed from a prototype by sending the prototype the `:new` message. The method for this message, inherited from the root object, takes the following actions:

- It creates a new object inheriting from the prototype.
- It adds slots specified by the prototype's `instance-slots` list to the new object, and initializes these slots to their values in the prototype.
- It sends the new object the `:isnew` initialization message with the arguments, if any, supplied in the call to the `:new` message.
- It returns the new object.

The `:isnew` method inherited from the root object requires no arguments, but it allows any slots in an object to be initialized using corresponding keyword arguments.

We can create an instance of the `data-set-proto` prototype by sending `data-set-proto` the `:new` message with a `:data` keyword argument to install a value for the `data` slot:

```
> (setf x (send data-set-proto :new :data (chisq-rand 20 5)))
#<Object: 2779762, prototype = DATA-SET>
```

The printed representation of the new object reflects the name of the prototype used to produce it. As before, we can examine our new data set by sending it the `:describe` message:

```
> (send x :describe)
This is a data set
The sample mean is 5.353018
The sample standard deviation is 3.679801
```

Since any new data set needs a value for its data slot, it is a good idea to write an `:isnew` method that requires a value to be specified for this slot. For example, we might define the new method as

```
(defmeth data-set-proto :isnew (data &key title)
  (send self :data data)
  (if title (send self :title title)))
```

This allows the title to be specified as a keyword argument.

The set of methods and slots accessible by inheritance is not frozen when an object is created. For example, having created `x` as an object inheriting from `data-set-proto`, we can now define a `:plot` method for `data-set-proto` as

```
(defmeth data-set-proto :plot () (histogram (send self :data)))
```

This method is inherited by `x` as well, and sending `x` the `:plot` message now produces a histogram of its data.

Finally, you might want to define a constructor function for your prototypes:

```
(defun make-data-set (x &key (title "a data set") (print t))
  (let ((object (send data-set-proto :new x :title title)))
    (if print (send object :describe))
    object))
```

You can also make up *generic functions* to replace formally sending messages:

```
(defun describe-data (data) (send data :describe))
(defun plot-data (data) (send data :plot))
```

6.3.2 Prototypes Inheriting from Prototypes

The `defproto` macro can be used to set up a prototype that inherits from other objects, usually other prototype objects. The full form of a call to this macro is

```
(defproto <name> <instance slots> <shared slots> <parents> <doc string>)
```

All arguments except the first are evaluated. The last four arguments are optional. Both `<instance slots>` and `<shared slots>` should be either `nil` or lists of symbols; they default to `nil`. The symbols in `<shared slots>` name additional slots to be installed in the new prototype but not in instances. `<parents>` should be an object or a list of objects, usually but not necessarily prototypes. If the `<parents>` argument is omitted, it defaults to the root object. If a documentation string is supplied, it is installed for use by the `:help` message.

If `defproto` is called with explicit parents, then the new object is constructed to inherit directly from `<parents>`. In addition, the list of instance

slots to be included in the new prototype is determined as the union of the instance slots of the parents and the list specified as the *(instance slots)* argument. Thus you only need to specify any additional instance slots you need that are not already included in the parents. As **defproto** installs instance slots required by the parents into the prototype, it initializes the slots to their inherited values.

Let's use **defproto** to set up a prototype for a time series object that inherits from the data set prototype defined above. The slots **data** and **title** will be included automatically. In addition, we might like to include slots **origin** and **spacing** to specify the origin and spacing of the observation times. This leads to the definition

```
(defproto time-series-proto
  '(origin spacing) () data-set-proto)
```

Since no shared slots are needed, the shared slots argument is the empty list. But we have to supply it explicitly in order to be able to specify the parent prototype **data-set-proto**.

The **title** slot for the new prototype object initially contains the value inherited from **data-set-proto**:

```
> (send time-series-proto :title)
"a data set"
```

The **:title** method, like all other methods in **data-set-proto**, is inherited by the new prototype. We can install a more appropriate title as

```
(send time-series-proto :title "a time series")
```

The value of this slot is used to initialize the **title** slots of time series instances. We can define accessors for the new **origin** and **spacing** slots as

```
(defmeth time-series-proto :origin (&optional (origin nil set))
  (if set (setf (slot-value 'origin) origin))
  (slot-value 'origin))
```

and

```
(defmeth time-series-proto :spacing (&optional (sp nil set))
  (if set (setf (slot-value 'spacing) sp))
  (slot-value 'spacing))
```

Default values for these slots can then be installed as

```
(send time-series-proto :origin 0)
```

and

```
(send time-series-proto :spacing 1)
```

We can now construct a time series object to represent a short moving average series as

```
(let* ((e (normal-rand 21))
      (data (+ (select e (iseq 1 20))
                (* .6 (select e (iseq 0 19))))))
  (setf y (send time-series-proto :new data)))
```

The `:isnew` method inherited from `data-set-proto` requires the `data` argument. If we send this new time series the `:describe` message, the method inherited from `data-set-proto` produces

```
> (send y :describe)
This is a time series
The sample mean is -0.3386171
The sample standard deviation is 0.9543879
```

6.3.3 Overriding and Modifying Inherited Methods

The `:plot` method inherited from `data-set-proto` produces a histogram of the time series. It would be more appropriate to plot the series against time. We can define a new method for `:plot` in the time series prototype, thus *overriding* the inherited `:plot` method, as

```
(defmeth time-series-proto :plot ()
  (let* ((data (send self :data))
         (time (+ (send self :origin)
                   (* (iseq (length data))
                      (send self :spacing))))))
    (plot-points time data)))
```

The summary produced by the inherited `:describe` method is not unreasonable, but it would be preferable to add some statistics that are appropriate for a time series, such as the autocorrelation. To add an autocorrelation to the information printed by `:describe`, we could override the method inherited from `data-set-proto` with a new method written from scratch. But it is easier to be able to invoke the inherited method, and then add on the new line. The way to call an inherited method from within the definition of an overriding method is to use the function `call-next-method`. This function finds the next method of the same name in the precedence list, starting its search after the object that owns the method being executed. It then applies that method to the current object and any additional arguments it is given. For the time series example, assuming we have a function `autocorrelation`, we can define a new `:display` method as

```
(defmeth time-series-proto :display (&optional (stream t))
  (call-next-method stream))
```

```
(format stream
        "The autocorrelation is ~g~%"
        (autocorrelation (send self :data))))
```

The function **autocorrelation** might be defined as

```
(defun autocorrelation (x)
  (let ((n (length x))
        (x (- x (mean x))))
    (/ (mean (* (select x (iseq 0 (- n 2)))
                (select x (iseq 1 (- n 1))))))
        (mean (* x x)))))
```

With this definition, the result of sending the **:display** method to the moving average time series **y** is

```
> (send y :describe)
This is a time series
The sample mean is -0.3386171
The sample standard deviation is 0.9543879
The autocorrelation is 0.4251622
```

The **call-next-method** function can also be used to produce an alternate definition of the **:isnew** method for the data set prototype. The definition

```
(defmeth data-set-proto :isnew (data &rest args)
  (apply #'call-next-method :data data args))
```

takes advantage of the fact that the next **:isnew** method, the method for the root object, allows slots to be initialized using keyword arguments. It is still possible to provide a title with the **:title** keyword, since any arguments beyond the required **data** argument are passed on to the next **:isnew** method.

At times we may wish to call a method belonging to an object that is not in the inheritance path of the current object, or a method that is not the next method in the precedence list. We can do this using the function **call-method**. This function is called as

```
(call-method <owner> <selector> <arg 1> ... <arg n>)
```

The precedence list of **<owner>** is searched for a method for the message **<selector>**, and that method is applied to the current object and any additional arguments. Using **call-method**, the **:describe** method for **time-series-proto** can be defined as

```
(defmeth time-series-proto :describe (&optional (stream t))
  (call-method data-set-proto :describe stream)
  (format stream
        "The autocorrelation is ~g~%"
        (autocorrelation (send self :data))))
```

Both `call-next-method` and `call-method` can only be used in the body of a method definition.

The object system does not allow you to hide slots and messages intended for internal use from public access. To protect yourself against breaking inherited methods, you should make sure not to modify slots or override methods unless you are sure your modifications are consistent with the way these slots and methods are used by other methods. Again, it is a good idea to avoid direct slot access by using accessor methods whenever possible.

Exercise 6.5

Set up a prototype to represent a paired sample. Define appropriate methods for `:describe` and `:plot`.

Exercise 6.6

Construct a prototype for a data set consisting of several independent samples, and define appropriate methods for `:describe` and `:plot`.

6.4 Additional Details

This section presents some lower-level details of the Lisp-Stat object system. It can be omitted on first reading.

6.4.1 Creating Objects and Prototypes

Instead of constructing an object using a prototype, you can construct an object using the `make-object` function. The arguments to this function are the parents of the new object. If `make-object` is called with no arguments, then it constructs an object that inherits from the root object. The data set object of Section 6.2.2 could thus be constructed as

```
(setf x (make-object))
```

Objects constructed by `make-object` contain no slots, even if they are constructed to inherit from a prototype object. Any slots the object needs have to be added explicitly using the `:add-slot` message.

A prototype is an object that contains `proto-name` and `instance-slots` slots. The easiest way to construct a prototype is to use the `defproto` macro. But you can also take an existing object, such as the object `x` constructed above, and turn it into a prototype by sending it the `:make-prototype` message. This message takes two arguments, a symbol to use as the value for the `proto-name` slot, and a list of symbols representing additional instance variables. The method for this message finds the instance slots specified by the object's parents, if any, combines them with the instance slots specified in the message, and installs the result in the `instance-slots` slot. It then

installs a slot for every element of the `instance-slots` list, unless such a slot already exists. New slots are initialized with the values inherited from the object's ancestors. We could thus convert our object `x` into a prototype by using the expression

```
(send x :make-prototype 'data-set-proto '(data title))
```

This approach can be useful in interactive development of prototypes. It can also be used to construct temporary prototypes that are to be used within the body of a function or `let` expression, but do not require the creation of a global prototype.

The Lisp-Stat prototype system is implemented by the `defproto` macro and the root object's methods for the `:new` and `:make-prototype` messages. These methods should not be overridden unless you intend to modify the prototype system.

6.4.2 Additional Methods and Functions

Several additional methods and functions are available for examining objects.

The precedence list of an object can be obtained by using the message `:precedence-list`, and the list of its immediate parents can be obtained by using `:parents`. For example, for the prototypes of Section 6.3

```
> (send time-series-proto :precedence-list)
(#<Object: 1470410, prototype = TIME-SERIES-PROTO>
 #<Object: 1471600, prototype = DATA-SET-PROTO>
 #<Object: 680374, prototype = *OBJECT*>)
> (send time-series-proto :parents)
(#<Object: 1471600, prototype = DATA-SET-PROTO>)
```

The predicate `kind-of-p` can be used to determine if one object inherits from another. For example, since `time-series-proto` inherits from `data-set-proto`

```
> (kind-of-p time-series-proto data-set-proto)
T
> (kind-of-p data-set-proto time-series-proto)
NIL
```

It is possible to change the inheritance of an object by sending it the `:reparent` message with the new parent or parents as arguments. We could, for example, ask our data set `x` to inherit from `time-series` by using the expression

```
(send x :reparent time-series-proto)
```

Reparenting an object does not affect the precedence lists of any objects created as its descendants. At the time of writing, the ability to reparent an

object is an experimental tool. It should be used with caution, as there is no way to ensure that the object has the slots expected by methods in its new precedence list.

6.4.3 Shared Slots

The `defproto` macro allows for the specification of shared slots that are installed in the prototype but are not duplicated in an instance. This facility is not used very often, but can occasionally be helpful. As an example, suppose that you would like to be able to keep track of all instances of a particular prototype. You can do this by defining the prototype as

```
(defproto myproto '(...) '(instances))
```

and defining a method for the `:new` message as

```
(defmeth myproto :new (&rest args)
  (let ((object (apply #'call-next-method args)))
    (setf (slot-value 'instances)
          (cons object (slot-value 'instances)))
    object))
```

Since this is a modification of the prototype mechanism, it is reasonable to override the root object's `:new` method in this case.

A word of caution may be appropriate at this point. Ordinarily, if an object is created, it will remain in existence so long as it is the value of some variable or the component of some structure. Once it is no longer referenced in any way, the Lisp system automatically reclaims its storage in the garbage collection process. You do not need to explicitly deallocate it. However, if a prototype keeps track of its instances, then the instances will always be referenced from the prototype and will never be reclaimed. You will therefore have to arrange to explicitly inform the prototype that an object is no longer needed, for example, by defining a `:dispose` method as

```
(defmeth myproto :dispose (object)
  (setf (slot-value 'instances)
        (remove object (slot-value 'instances))))
```

After sending the `:dispose` message to the prototype with an object as its argument, the object is no longer referenced from the prototype and will eventually be garbage collected if there are no other references to it. The `:dispose` message must be sent to the prototype, not the object, since it needs to be able to modify the slot `instances` owned by the prototype.

6.4.4 The Object Documentation System

The object system allows you to install documentation strings for any topic of your choice in an object. Topics are identified by symbol names, and can

be set or retrieved by using the `:documentation` message. For example, the expression

```
(send data-set-proto :documentation :title)
```

returns the documentation string for the topic `:title`, if there is one. If there is no documentation string for this topic, the message returns `nil`. Documentation strings for message selector keywords are usually installed by the `defmeth` macro, but can also be installed by using the `:documentation` message. The expression

```
(send data-set-proto :documentation
      :title "sets or returns the title")
```

installs a documentation string for `:title`. This string is used by the `:help` message when asked for help on `:title`:

```
> (send data-set-proto :help :title)
:TITLE
sets or returns the title
```

Documentation strings are stored in `documentation` slots. When retrieving the documentation for a topic, the `:documentation` method searches all `documentation` slots in an object's precedence list. When asked to install a new documentation string, the method installs it in the object receiving the message. If the object does not have a `documentation` slot, one is created. In order not to interfere with this system, you should not change the value of the `documentation` slot of an object directly.

The message `:doc-topics` returns a list of all symbols for which documentation strings are available. For the data set prototype, the result of sending this message is

```
> (send data-set-proto :doc-topics)
(:TITLE :REARENT :INTERNAL-DOC :OWN-METHODS ...)
```

This message is used by the `:help` message when it is sent with no argument:

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

```
:ADD-METHOD :ADD-SLOT :DELETE-METHOD ...
```

The heading in this help message is based on the documentation for the topic `proto`. If `defproto` is given a documentation string, then that string is installed under the topic `proto`. If we install a documentation string directly, for example, as

```
(send data-set-proto :documentation
      'proto "A generic data set.")
```

then the heading produced by the help message is more reasonable:

```
> (send data-set-proto :help)
DATA-SET-PROTO
A generic data set.
Help is available on the following:
...
```

The message :delete-documentation takes one argument, a topic symbol, and deletes the documentation string for the topic, if there is one, from the documentation for the receiving object.

6.4.5 Saving Objects

At times it is useful to save an object to a file in order to be able to restore it for use in another session or on another machine. It does not appear to be possible to develop a strategy that will work for all possible objects. Lisp-Stat therefore adopts the convention that an object that can be saved to a file should respond to the :save message. This message should return an expression that can be printed to a file. The expression should construct a replica of the object when it is read back in and evaluated. For example, for the data-set-proto prototype a :save method could be defined by using the backquote mechanism as

```
(defmeth data-set-proto :save ()
  '(send data-set-proto :new
         ',(send self :data)
         :title ',(send self :title)))
```

The result of sending this message to an instance of data-set-proto is

```
> (send x :save)
(SEND DATA-SET-PROTO :NEW
      (QUOTE (-1.651335 1.912723 ...))
      :TITLE (QUOTE "a data set"))
```

The result expression has been edited to make it more readable.

The savevar function uses the :save message to save variables whose values are objects to files. Some objects do not provide :save methods. Attempting to save such objects will produce an error.

6.5 Some Built-in Prototypes

Lisp-Stat contains a number of built-in prototypes. A subset of the inheritance graph for these prototypes is shown in Figure 6.1. Many of these

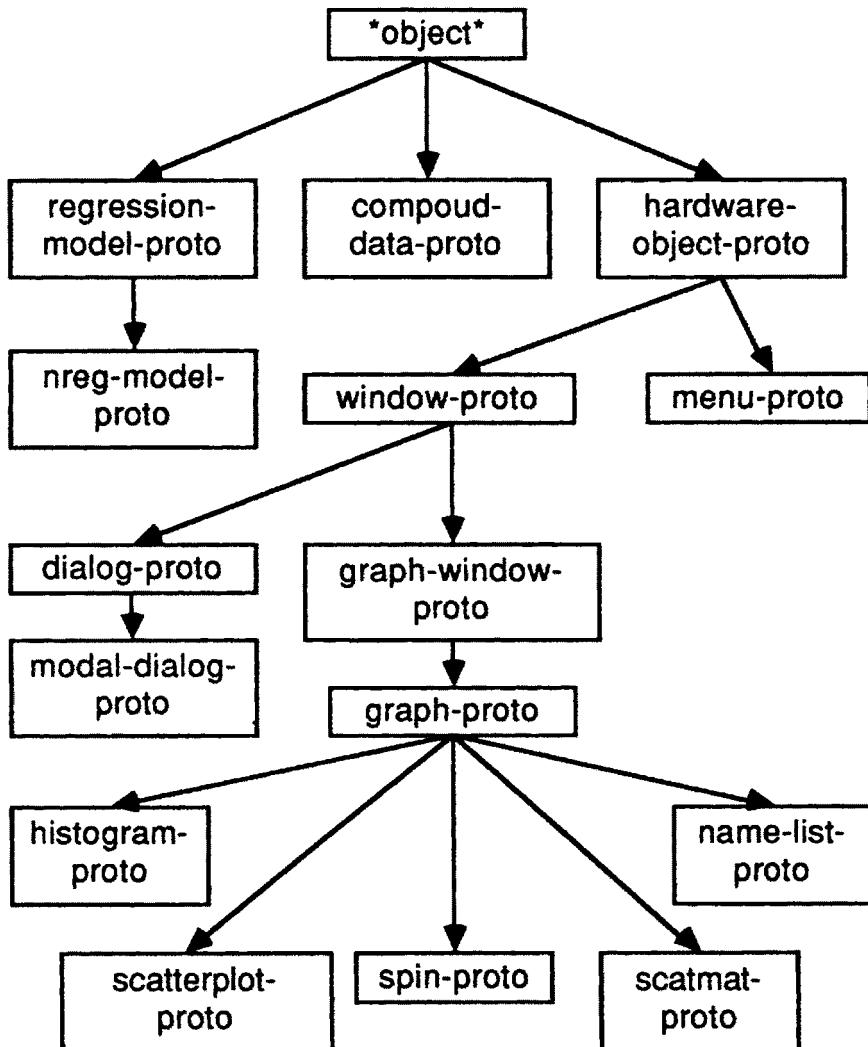


Figure 6.1: Inheritance graph for some Lisp-Stat prototypes.

prototypes are related to the graphical part of the system and are described in the following chapters. This section briefly presents three of the non-graphical prototypes: the compound data prototype, the regression model prototype, and the nonlinear regression model prototype.

6.5.1 Compound Data Objects

So far, we have only considered lists and arrays as possible compound data items. More general collections can also be defined as *compound data objects*. The vectorized arithmetic system and the function `map-elements` are designed to apply to these objects as well. A compound data object is any object inheriting from `compound-data-proto`, and should have the following characteristics:

- It contains a sequence of elements that can be extracted by using the message `:data-seq`.
- The length of its data sequence can be determined by using the message `:data-length`.
- It can make a new object that is like itself, except for a new data sequence, by using the `:make-data` message. The argument to this message can be a list or a vector.

As a simple example, let's define an object to represent a point in two-dimensional space as an object with two instance slots, its cartesian coordinates `x` and `y`:

```
(defproto point-proto '(x y) '() compound-data-proto)
```

We can define methods for accessing the slots as

```
(defmeth point-proto :x (&optional (x nil set))
  (if set (setf (slot-value 'x) x))
    (slot-value 'x))
```

and

```
(defmeth point-proto :y (&optional (y nil set))
  (if set (setf (slot-value 'y) y))
    (slot-value 'y))
```

An initialization method can then be defined by

```
(defmeth point-proto :isnew (x y)
  (send self :x x)
  (send self :y y))
```

The definitions for the three required methods are

```
(defmeth point-proto :data-length () 2)
```

```
(defmeth point-proto :data-seq ()
  (list (send self :x) (send self :y)))
```

and

```
(defmeth point-proto :make-data (seq)
  (send point-proto :new (select seq 0) (select seq 1)))
```

We can also define a :print method that shows the two coordinates by

```
(defmeth point-proto :print (&optional (stream t))
  (format stream
    "#<a point located at x = ~d and y = ~d>"
    (send self :x)
    (send self :y)))
```

Now we can construct a sample point a:

```
> (setf a (send point-proto :new 3 5))
#<a point located at x = 3 and y = 5>
```

The vectorized arithmetic system extracts the data sequence of a compound data object by sending it the :data-seq message. If the first compound argument to a vectorized function is a compound data object, then the result is constructed by sending that object the :make-data message. Since our point objects obey the required protocol, this means that we can, for example, use the functions + to add two points, * to multiply a point by a constant, or log to take the logarithm of the coordinates of a point:

```
> (+ a a)
#<a point located at x = 6 and y = 10>
> (* a 2)
#<a point located at x = 6 and y = 10>
> (log a)
#<a point located at x = 1.098612 and y = 1.609438>
```

The convention of determining the type of result based on the first compound data argument in a vectorized function call is of course arbitrary, but it generally produces reasonable results.

Exercise 6.7

Define a matrix object prototype that produces a nicely printed representation and allows arbitrary row and column labels.

6.5.2 Linear Regression Models

Linear regression model objects have already been introduced in Section 2.6. This subsection gives some additional details.

The regression model prototype is `regression-model-proto`. It is constructed by an expression like

```
(defproto regression-model-proto
  '(x y intercept weights included
    predictor-names response-name case-labels
    sweep-matrix basis
    total-sum-of-squares
    residual-sum-of-squares))
```

Thus the prototype has 12 instance slots, no shared slots, and inherits from the root object. The first eight slots contain information describing the model. Each of these slots can be read and modified by accessor methods named by the corresponding keywords `:x`, `:y`, `:intercept`, and so on. Changing any of the first five slots requires that the estimates in the model object be recomputed. Recomputation occurs the next time a computed result is needed. The last four slots are used internally by the computation method. Reader methods named by the corresponding keywords can be used to read the contents of the slots, but not to change their values. The slot accessor and reader methods are the only methods that use the slots directly; all other methods use the reader and accessor methods.

The remaining methods in the regression model prototype can be broken down into several groups. The methods for `:df`, `:num-cases`, `:num-coefs`, and `:num-included` return information on the dimensions of the model. The `:sum-of-squares` method returns the residual sum of squares. The method for `:x-matrix` returns a matrix of a column of ones, if the intercept term is included, followed by the independent variables corresponding to the basis used in the fit. The matrix $(X^T X)^{-1}$, or $(X^T W X)^{-1}$ in the weighted case, is returned by `:xtxinv`, where X is the matrix returned by `:x-matrix`. A list of the indices of the columns used in the fit is returned by the `:basis` message.

Estimates and standard errors of coefficients are returned by `:coef-estimates` and `:coef-standard-errors`. `:sigma-hat` returns the estimated residual standard deviation. Fitted values are returned by `:fit-values`, and the sample squared multiple correlation coefficient is returned by `:r-squared`.

Messages useful for residual and influence analysis are `:cooks-distances`, `:externally-studentized-residuals`, `:leverages`, `:raw-residuals`, `:residuals` and `:studentized-residuals`. Two residual plotting methods are `:plot-bayes-residuals` and `:plot-residuals`.

A summary of the model is printed by `:display`. The `regression-model` function sends this message to the new model if its `:print` keyword argument

is non-nil. The method for `:save` attempts to return an expression that can be evaluated to reproduce the model object.

The messages `:compute` and `:needs-computing` are intended primarily for internal use. The `:compute` method recomputes the estimates. The method for `:needs-computing` called with no arguments determines whether the model needs to be recomputed. Called with an argument of `t`, it tells the object to recompute itself the next time one of the computed slots is accessed. This allows several characteristics of the model to be modified at once without having the model recomputed after each modification.

The `:isnew` method only sets up the model object and marks it as needing to be recomputed. It does not allow slots to be specified by keyword arguments. Thus if you want to build up a model without using the constructor function `regression-model`, you can send the prototype the `:new` message to construct a new object, and then add the model components to the object using the accessor methods.

6.5.3 Nonlinear Regression Models

The nonlinear regression prototype `nreg-model-proto` is defined by an expression like

```
(defproto nreg-model-proto
  '(mean-function theta-hat epsilon count-limit verbose)
  nil
  regression-model-proto)
```

Thus it inherits from `regression-model-proto` and has five additional instance slots. The slot `mean-function` holds the model mean function, and `theta-hat` contains the parameter estimates. The remaining slots are used to control the iteration process for calculating the estimates. The `theta-hat` slot can be read with a reader method named by the corresponding keyword. The remaining slots can be read and modified by using accessor methods. Use of the accessors to change any of these slots marks the object as needing to be recomputed.

The approach of having the nonlinear regression model prototype inherit from the linear model prototype may seem rather peculiar. Nonlinear, or rather not necessarily linear, regression models are a superset of linear regression models, and in all examples given so far, new prototypes have been introduced as specializations of their parent. In this case I am using the inheritance relationship to represent an analogy rather than a specialization. Fitting and calculating fitted values is of course done very differently in nonlinear models than in linear models. But standard errors of estimates, leverages and similar quantities are often calculated by linearizing the nonlinear model using an expansion around the estimated parameter values. In this linearized approximation, the role of the X matrix is played by the Jacobian of the mean function at the estimated parameter values. With this

identification, formulas for standard errors, leverages, and so on, for the linear case make sense in the nonlinear case as well, at least as first order approximations.

This analogy is captured by the inheritance relationship. The nonlinear model prototype contains a new method for `:compute` that finds the maximum likelihood estimates by a Gauss-Newton search with backtracking. It then installs the Jacobian as the `x` slot, and applies the inherited `:compute` method to fill in the slots for the linear regression computations using the linearized approximation. Finally, it installs the appropriate value of the residual sum of squares in its slot. New methods are also defined for `:fit-values`, for `:coef-estimates` and for `:save`. A new message called `:parameter-names` is added to provide names for use in the display method; these are stored in the `predictor-names` slot. Since it no longer makes sense to change the X matrix, the intercept, or the predictor names, the corresponding messages are overridden with new methods that only allow read access, not modification. Finally, a method for the message `:new-initial-guess` is added to allow the iterative search for estimates to be restarted at a new set of initial values. All other methods are inherited from the linear regression prototype. They provide reasonable approximations based on the linearized model. If, for example, a better definition of leverages is found, then the inherited method can be overridden with a new method for the nonlinear model. But until then the first order approximation is automatically available.

The nonlinear regression model could have been defined as the parent of the linear model. There are other possible generalizations of the normal linear regression model that could be placed ahead of the linear model as well. One example is a generalized linear model in the sense of McCullagh and Nelder [42]. Both could be used as parents to the linear model with multiple inheritance, as described in the next section. It is not clear which approach is to be preferred, the approach used here of reasoning by analogy or an approach based on an ordering by generality. Both approaches have advantages. The approach used here may be more suited to a situation in which a new type of model is being explored and needs to benefit from relationships to a simpler, better understood setting. The ordering by generality, on the other hand, is probably better suited for a situation in which the parts of the hierarchy are well understood and not likely to undergo further development. Even though the normal nonlinear regression model has been studied extensively, there is still considerable room for improvement in the methodology for its analysis. This is the reason I chose the representation described here.

6.6 Multiple Inheritance

All examples used so far involved objects inheriting from a single parent. The precedence lists for these objects can therefore be determined by simply

backtracking from an object through its parents to the root object. With multiple inheritance (the ability of an object to have several parents) there is no longer an obvious way to construct a precedence list, and we need to adopt some conventions.

The precedence list is designed to provide a total ordering on an object and its ancestors. This ordering is constructed using the following rules¹:

- An object always precedes its parents.
- The list of an object's parents provides a local ordering on these objects. This ordering must be preserved.
- Duplicate objects are eliminated from the ordering. An object that appears more than once is placed as close to the beginning of the ordering as possible without violating the other rules.
- An error is signaled if no such ordering exists.

Here is an example of the ordering produced on a set of prototypes.² Define six prototypes as

```
(defproto food)

(defproto spice () () food)

(defproto fruit () () food)

(defproto cinnamon () () spice)

(defproto apple () () fruit)
```

and

```
(defproto pie () () (list apple cinnamon))
```

Then the precedence list of pie is

```
> (send pie :precedence-list)
(#<Object: 3557930, prototype = PIE>
 #<Object: 3562838, prototype = APPLE>
 #<Object: 3340422, prototype = FRUIT>
 #<Object: 3567602, prototype = CINNAMON>
 #<Object: 3408398, prototype = SPICE>
 #<Object: 3445690, prototype = FOOD>
 #<Object: 2668162, prototype = *OBJECT*>)
```

¹These rules are based on the rules used in CLOS and in the Symbolics Flavors system as described in Bobrow et al. [12,56] and Moon [46].

²This example is taken from Bobrow et al. [12,56] and Moon [46].

Not all parent lists lead to consistent orderings. For example,

```
> (defproto spice-cake () () (list spice pie))
error: inconsistent precedence order
```

This attempted definition requires `spice` to precede `pie`, whereas the definitions already made require the reverse ordering. Using the reverse ordering in `spice-cake` does work:

```
> (defproto spice-cake () () (list pie spice))
SPICE-CAKE
```

A useful way to take advantage of multiple inheritance is to create a library of several prototypes, each including a specific feature, such as the ability to print or to keep track of sets of labels. New prototypes that need some of these features can then be created as descendants of the group of objects representing the features that are needed. Prototypes created for the purpose of being mixed together with other prototypes are called *mixins*. Mixins are used in Section 10.4 to implement grand tours.

6.7 Other Object Systems

The origins of object-oriented programming are usually traced back to the simulation language Simula developed in the late 1960s and early 1970s. The most extensive use of the paradigm can be found in the Smalltalk language developed at Xerox Palo Alto Research Center in the mid 1970s [33]. Smalltalk is a pure object-oriented language: all data are objects. In Smalltalk an expression like

```
1 + 2
```

is interpreted as “send the object 1 the message to add the object 2 to itself and return an object representing the result.”

Smalltalk is unusual in the extent to which it adheres to the object-oriented paradigm. Most other systems supporting object-oriented programming take a hybrid approach by adding object-oriented programming tools to an existing language. Examples of such systems are C++ and Objective C, and the Classcal and Object Pascal extensions of Pascal [22,54,57]. A number of object-oriented extensions have also been developed for Lisp, including Object Lisp, Flavors, LOOPS, Common Objects, and the Common Lisp Object System CLOS [12,36,46,55].

Object-oriented ideas have received considerable attention in recent years because they are ideal for use in designing graphical user interfaces, such as the interface developed as part of the Smalltalk system and popularized by the Apple Macintosh. Graphical objects such as windows and menus are very naturally represented as software objects, and actions taken by a user, such as asking a window to resize itself or pressing a button to initiate an action,

translate naturally into messages sent to these objects. The two object-oriented Pascal dialects mentioned above were both developed primarily to support programming the Macintosh.

Most compiled object systems are based on the idea of *classes*. A class is a general template or definition of an object, much like a structure definition in C or a record definition in Pascal. In particular, a class is typically not an object. Objects are constructed as instances of a particular class. A class definition specifies the slots, or instance variables, that its instances should have, and it also specifies one or more other classes as its superclasses. Thus to construct an object with a particular set of slots, you must first define an appropriate class, and then construct the object as an instance of this class. In compiled systems such as C++ or Object Pascal, classes are defined in a program's code and remain fixed once the program is compiled. In contrast, instances of classes can be allocated dynamically at run time. Smalltalk and the object system based on the CLOS standard class are class-based systems.

An alternative to the class-based approach is to allow any object to inherit directly from any other object, as in the system incorporated in Lisp-Stat. Other examples of such systems include the Object Lisp system and the Trellis/Owl language [53]. These systems are often called prototype-based systems. Prototypes, also called *exemplars*, can be used much like classes to help organize an inheritance hierarchy, but their use is not mandatory. This gives prototype-based systems some additional flexibility. The price for this flexibility is that a class specification cannot be used to provide type-checking information, or to automatically ensure that an object has the appropriate set of slots needed by a particular method. Since slots can be added dynamically, it is also more difficult for a compiler to optimize methods during compilation.

The debate on the relative merits of class-based and prototype- or exemplar-based systems is still under way in the object-oriented programming literature [38,40]. One point of view that has been expressed is that the class-based approach may be more suited for the implementation and maintenance of well understood, mature programs, while the prototype-based approach may be better for the development stage of a program. Since interactive statistical computing is in many ways a form of experimental programming [45], I believe that this argues in favor of using the prototype-based approach as part of a statistical system. The ability to add slots to objects at any time is also analogous to the ability to add components to *S* structures, which has proved to be very useful in the *S* environment [6,7]. For these reasons I chose to develop a prototype-based system for use in Lisp-Stat.

The Common Lisp community has recently developed a standard for object-oriented programming called the Common Lisp Object System CLOS [12,36,56,67]. This system provides a standard class-based object system incorporating multiple inheritance and multimethods for dispatching on multiple arguments. It also provides a *metaclass protocol* for implementing alternative object systems. The Lisp-Stat system differs significantly from the standard CLOS system, but can be implemented within the framework pro-

vided by the metaclass protocol.

6.8 Some Examples

6.8.1 A Rectangular Data Set

Data sets used in regression analyses, for example, are often viewed as a table or a matrix, with cases represented by rows and variables by columns. We can add a data set prototype for this type of data that takes a “column-oriented” view by representing each variable as a list, along with a label string. In addition, we can also allow for case labels for use in plots. The prototype can be defined as

```
(defproto rect-data-proto '(vlabels clabels) () data-set-proto)
```

The slots `vlabels` and `clabels` hold the variable and case labels, respectively.

The initialization method installs the data, a title, and a set of supplied labels. If no labels are supplied, default labels are constructed using the variable and case indices:

```
(defmeth rect-data-proto :isnew
  (data &key title variable-labels case-labels)
  (let ((n (length (first data)))
        (m (length data)))
    (send self :data data)
    (if title (send self :title title))
    (send self :variable-labels
      (if variable-labels
          variable-labels
          (mapcar #'(lambda (x) (format nil "X~a" x))
                  (iseq 0 (- m 1)))))
    (send self :case-labels
      (if case-labels
          case-labels
          (mapcar #'(lambda (x) (format nil "~a" x))
                  (iseq 0 (- n 1)))))))
```

This method should be modified to check that all data lists are the same length.

We need methods for the messages `:variable-labels` and `:case-labels` for setting and retrieving labels,

```
(defmeth rect-data-proto :variable-labels
  (&optional (labels nil set))
  (if set (setf (slot-value 'vlabels) labels))
  (slot-value 'vlabels))
```

```
(defmeth rect-data-proto :case-labels
  (&optional (labels nil set))
  (if set (setf (slot-value 'clabels) labels))
  (slot-value 'clabels))
```

and we can give the prototype a new title, for use as the default title in instances:

```
(send rect-data-proto :title "a rectangular data set")
```

As a test example, we can again use the stack loss data introduced in Section 5.6.2:

```
(setf stack
  (send rect-data-proto :new (list air temp conc loss)
        :variable-labels
        (list "Air" "Temp." "Concent." "Loss")))
```

The inherited plot method is not particularly appropriate. We should use different plots depending on the number of variables in the data set:

```
(defmeth rect-data-proto :plot ()
  (let ((vars (send self :data)))
    (labels (send self :variable-labels)))
  (case (length vars)
    (1 (histogram vars :variable-labels labels))
    (2 (plot-points vars :variable-labels labels))
    (3 (spin-plot vars :variable-labels labels))
    (t (scatterplot-matrix vars :variable-labels labels)))))
```

The inherited :describe method still works, but it is not very useful:

```
> (send stack :describe)
This is a rectangular data set
The sample mean is 46.3333
The sample standard deviation is 29.6613
```

Exercise 6.8

Write a more reasonable :describe method for the rectangular data set prototype.

Exercise 6.9

Suppose you have some data files containing columns consisting of label strings enclosed in double quotes and followed by numbers. Write a function that reads data in from a file and produces a rectangular data set object.

6.8.2 An Alternate Data Representation

Another approach to representing rectangular data, proposed by McDonald [44], is to take a more “row-oriented” view by representing individual observations, or cases, as objects. Each case is given slots to hold basic case data, and methods are defined to extract this information. These methods are thus used to represent the variables. Other methods, representing derived variables, can be defined in terms of these basic ones.

To start off, an observation prototype might be defined as

```
(defproto observation-proto '(label))
```

All observations should at least contain an observation label. To ensure this, we can use the function `gensym` in the initialization method. It constructs a symbol whose name is a specified string followed by an integer that is incremented each time `gensym` is called. The function `string` takes the symbol and returns its print name:

```
(defmeth observation-proto :isnew (&rest args)
  (setf (slot-value 'label) (string (gensym "OBS-")))
  (apply #'call-next-method args))
```

If we assume that the label, and all other slots an observation might have, are only filled by the `:isnew` method when the object is constructed, then we only need methods for reading the slots. For example, a method for reading the label slot is given by

```
(defmeth observation-proto :label () (slot-value 'label))
```

To examine the stack loss data set using this approach, we might define a prototype observation for that data set to have slots representing the four values measured on each observation day,

```
(defproto stack-obs-proto
  '(air temp conc loss) () observation-proto)
```

Reader methods for these four slots can then be defined as

```
(defmeth stack-obs-proto :air () (slot-value 'air))
```

```
(defmeth stack-obs-proto :temp () (slot-value 'temp))
```

```
(defmeth stack-obs-proto :conc () (slot-value 'conc))
```

and

```
(defmeth stack-obs-proto :loss () (slot-value 'loss))
```

An advantage of this approach is that we can add additional information to any of the observations. If one observation happens to have been recorded under unfavorable conditions, a slot containing a note to this effect can be added. This is more difficult if we record our variables as lists of numbers, or if we combine our data into a matrix. Another advantage is that derived variables can be constructed to behave like the basic variables represented by the messages :air, :temp, :conc and :loss. For example, a variable representing the logarithm of the loss can be defined as

```
(defmeth stack-obs-proto :log-loss () (log (send self :loss)))
```

Once we have a list of observation objects, we can place them in a data set object for further examination. If we define the data set to contain a slot for the list of observations,

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

then we can define a method for obtaining the list of values of a variable for each observation as

```
(defmeth data-set-proto :values (var)
  (mapcar #'(lambda (x) (send x var)
              (slot-value 'data))))
```

Using this method, we can define a `:plot` method to take one or more variable message keywords, find the values for the variables, and plot these values using a plot that seems most appropriate for the number of variables specified:

Since the variables are given to :plot as message keywords, it makes no difference whether they refer to data stored in slots or represent derived variables.

It is no longer possible to define a :describe method that describes all variables in the data set, since this is no longer a meaningful concept: the observations could contain a variety of different variables with only a few in common. Instead, we can have :describe take one or more variable message keywords as arguments, and provide a reasonable summary for these variables. A simple method that takes only one variable can be defined as

```
(defmeth data-set-proto :describe (var)
  (let ((vals (send self :values var)))
    (format t "Variable Name: ~a~%" var)
    (format t "Mean: ~g~%" (mean vals))
    (format t "Median: ~g~%" (median vals))
    (format t "Standard Deviation: ~g~%"
            (standard-deviation vals))))
```

Returning to the stack loss example, we can construct a list containing observation objects for each observation in the data set with the expression

```
(setf stack-list
  (mapcar
   #'(lambda (air temp conc loss label)
       (send stack-obs-proto :new
             :air air
             :temp temp
             :conc conc
             :loss loss
             :label (format nil "~d" label)))
   air temp conc loss (iseq 0 20)))
```

The data set object is then constructed by

```
(setf stack-loss (send data-set-proto :new :data stack-list))
```

We can now examine the :loss variable,

```
> (send stack-loss :describe :loss)
Variable Name: :LOSS
Mean: 17.52381
Median: 15
Standard Deviation: 10.17162
```

or the derived :log-loss variable,

```
> (send stack-loss :describe :log-loss)
Variable Name: :LOG-LOSS
Mean: 2.72591
Median: 2.70805
Standard Deviation: 0.5235305
```

or we can construct a rotating plot of the variables :log-loss, :air, and :temp using

```
(send stack-loss :plot :log-loss :air :temp)
```

Exercise 6.10

Write a :describe method that takes one or more variable keywords as arguments and produces a reasonable summary for these variables.

Exercise 6.11

Suppose you have some data files containing columns consisting of label strings enclosed in double quotes and followed by numbers. Write a function that takes a file name string, reads data in from a file, and produces a data set object containing appropriate observation objects.

Exercise 6.12

Write a :regression method for a data set that returns a regression model object for a specified set of variables. Use the variable keywords to provide predictor and response labels for the regression model object.

Chapter 7

Windows, Menus, and Dialogs

The remaining chapters of this book describe the Lisp-Stat graphics system. The main purpose of the system is to support using, customizing, and developing dynamic statistical graphs. As part of this effort, the system provides access to user interface tools such as menus and dialog windows. The system is designed as a high-level toolkit that is implemented on top of a microcomputer or workstation window system, such as the Macintosh *Toolbox* or the *X11* system.

This chapter introduces some basic window system concepts, and outlines the design of the Lisp-Stat graphics system. In addition, this chapter also describes basic Lisp-Stat windows, menus, and dialog windows. The following two chapters describe Lisp-Stat's graphics windows.

7.1 The Window System Interface

Many modern microcomputers and workstations use a graphical interface to make them and their programs easier to use. These interfaces divide the screen into separate windows that can overlap and be moved around like pieces of paper on a desktop. Menus are used to issue commands to programs, and dialog windows are used to ask the user for information.

Graphical interfaces have many features in common, which is not surprising as they are all descendants of the system developed by Xerox and popularized by the Macintosh. But there are some differences in the conventions used for presenting menus, determining which window is to receive characters typed at the keyboard, and so on. These conventions are called the *user interface guidelines*. For example, on the Macintosh menus are usually placed in a menu bar. In the *SunView* system, menus are popped up when the right button on a three button mouse is clicked on various parts of

a window. As another example, to select a section of text on the Macintosh, you can click the single mouse button at the start of the selection and drag the mouse over the selection. You can also extend a selection by clicking the mouse at the end of the extended selection while holding down the *shift* key. In *SunView*, the convention is to start a selection by clicking the left mouse button, and to extend it by dragging or by clicking the middle mouse button.

A window system is a set of programming tools designed to support writing programs with a graphical user interface. The window system is responsible for drawing windows and menus, and for monitoring the user's actions. The window system also takes care of responding to certain user actions, such as moving or resizing a window, and informs the program owning the window if the window needs to be updated in any way. Some window systems, for example, the Macintosh *Toolbox* and the *SunView* system, are designed to implement a particular set of user interface guidelines. Others, such as the *X11* system, are designed to support a variety of different user interfaces. The *X11* system is also intended to be used on a wide variety of different hardware. In particular, it supports systems with one-, two-, or three-button mice.

A major feature of programs with a graphical interface is the need to respond to user-generated *events*: windows may need to be redrawn when they are moved or resized, menus have to be presented when they are requested, key strokes and mouse actions need to be interpreted. This can make writing such programs considerably harder than writing a program that just runs a task to completion, or even a program that interprets text typed in one line at a time. Choosing a good programming strategy can, however, make this programming task much easier. In particular, the desktop metaphor makes it very natural to take an object-oriented approach in which physical windows and menus are represented in software as objects. User actions directed at these objects can then be viewed as messages.

In an object-oriented window system, a programmer only needs to construct an appropriate set of objects, and write methods for any messages to which the objects need to respond. The window system then runs an *event loop* that monitors user actions and converts any events that need to be handled by the program into messages. For example, if the window system detects that a portion of a window has been uncovered, then it sends the object representing the window a message asking it to redraw itself. An object-oriented window system can also provide a library of standard graphics objects. A programmer can then arrange for a new object to inherit from the most suitable object, or objects, in the library. Since the objects in the library provide reasonable default methods for most messages, this greatly reduces the number of new methods a programmer has to write. In addition, it helps to reduce departures from the user interface guidelines, and thus makes programs easier to use.

The Lisp-Stat graphics system is a high-level, object-oriented toolkit designed to be implemented on top of a more general window system. Some

compromises have been made to allow the system to remain reasonably simple, but at the same time to fit reasonably well into different user interface guidelines. Several concepts are treated somewhat abstractly, allowing a particular implementation to handle them in a way that is most suitable to the native user interface. For example, each Lisp-Stat graphics window has a single menu associated with it. In XLISP-STAT on the Macintosh, the system makes sure this menu is installed in the menu bar whenever the window is the front window. Under *SunView*, the system pops up the menu when the right mouse button is clicked in the window. The *X11* version of XLISP-STAT provides a button on the graphics window that pops up the menu when it is pressed. Similarly, when a window is notified that a mouse click has occurred, it is informed about any modifiers that are in effect. On the Macintosh, which has a one button mouse, modifiers are signaled by holding down the *shift* or *option* keys while pressing the mouse button. On systems with three button mice, modifiers may identify the mouse button that is pressed.

7.2 Windows

Lisp-Stat supports at least two kinds of windows: graphics windows and dialog windows. Some implementations may support additional window types, such as text windows. All windows share certain basic features. These features are collected together in the basic window prototype `window-proto`. This prototype is not intended to be used directly; its purpose is merely to collect together these common features. Both the dialog window and graphics window prototypes inherit from `window-proto`. The `:isnew` method for `window-proto` allows several features to be set with keyword arguments. Unless otherwise noted, the `:isnew` methods for more specialized window types also accept these keywords.

Every window has a title. In many user interfaces this title is displayed in a title bar at the top of the window. The message `:title` can be used to set and retrieve the current window title. For example, if `p` is a histogram window created by the `histogram` function, then its title is initially given by

```
> (send p :title)  
"Histogram"
```

If the plot contains a histogram of a variable `hardness`, then we can give the window a more appropriate title by

```
> (send p :title "Hardness")  
"Hardness"
```

The window `:isnew` method allows the initial title of a window to be specified with the `:title` keyword.

Sizes and locations of windows are measured in pixels. The `:size` message can be used to determine a window's current size. The result returned

is a list of the width and the height of the window. If *p* is our histogram window, then

```
> (send p :size)
(200 100)
```

means the window is currently 200 pixels wide and 100 pixels high. The window can be instructed to resize itself to a new size by sending it the **:size** message with two arguments, the new width and height:

```
> (send p :size 250 150)
(250 150)
```

The location of a window is described by the coordinates of the left top corner of the window, relative to the left top corner of the screen.¹ The current location can be set and retrieved with the **:location** message. The result of

```
> (send p :location)
(100 50)
```

means the left top corner of *p* is 100 pixels to the right and 50 pixels below the left top corner of the screen. The window can be moved to a new location by

```
> (send p :location 200 100)
(200 100)
```

The window prototype **:isnew** method allows the initial size and location to be specified with the keywords **:size** and **:location**. The arguments supplied with these keywords should be lists of two integers: the width and height for the size, and the left and top coordinates for the location.

The **:size** and **:location** messages determine size and location of the content of a window. Many window systems place a frame around a window's content. You can determine the size of the entire frame and the location of the frame's left top corner with the **:frame-size** and **:frame-location** messages.

For most purposes it is convenient to think of a Lisp-Stat window object and the image of the window on the screen as identical. But they are in fact distinct. The object represents a means for communicating with the image in the native window system. When a new window object is created, its image is immediately made visible, unless the **:isnew** method is given the **:show** keyword with value **nil**. The image can be removed from the screen

¹The Lisp-Stat graphics system assumes that the computer or workstation has a single screen. Systems supporting multiple screens usually identify one screen as the default screen. On such systems, locations are measured relative to the left top corner of the default screen.

temporarily by sending the object the :**hide-window** message. The :**show-window** message makes a hidden window visible again, and moves it to the front of all visible windows. Thus to hide the image of a window represented by the object *p*, we would use the expression

```
(send p :hide-window)
```

The expression

```
(send p :show-window)
```

makes the image visible again. The :**show-window** message can also be used to bring a window that is partially or completely obscured by other windows to the front.

A second method for removing the image of a window is to send the window object the :**remove** message, using an expression like

```
(send p :remove)
```

This message is intended to dispose permanently of the window image.

One reason for distinguishing between temporary hiding and permanent removal of a window image is that the native window system may need to allocate memory to represent the image. Hiding a window makes a window temporarily invisible without releasing this memory. In contrast, once a window is removed, this memory is released. Ideally, this memory management should be handled automatically, but this may not be possible for all native window systems. As a result, it is important to send the :**remove** message to windows that are no longer needed.

Most user interfaces provide a standard means for dismissing a window when it is no longer needed. On the Macintosh, windows usually contain a small rectangle, the close box, in their left top corner. In the *SunView* system, a menu attached to the window frame contains a close item. Clicking in the close box, or taking the standard action to dismiss a window, sends the :**close** message, with no arguments, to the window object. The default method for this message sends the object the :**remove** message. If you want a window *p* to be hidden when it is closed, instead of having it removed, you can define a new :**close** method as

```
(defmeth p :close () (send self :hide-window))
```

The window :**isnew** method allows the :**go-away** keyword to be used to specify whether a window is to include a closing device. Giving this keyword with value *nil* produces a window without a closing device.

At times it is useful to create dialogs or plots that are intended as subordinates to another plot. When the main plot is removed from the screen, the subordinates should be removed as well. To support this notion, you can add a subordinate to a window by sending the main window the :**add-subordinate** message with the subordinate window as its argument. A

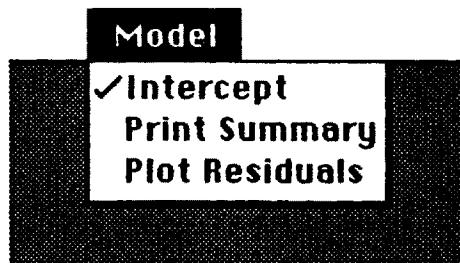


Figure 7.1: A menu for examining a regression model.

subordinate can be deleted with the `:delete-subordinate` message. The `window-proto` prototype's `:remove` method makes sure that all subordinates installed at the time a window is removed are sent the `:remove` message as well.

Exercise 7.1

The function `screen-size` returns a list of the width and height of the screen in pixels.² Use this function to write a method for a window that centers the window on the screen.

7.3 Menus

Graphical menus are devices used to ask a program to take one of several possible actions. Figure 7.1 shows a simple menu for examining a regression model. Menus are usually displayed in response to a mouse click in a menu bar, or some particular region of a window, and remain displayed until the mouse button is released. As the cursor is moved through the menu, the menu item containing the cursor is highlighted. If the mouse button is released with an item highlighted, then the action corresponding to that item is taken. If no item is highlighted when the button is released, then no action is taken. Items in a menu that are not appropriate at a particular time can be disabled. Items that are used to toggle some feature on and off can be marked with a check mark when the feature is turned on. The `Intercept` item in the menu in Figure 7.1 is checked, indicating that the model currently includes an intercept.

Lisp-Stat menus are constructed as objects inheriting from the prototype `menu-proto`. Each menu contains a list of menu items, which are objects

²On systems with multiple screens, this function returns the size of the default screen.

inheriting from the `menu-item-proto` prototype. On systems supporting hierarchical menus, items in a menu may also be other menus.

Menus can be made available for selection in several ways. The simplest approach is to install the menu in the *menu bar*. When a menu is installed in the menu bar, its title appears in the bar. Clicking on the title then pops up the menu. The menu bar is a standard component of the Macintosh user interface. Lisp-Stat implementations for other user interfaces provide a similar device, such as a window containing the titles of installed menus. Other approaches for making menus available include installing them in graphics windows or popping them up in response to a mouse click in a graphics window. These approaches are discussed further in the next chapter.

When the system is asked to put up a menu, it takes the following actions:

- It sends the `:update` message, with no arguments, to each item in the menu. This allows menu items to check whether they should be enabled or should contain a check mark.
- It presents the menu, highlighting an item if it contains the cursor. When the mouse button is released, it removes the menu.
- If an item was highlighted when the mouse button was released, the corresponding menu item object is sent the `:do-action` message, with no arguments.

Let's see how to construct the menu in Figure 7.1. The menu itself can be created by

```
(setf model-menu (send menu-proto :new "Model"))
```

The `:isnew` method for `menu-proto` requires one argument, a title string for the menu. This title is stored in the `title` slot. It can be read and changed with the `:title` message. A menu also contains an `enabled` slot. The value of this slot indicates whether the menu is enabled for selection when it is installed in the menu bar. The value of this slot can be set and retrieved with the `:enabled` accessor method. The `:isnew` method accepts a value for this slot specified with the `:enabled` keyword. The default value is `t`.

The `:items` message returns a list of the items installed in a menu. For our menu,

```
> (send model-menu :items)  
NIL
```

Initially, the menu contains no items. To complete the menu, we need to construct three items and place them in the menu.

Menu items contain the slots `title`, `mark`, `enabled`, and `action`. These slots can be read and modified using corresponding accessor methods. The `title` slot contains the string that appears in the menu. The `mark` slot is `t` or `nil`, depending on whether the item is preceded by a check mark. The

enabled slot indicates whether the item is enabled for selection or not. The `menu-item-proto` method for the `:do-action` message is defined to simply `funcall` the contents of the `action` slot with no arguments, if the value of the slot is not `nil`. Thus you can control the action of a menu item in two ways: by placing a function in the `action` slot, or by defining a new `:do-action` method.

Suppose we have set up a regression model object for the data in Section 2.5.1 as the value of a global variable,

```
(setf *current-model*
      (regression-model (list hardness tensile-strength
                                abrasion-loss)))
```

We can then construct the summary and plotting items for our menu as

```
(setf display-item
      (send menu-item-proto :new "Print Summary"
            :action #'(lambda ()
                        (send *current-model* :display))))
```

and

```
(setf plot-item
      (send menu-item-proto :new "Plot Residuals"
            :action #'(lambda ()
                        (send *current-model* :plot-residuals))))
```

The intercept item is constructed by

```
(setf intercept-item
      (send menu-item-proto :new "Intercept"
            :action
            #'(lambda ()
                (send *current-model* :intercept
                      (not (send *current-model* :intercept))))))
```

The action function for `intercept-item` toggles the intercept on and off. We can define a method for the `:update` message that ensures that this item contains a check mark whenever the model contains an intercept:

```
(defmeth intercept-item :update ()
  (send self :mark
    (if (send *current-model* :intercept) t nil)))
```

The `if` expression is used to ensure that the argument given to the `:mark` message is either `t` or `nil`.

Items are placed in a menu by sending the menu the `:append-items` message with one or more menu items as arguments. The new items are appended in order to the end of the current set of items. To install the three items in our menu, we would use

```
(send model-menu :append-items
      intercept-item display-item plot-item)
```

Items can be removed from a menu by sending the `:delete-items` message with one or more items as arguments. A menu item can only be installed in one menu at a time. If an item is installed in a menu, the `:menu` message returns the menu containing the item. For the intercept item,

```
> (send intercept-item :menu)
#<Object: 1803062, prototype = MENU-PROTO, title = "Model">
```

If the item is not installed in a menu, the `:menu` message returns `nil`.

Finally, we can install the menu in the menu bar by sending it the `:install` message:

```
(send model-menu :install)
```

After evaluating this expression, the title of the menu should appear in the menu bar. Clicking the mouse on the menu title then produces the menu shown in Figure 7.1. You can remove a menu from the menu bar by sending the menu object the `:remove` message.

If you use a menu like this as part of a larger program, you may not always want to allow the intercept to be changed. You can disable the intercept item by using the expression

```
(send intercept-item :enabled nil)
```

Menus contain the slots `title`, `enabled`, and `items`. Menu items contain the slots `title`, `enabled`, `mark`, and `action`. Both menus and menu items may contain additional slots other than the ones listed here. These slots are intended for internal use and should not be modified. Several of the slots mentioned here are also used by the interface to the native window system and will not be handled properly if they are modified directly. They should therefore only be changed by using the accessor methods.

Two additional tools that are useful in constructing and debugging menus are the prototype `dash-item-proto` and the function `sysbeep`. The dash item prototype inherits from the menu item prototype and represents a disabled item containing a dashed line. Such items are useful for separating groups of items in a menu. You can construct a new dash item with the expression

```
(send dash-item-proto :new)
```

The `sysbeep` function simply generates a tone, and can be used either as a warning or as part of a dummy routine during debugging. With no arguments, it produces a tone of a standard length. You can vary the length of the tone by supplying an integer argument.

Exercise 7.2

Design a menu to use with the data set prototypes in the examples of Chapter 6. You might have the menu send the :**describe** and :**plot** messages.

7.4 Dialog Windows

Dialog windows are special windows used to obtain information from a user, or to give a user the opportunity to send a program instructions. A dialog window can contain several different kinds of *dialog items*:

- buttons
- check boxes
- clusters of radio buttons
- scroll bars
- static and editable text fields
- scrollable one- and two-dimensional lists of text strings

There are two kinds of dialog windows: *modal* dialogs and *modeless* dialogs. Modal dialogs are more common. A modal dialog is a means for asking a user a direct question that has to be answered before the program can proceed. When a modal dialog is presented, all input is directed to that dialog until it is dismissed, usually by clicking a button. Modeless dialogs, on the other hand, are like any other window in a window-based program, except they contain buttons, check boxes, and so on. Modeless dialogs are used much like menus, to cause an event-driven program to take a particular action when a user, say, presses a button. The standard dialog for opening a file on the Macintosh is a modal dialog; the Macintosh *Control Panel* is a Modeless dialog.

A number of the menu items in Lisp-Stat's plot menus open dialogs to obtain additional information. Most of these dialogs are modal and can be constructed by using a few simple functions for constructing standard dialogs.

7.4.1 Some Standard Modal Dialogs

The simplest modal dialog notifies the user that something important has occurred. The function **message-dialog** takes a string argument and puts up a modal dialog containing the string and a single **OK** button. The dialog then waits until the user presses the button. For example, the expression

```
(message-dialog "There is no variable named X")
```

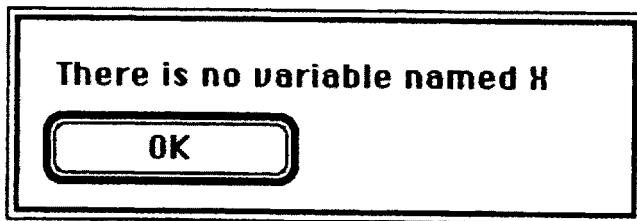


Figure 7.2: A message-dialog.

presents the dialog shown in Figure 7.2.

Instead of announcing that a variable does not exist, it may be better to offer the user a choice of continuing or aborting an operation. The function `ok-or-cancel-dialog` presents a modal dialog with two buttons, an **OK** and a **Cancel** button. If the **OK** button is clicked, `t` is returned. Otherwise, `nil` is returned. Thus you might use the expression

```
(let ((s (format nil "There is no variable named X. ~%"
                  Do you want to try another variable?"))))
  (ok-or-cancel-dialog s))
```

instead of the previous one.

`ok-or-cancel-dialog` takes an additional optional argument. If this argument is true (the default) then the **OK** button is the *default button*. That is, pressing the *return* key is equivalent to pressing the **OK** button. If this argument is supplied as `nil`, the **Cancel** button is the default button.³

A wider choice can be presented using `choose-item-dialog`. This function takes a prompt string and a list of strings, and presents a dialog with the prompt, a radio button for each element of the list, and **OK** and **Cancel** buttons. If the **Cancel** button is pressed, `nil` is returned. If **OK** is pressed, the zero-based index of the chosen element is returned. As an example, to allow the choice of a dependent variable for a regression, we might use

```
(choose-item-dialog "Dependent variable:" '("Y0" "Y1" "Y2"))
```

to presents a dialog with three choices. If the item labeled "Y1" is selected, this expression would return the value 1. The keyword argument `:initial` can be used to specify the index of the choice to highlight when the dialog is presented; the default index is 0.

The function `choose-subset-dialog` is similar, but allows one or more items to be selected by using a dialog with a series of check boxes. It returns

³Some systems do not include the notion of a default button. On such systems the default button specification is ignored, and you always have to press the appropriate button to dismiss the dialog.

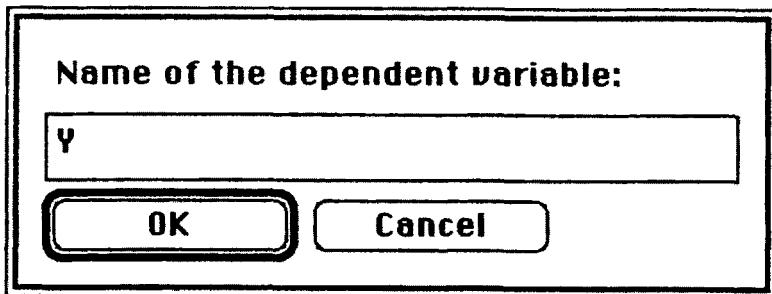


Figure 7.3: A `get-string-dialog`.

a list containing a list of the selected indices if the **OK** button is clicked, and `nil` for the **Cancel** button. This form of the return value allows you to distinguish between a cancellation and a selection of no items. The `:initial` keyword can be used to specify the list of indices of the items to check when the dialog is presented; by default, none are selected. To choose the independent variables in a regression, with all variables chosen as the default, we can use

```
(choose-subset-dialog "Independent variables" '("X0" "X1" "X2")
                      :initial '(0 1 2))
```

If the first and third are chosen, the result returned is

```
((0 2))
```

If you can't reduce the information you want from the user to a choice among a few items, you can ask for a string or an expression to be typed in. To request a string, you can use `get-string-dialog`. The expression

```
(get-string-dialog "Name of the dependent variable:"
                      :initial "Y")
```

opens the modal dialog shown in Figure 7.3. If you click the **OK** button, the string of the current contents of the editable text field is returned. Otherwise, `nil` is returned.

`get-value-dialog` is similar to `get-string-dialog`, but it treats the text in its editable text field as a Lisp expression that is to be evaluated. If the **OK** button is clicked, then the expression in the text field is read and evaluated, and a list of the value is returned. If the **Cancel** button is clicked, `nil` is returned. Thus if the user enters the expression

```
(+ 1 2)
```

and clicks the **OK** button, then the result is the list

(3)

If the user enters the expression

nil

then the result is the list

(NIL)

This allows you to distinguish a result expression with value **nil** from a cancellation. An initial expression for the editable text field supplied by the **:initial** keyword is converted to a string by using **format** and the **print-style** conversion of the **~s** directive.

Exercise 7.3

Extend the menu developed in the previous exercise to allow a new data set to be read in from a file.

Exercise 7.4

Modify the data set menu's plot item to present a dialog asking for the variables to be plotted. Choose the type of plot most appropriate to the number of variables.

7.4.2 Modeless Slider Dialogs

The most useful modeless dialogs are sliders constructed to control an animation. In Section 2.7 we used the function **sequence-slider-dialog** to construct a slider to control an animated power transformation plot. This function requires one argument, a Lisp sequence. The dialog scrolls through the sequence, calling an action function each time the scroll bar is moved. The action function is called with one argument, the current element of the sequence. The action function is specified by the **:action** keyword argument to **sequence-slider-dialog**. As a simple example, the expression

```
(sequence-slider-dialog
  (iseq 100 150)
  :action #'(lambda (x) (format t "Current element: ~s~%" x)))
```

opens a sequence slider that prints the current sequence element each time the scroll bar is adjusted. By default, the current element of the argument sequence is displayed in the value field. If this is not appropriate, you can use the keyword argument **:display** to provide an alternate display sequence of the same length. The **:title** keyword can be used to specify a title string for the window. An alternate label string for labeling the value can be given with the **:text** keyword; the default is the string "Value".

The sequence slider dialog object can be sent the `:value` message to set or retrieve the index of the current sequence element. The `:action` message can be used to retrieve or change the action function.

A second form of slider is constructed by `interval-slider-dialog`. This function requires one argument, a list of the lower and upper limits of the interval, and returns an interval slider object. The interval is discretized into a sequence of real numbers. The number of points used can be specified using the `:points` keyword; the default is 30. The values for the number of points and the end points in the range argument are used exactly if the argument `:nice` is `nil`. Otherwise, they will be adjusted slightly to produce nice printed values. By default this argument is `t`. The `:text`, `:title` and `:action` keyword arguments to this function are used as for sequence sliders. The expression

```
(interval-slider-dialog
 '(0 1)
 :points 50
 :action #'(lambda (x) (format t "Current value: ~s~%" x)))
```

constructs an interval slider that scrolls through the unit interval, discretized to approximately 50 points, and prints the current value each time it is changed.

The interval slider dialog object can be sent the `:value` message to set or retrieve the current value. A new value is rounded to the nearest point in the discretization of the interval. The action function for interval sliders can also be set and retrieved with the `:action` message.

7.5 Constructing Custom Dialogs

The standard dialogs produced by the functions described in the previous section are adequate for most purposes. But occasionally we may want to construct a more elaborate dialog for a particular problem. For example, we might want to use the dialog shown in Figure 7.4 for specifying a regression model. We can construct such a dialog using the `dialog` and `dialog item` prototypes.

7.5.1 The Dialog Prototypes

There are two dialog prototypes, `dialog-proto` for modeless dialogs and `modal-dialog-proto` for modal dialogs. These prototypes inherit from the window prototype `window-proto`. A new dialog is constructed by sending the appropriate prototype the `:new` message with a list of dialog items and possibly some keyword arguments. The `:title`, `:location`, `:size`, and `:go-away` keywords can be used to set the corresponding properties for the new dialog. The `:go-away` keyword is ignored for modal dialogs. An additional

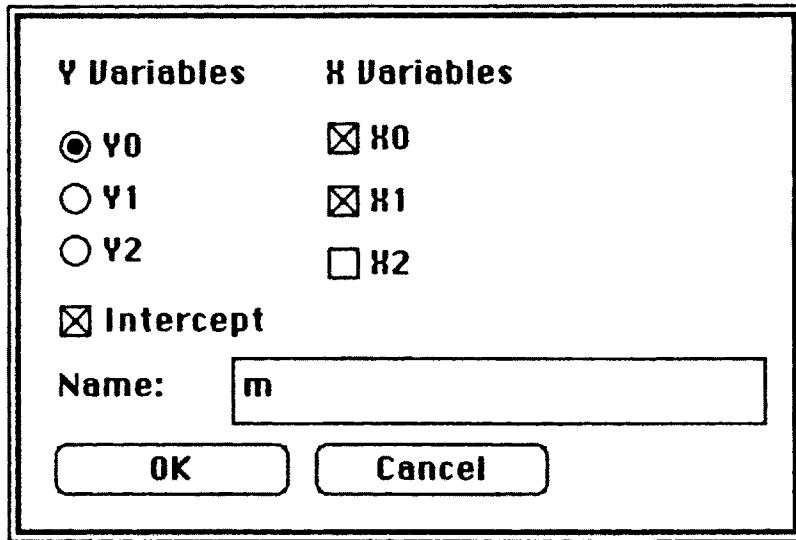


Figure 7.4: A dialog for building a regression model.

keyword argument is `:default-button`. If supplied, it should be a dialog button object or `nil`. The default is `nil`, meaning no default button.⁴ The default button in a dialog can be set and retrieved with the `:default-button` message.

The item list given as the first argument to the dialog `:isnew` method specifies the layout of the dialog. If the list consists only of dialog items, then the items are arranged in a column. A list appearing in the item list represents a row. A list within a row is a subcolumn, and so on. For example, the item list used to construct the dialog in Figure 7.3 looks like

```
(prompt-text editable-text (ok-button cancel-button))
```

There is one column consisting of a static text item for the prompt, an editable text item for entering a string, and a row of two buttons. Like menu items, the dialog items are themselves objects. The list of items in a dialog can be retrieved with the `:items` message.

When a dialog is initially constructed, it acts in a modeless fashion, regardless of the prototype used to create it. When a user action occurs that is directed at a particular dialog item, the system adjusts the item by, for example, entering a character or moving a scroll bar, and sends the item object the `:do-action` message. Generally only buttons, scroll bars, and scrollable lists need to react to these messages.

⁴This argument is ignored on systems that do not support the concept of a default button.

To use a modal dialog, you send it the `:modal-dialog` message. The method for this message requires no arguments, but accepts one optional argument. Unless this optional argument is supplied as `nil`, the method sends the dialog the `:remove` message before returning. The `:modal-dialog` method runs a loop that forces all user actions to be directed at the dialog. The loop continues until the dialog is sent the `:modal-dialog-return` message with one argument. That argument can be any Lisp item, and is returned as the result of the `:modal-dialog` message.

7.5.2 Dialog Items

All dialog items inherit from `dialog-item-proto`. Like the `window-proto` prototype, `dialog-item-proto` is not used directly. It is used as a parent to collect common functionality for the six specific kinds of dialog items: toggle items, choice items, text items, buttons, scroll bars, and list items.

Dialog items can respond to the `:do-action` and `:action` messages. The `:action` message retrieves or sets the value of the `action` slot. This slot should hold `nil` or a function taking zero or one arguments, depending on the type of item. The default method for `:do-action` funcalls the value of the `action` slot if the value of the slot is not `nil`. Dialog item `:isnew` methods also accept an `:action` keyword argument to set the action function.

Dialog item `:isnew` methods take `:location` and `:size` keyword arguments. If they are supplied, they are used to position the item within the dialog. Both should be lists of two integers, measured in pixels. If size or location are not specified, the item list supplied to the dialog `:isnew` message and the data in the dialog items are used to structure the dialog around the items. You will probably never need to set the location of an item directly, unless you need things to line up particularly nicely. But you may want to use the `:size` keyword to set the size of editable text items or scroll bars. The value of the size argument should be a list of the width and height of the item. If you do need to supply a location, it should be a list of the left top corner coordinates of the item, relative to the dialog window's left top corner.

The dialog in Figure 7.4 contains several check boxes, or toggle items, for determining whether the model should include an intercept or a particular independent variable. Toggle items inherit from `toggle-item-proto`. The `:isnew` method for this prototype requires a string argument, which is to be used as a label for the item. In addition to the standard keyword arguments, a `:value` keyword can be supplied. The value should be `t` or `nil` to toggle the item on or off, respectively. The `:value` message can be sent to a toggle item to set or retrieve the current value. The intercept item for our dialog can be created as

```
(setf intercept
      (send toggle-item-proto :new "Intercept" :value t))
```

The item is initially checked, since :value is supplied as t. No action function is installed, since the value of the item is not needed until the **OK** button is pressed. The items for the three independent variables can be constructed by

```
(setf x-item-0 (send toggle-item-proto :new "X0"))
(setf x-item-1 (send toggle-item-proto :new "X1"))
(setf x-item-2 (send toggle-item-proto :new "X2"))
```

The dependent variable for the regression model is set in the dialog by using a choice item, a cluster of radio buttons. As on a car radio, only one of the buttons is turned on at a given time. The prototype is **choice-item-proto**, and the :isnew method requires a list of strings as the labels for the buttons. The :value keyword can be used to specify the index of the item to be selected initially; by default, this index is zero. The :value message sets and retrieves the index of the current choice. A choice item for selecting the dependent variable is produced by

```
(setf y-item (send choice-item-proto :new
                    (list "Y0" "Y1" "Y2") :value 1))
```

This sets the initial choice to "Y1".

Our dialog contains three static text items: one to label the choice for the dependent variable, one to label the choices for the independent variables, and one as a prompt for the model name. There is also one editable text item for entering the model name. Static text items inherit from **text-item-proto**, and editable text items inherit from **edit-text-item-proto**. New text items are created by sending the appropriate prototype the :new message with the initial text as the first argument. The system determines the size of a text item using the initial text, unless you use the keyword argument :text-length to specify the number of characters that should fit in the field. If the dialog font is proportionally spaced, the average character size is used. The text of both static and editable text items can be set or retrieved using the :text message. The three static text items for our dialog can be constructed as

```
(setf x-label (send text-item-proto :new "X Variables"))
(setf y-label (send text-item-proto :new "Y Variables"))
(setf prompt (send text-item-proto :new "Name:"))
```

The editable item is set up as

```
(setf name (send edit-text-item-proto :new "" :text-length 15))
```

The final items needed for our dialog are the two buttons. Buttons inherit from **button-item-proto**, and are created by sending this prototype the :new message with a string, the button's label, as the argument. Buttons are usually used to initiate an action and thus need either an action function, supplied with the :action keyword, or a specialized :do-action method. For example, a button for starting a simulation might be defined as

```
(send button-item-proto :new "Start"
      :action #'(lambda (x) (send simulation :start)))
```

Buttons in a modal dialog are usually used to dismiss the dialog and should therefore send the dialog the :modal-dialog-return message with an appropriate argument. For a **Cancel** button the argument is usually `nil`. The **Cancel** button for our regression dialog could thus be defined as

```
(setf cancel (send button-item-proto :new "Cancel"
                     :action
                     #'(lambda ()
                         (let ((dialog (send cancel :dialog)))
                           (send dialog
                                 :modal-dialog-return nil)))))
```

The `:dialog` message returns the dialog object containing the item.

To simplify constructing buttons for modal dialogs, the `modal-button-proto` prototype has a `:do-action` method that calls the action function, if there is one, and sends the result to the dialog with the `:modal-dialog-return` message. If the value of the action slot is `nil`, then `nil` is given as the argument to the return method. Thus the **Cancel** button can be constructed using this prototype as

```
(setf cancel (send modal-button-proto :new "Cancel"))
```

Since no action function is supplied, the value of the `action` slot defaults to `nil`.

Pressing the **OK** button in our dialog should return the information collected in the dialog. We can define a function `collect-values` to collect the values in the dialog items into a list as

```
(defun collect-values ()
  (list (send name :text)
        (send y-item :value)
        (which (list (send x-item-0 :value)
                     (send x-item-1 :value)
                     (send x-item-2 :value))))
        (send intercept :value)))
```

The **OK** button is then produced by

```
(setf ok (send modal-button-proto :new "OK"
                  :action #'collect-values))
```

We now have all the items needed for the regression dialog. The dialog itself can be set up as

```
(setf reg-dialog
      (send modal-dialog-proto :new
            (list
              (list
                (list y-label y-item intercept)
                (list x-label x-item-0 x-item-1 x-item-2))
                (list prompt name)
                (list ok cancel))))
```

The items are arranged in one column of three rows. The first row has two subcolumns, one with *y-label*, *y-item*, and *intercept*, and one with the items for selecting the independent variables. The second row contains the prompt and the editable field for entering the model name, and the third row contains the **OK** and **Cancel** buttons. The expression

```
(send reg-dialog :modal-dialog)
```

runs the modal dialog loop, allowing the choice, toggle, and text items to be modified until one of the buttons is clicked. If the **OK** button is clicked with the configuration shown in Figure 7.4, then the result returned by this expression is

```
("m" 0 (0 1) T)
```

Since the **:modal-dialog** message was sent with no argument, the dialog is removed automatically before this expression returns.

Two other types of items are available that were not used in the regression dialog: scroll bars and scrollable lists. Basic scroll bar items scroll through a range of integers. They are constructed by sending **scroll-item-proto** the **:new** message. No arguments are required by the **:isnew** method, but several keyword arguments are available. The minimum and maximum can be set using **:min-value** and **:max-value** keywords; the defaults are 0 and 100. They can be accessed and changed later by using the **:min-value** and **:max-value** messages. The initial value of the scroll bar is the minimum. An alternate initial value can be specified with the **:value** keyword to the **:isnew** method, and the value can be set and retrieved by using the **:value** message. On most systems scroll bars can scroll in increments of one unit or in larger increments, often corresponding to a page. The size of the page increment can be specified by giving the **:page-increment** keyword to the **:isnew** method. The default is 5.

Scroll bars are usually adjusted continuously while the mouse is pressed in one of the active portions of the scroll bar. Each time this automatic scrolling operation occurs, the system sends the scroll bar object the **:scroll-action** message with no arguments. The default method for this message is identical to the **:do-action** method.

Scroll bars are often used to scroll through a sequence or an interval. Two prototypes inheriting from **scroll-item-proto** are designed to deal

with these standard cases. The `:isnew` method for `interval-scroll-item-proto` takes a list representing an interval as its argument. In addition, it accepts the keyword argument `:points` to set the number of points to use and `:text-item` to specify a text item object in which to display the current value. The `:isnew` method for `sequence-scroll-item-proto` requires the sequence as its argument. In addition to the `:text-item` keyword, it accepts the `:display` keyword for specifying an alternate display sequence. The action functions called by the default `:do-action` methods for these prototypes take one argument, the current sequence element or the current point in the interval.

As an example, we can use sequence and interval scroll items to create a modeless dialog containing two sliders for specifying the parameters of a binomial distribution with at most 50 trials. We can start with two labels for identifying the displayed values constructed by

```
(setf p-label (send text-item-proto :new "p"))
```

and

```
(setf n-label (send text-item-proto :new "n"))
```

Two text items for displaying the current values are given by

```
(setf p-value (send text-item-proto :new "" :text-length 10))
```

and

```
(setf n-value (send text-item-proto :new "" :text-length 10))
```

The scroll items can then be constructed as

```
(setf p-scroll (send interval-scroll-item-proto :new
                      '(0 1)
                      :text-item p-value
                      :action
                      #'(lambda (x) (format t "p = ~g~%" x))))
```

and

```
(setf n-scroll (send sequence-scroll-item-proto :new
                      (iseq 1 50)
                      :text-item n-value
                      :action
                      #'(lambda (x) (format t "n = ~g~%" x))))
```

The action functions simply print the new values when the scroll bars are adjusted. Finally, the dialog is produced by

```
(send dialog-proto :new (list (list p-label p-value)
                               p-scroll
                               (list n-label n-value)
                               n-scroll))
```

The final item type is a list item.⁵ This item represents a one- or two-dimensional scrollable list of text items. At most one cell can be selected at a time, the selection can be set and retrieved, and a cell's text can be changed. In addition, a double click in a cell can be detected. To create a list item, send the :new message to `list-item-proto` with a sequence or two-dimensional array of strings as argument. By default, a one column list is constructed. The number of visible columns can be set with the :columns keyword. The :do-action method takes an optional argument that is `nil` by default and is used in the call to the action function. The system calls :do-action for every click, and gives a non `nil` argument if the click is a double click - that is, if it was sufficiently close in time and location to the previous click.

The :set-text method takes a string and an index, and changes the text of the cell specified by the index to the string. The index should be a number if the list item was constructed using a sequence of strings, and it should be a list of row and column indices of the cell if an array was used.

The method :selection sets or returns the index of the currently selected cell. A `nil` index means no cell is selected.

Exercise 7.5

Design a simple menu- and dialog-driven regression analysis system.

7.6 Additional Details

The function `screen-has-color` can be used to determine whether the screen is a color screen.⁶ It take no arguments, and returns `t` or `nil` to indicate whether the screen supports color or not.

The function `active-windows` can be called, with no arguments, to obtain a list of all window objects currently displayed on the screen or temporarily hidden.

The `*features*` variable, described in Section 4.5.2, contains the symbol `windows` if a window system interface is available. If the screen supports color, then the features list also contains the symbol `color`. The feature `hierarchical-menus` is defined if the system supports hierarchical menus. These features can be used with the `#+` and `#-` read macros for conditional evaluation.

The Lisp-Stat system is designed to work in a single sequential process. As a result, a graphical method that takes a lot of time will prevent any other events from being processed until it has finished. To deal with this

⁵At the time of writing, this item type is only supported in XLISP-STAT on the Macintosh.

⁶On a system with multiple screens, the result refers to the default screen.

problem, graphics windows allow for an idle message that is sent to each window when no events are available for processing. Some systems may also provide a system-level idle event queue that can be used to execute actions not associated with a particular widow when no events are being processed.

Chapter 8

Graphics Windows

The foundation for the statistical graphs provided in Lisp-Stat consists of two layers. The lower layer, described in this chapter, provides tools for drawing lines and shapes in a window, and for responding to events generated by user actions. The second layer, described in the next chapter, adds the ability to handle data. All plots described in Chapter 2 inherit from the higher-level prototype described in the next chapter. If you are primarily interested in customizing some of the standard graphs, you might want to briefly skim this chapter and move on to the next chapter. However, even if you do not need to use the details presented here directly, you may find it helpful to understand the basic drawing model that supports the higher-level graphics objects.

8.1 The Graphical Model

Images are drawn in a window by changing the colors of the *picture elements*, or *pixels*, in the window. A graphical model provides a higher-level, unified framework for thinking about the drawing process. The graphical model underlying the Lisp-Stat graphics system is essentially a simplified version of the Macintosh *QuickDraw* model, combined with some features of the *SunView* and *X11* systems.

8.1.1 Coordinates, Drawing Modes, and Colors

Drawing operations in a graphics window take place in a conceptual *drawing canvas*. The dimensions of the canvas can be either *elastic* or *fixed*. If the dimensions are elastic, then the canvas is identical to the content of the window. If the dimensions are fixed, then the window shows only a subrectangle of the canvas. In this case the window can be positioned within the canvas by using scroll bars.

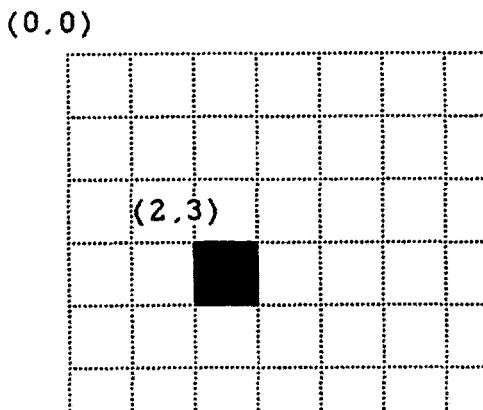


Figure 8.1: Coordinate system on a raster display.

On a raster display, the smallest unit that can be addressed on its own is the pixel. Pixels are assumed to be square. Their width is thus the natural basic unit for measuring lengths on the screen.¹ When we draw a “point” on the screen, the smallest representation we can use for the point is a pixel. Since the mathematical notation (x, y) represents a point as an entity with zero area, we need a convention for relating pixels to a mathematical coordinate system. Figure 8.1 shows a coordinate system superimposed on a set of pixels. For a graphics window, the left top corner of its canvas is the origin of the coordinate system. The x , or horizontal, axis increases to the right in units of one pixel. The y , or vertical, axis increases *downward* in units of one pixel. This is the opposite of the usual mathematical convention, but it is the standard system used for addressing raster displays at the pixel level. A pixel is referenced using the coordinates of its left top corner. Thus the pixel $(2, 3)$ represents the rectangle with vertices $(2, 3)$, $(3, 3)$, $(3, 4)$, and $(2, 4)$.

Each pixel can be drawn in a different color. On a monochrome display the only colors available are black and white. On a color display other colors may be available.² On most color displays the available colors include black, white, red, green, blue, cyan, magenta, and yellow. Initially, all pixels in a window are the same color, the *background color*. As you draw in the window you change the color of certain pixels to the *drawing color*. *Erasing* certain objects, like rectangles, replaces the pixels that make up the objects by the

¹On some PC's pixels are not square, which may lead to some distortions. On most workstations and on the Macintosh, pixels are in fact square.

²Even on a color display it may be useful to work only in black and white, since black and white drawing is sometimes faster than color drawing.

background color. *Painting* such objects replaces their pixels by the drawing color.

Background and drawing colors are *states* of the drawing system. You can change them at any time, and the new colors remain in effect until they are changed again. Changing the current background or drawing colors does not affect any of the pixels directly. It only affects the results of future drawing operations. Each window has its own set of graphical states.

Another state of the drawing system is the *drawing mode*. The drawing mode can be either *normal* or *XOR*. In normal mode, the effect of a drawing operation on a pixel is independent of the previous color of the pixel. In XOR mode, coloring a pixel reverses the pixel's color. This is well-defined when drawing in black and white: if the pixel is black, it becomes white; if it is white, it becomes black. In color the results are not predictable. But even on color displays the XOR mode has the property that drawing an object twice, thus reversing the pixels that make up the object twice, restores the screen to its original state. This feature makes XOR drawing a useful animation tool, since it allows you to move an object over a background without having to explicitly save and restore the background. XOR drawing is used to implement the brush rectangle in Lisp-Stat plots.

Two final states of the drawing system are the *line type* and the *line width*. Lines are drawn using a *pen*. A pen is a square set of pixels, parallel to the axes, that can be viewed as moving from one point to another, leaving ink on the pixels it touches. The size of the pen is specified by the line width. A pen can be used to draw lines in different styles or types. Two types are supported: *solid* and *dashed*.³

8.1.2 Drawable Objects

The graphics system provides higher-level drawing routines for rendering a number of different objects, such as rectangles, characters, and bitmaps.

Rectangles, Ovals, and Arcs

Several basic drawing routines deal with rectangles. A rectangle is a rectangular set of pixels with sides parallel to the screen boundaries. Rectangles are specified in terms of four numbers: the coordinates of the left top corner, the width, and the height. These four numbers are the coordinates of a rectangle. Rectangles can be erased, painted, or framed. When a rectangle is framed, the pen is run along the inside of the rectangle's boundary. The current drawing mode and line type are used. Painting the rectangle sets the color of its pixels to the drawing color; erasing it sets the pixel colors to the background color.

³On some systems drawing solid lines of width 1 may be substantially faster than drawing thicker lines or dashed lines.



Figure 8.2: A symbol and a character bitmap.

Ovals are ellipses with their axes parallel to the coordinate axes. They are specified by their surrounding rectangles. Arcs of ovals are specified by the surrounding rectangles of their ovals and two angles, a start and an increment. The start angle is measured counterclockwise from the horizontal. The increment is the angle from the start to the end of the arc. The angles are measures in degrees. Ovals and arcs can also be framed, painted, and erased.

Polygons

Polygons are specified by a list of pairs of integers, representing the coordinates of the vertices of the polygon. By default, these coordinates are interpreted as absolute coordinates relative to the origin. They can also be interpreted as relative coordinates. In this case the second set of coordinates represents displacements relative to the first, the third represents displacements relative to the second, and so on. Polygons can be framed, filled, and erased. For filling and erasing, a chord is drawn between the first and last points if they are not identical. Framing does not close the polygon; if you want to frame a closed polygon, you must make sure the first and last vertices are identical.

Symbols, Strings, and Bitmaps

Characters and point symbols consist of *bitmaps*, rectangular collections of black and white pixels. Examples are shown in Figure 8.2. Characters are drawn by superimposing their bitmaps on the window. In the normal drawing mode, window pixels under the black parts of a character are colored with the drawing color, and pixels under the white part are left unchanged. In XOR mode the pixels under the black part are reversed.

Plot symbols are pairs of small bitmaps, one for a highlighted point and one for an unhighlighted point. The rules for drawing symbols in normal mode are a little different than for characters: window pixels under a sym-

bol's white area are painted with the background color. This ensures that a highlighted symbol can be replaced by an unhighlighted symbol by drawing the new symbol on top of the old symbol. The symbols provided in Lisp-Stat come in different sizes. They can be three, four, or five pixels wide.

Symbol bitmaps are specially designed and implemented for fast drawing. You can also draw arbitrary rectangular bitmaps which are specified as two-dimensional arrays of zeros and ones, with zeros representing white and ones representing black. A bitmap image can be accompanied by a *mask* bitmap. Only pixels corresponding to ones in the mask are affected by the drawing operation. In the normal drawing mode, black pixels in the image that are included in the mask are drawn in the drawing color, white pixels in the background color. The rule for drawing a character corresponds to using a mask identical to the character image; the rule for a symbol uses a mask that is a square enclosing the symbol image.

8.2 Graphical Messages

8.2.1 Initialization and State Messages

The prototype for a basic graphics window is `graph-window-proto`. A new window is created by sending the prototype the `:new` message. The `:isnew` method for this prototype does not require any arguments, but the standard window keywords `:title`, `:location`, `:size`, and `:go-away` can be supplied. The default title is "Graph Window", and the default value of `go-away` is `t`. Several other keyword arguments are available as well. The window can be given a menu with the `:menu` keyword. The keywords `:black-on-white`, `:has-v-scroll` and `:has-h-scroll` set the initial drawing and background colors and determine whether the window has vertical or horizontal scroll bars. By default, the drawing color is black, the background color is white, and the window has no scroll bars. The `:isnew` method displays an image on the screen unless the `:show` keyword is supplied with value `nil`. The expression

```
(setf w (send graph-window-proto :new))
```

constructs a window with all options at their default settings.

The states of a window's drawing system can be determined or set using the `:back-color`, `:draw-color`, `:draw-mode`, `:line-width`, and `:line-type` messages:

```
> (send w :back-color)
WHITE
> (send w :draw-color)
BLACK
> (send w :draw-mode)
NORMAL
```

```
> (send w :line-width)
1
> (send w :line-type)
SOLID
```

These accessor messages take new state values as optional arguments. The line width must be a positive integer. The drawing mode is specified as one of the symbols `normal` or `xor`. The drawing mode of our window can be changed to `xor` and back to `normal` by

```
> (send w :draw-mode 'xor)
XOR
> (send w :draw-mode 'normal)
NORMAL
```

The line type is specified as `solid` or `dashed`. Colors can be any of the symbols in the list returned by the function `color-symbols`. On monochrome displays the only color symbols available are `black` and `white`; on color displays the available symbols should include `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta`, and `yellow`.

Color-related commands may not have any effect unless the window is using color drawing. The message `:use-color` can be used to determine if this is the case. It can also be used to instruct the window to use or not use color by giving an optional argument that is true or `nil`, respectively. For our window,

```
> (send w :use-color)
NIL
```

Initially, windows use black and white drawing. If your system supports color, you can instruct the window to use color drawing with the expression

```
(send w :use-color t)
```

On a monochrome display all colors other than white are treated as black.

The message `:reverse-colors` is occasionally useful. It interchanges the current drawing and background colors, and sends the window the `:redraw` message.

8.2.2 Basic Drawing Messages

To draw a single pixel in a window, you can send the window object the `:draw-point` message. This message takes two integer arguments, the coordinates of the point. For example,

```
(send w :draw-point 15 10)
```

colors the pixel located at (15, 10) in the drawing color.

To draw a line, you can send the window object the :draw-line message with four integer arguments, the *x* and *y* coordinates of the start, and the *x* and *y* coordinates of the end of the line. The expression

```
(send w :draw-line 5 10 50 70)
```

draws a line from (5, 10) to (50, 70).

Rectangles are drawn using the messages :frame-rect, :erase-rect, and :paint-rect. These messages require four integer arguments, the *x* and *y* coordinates of the left top corner of a rectangle, and the rectangle's width and height. The expression

```
(send w :frame-rect 50 70 100 50)
```

draws a framed rectangle 100 pixels wide and 50 pixels high, with its left top corner at the end of the line just drawn.

Ovals inscribed in rectangles are drawn with the :frame-oval, :erase-oval, and :paint-oval messages. These messages require four integer arguments to specify the surrounding rectangle. We can put a solid oval 10 pixels below the rectangle with the expression

```
(send w :paint-oval 50 130 100 50)
```

Arcs are drawn using the :frame-arc, :paint-arc, and :erase-arc messages. These messages take six arguments. The first four are integers describing the enclosing rectangle of the arc ellipse. The remaining two arguments are real numbers specifying the start and increment angles of the arc. A painted arc starting 30 degrees above the horizontal with an arc increment of 45 degrees can be added to our window by

```
(send w :paint-arc 50 70 100 50 30 45)
```

Polygons are drawn with the messages :frame-poly, :paint-poly, and :erase-poly. These messages require a list of lists of pairs of integers representing the vertices. By default, these pairs are interpreted as coordinates relative to the origin. If an optional argument is supplied as nil, then the coordinates of each vertex but the first are taken to be relative to the previous vertex. We can use :erase-poly to draw a triangle above the rectangle in our window:

```
(send w :frame-poly '((50 50) (150 50) (100 10) (50 50)))
```

This expression uses absolute coordinates. An equivalent expression using relative coordinates is

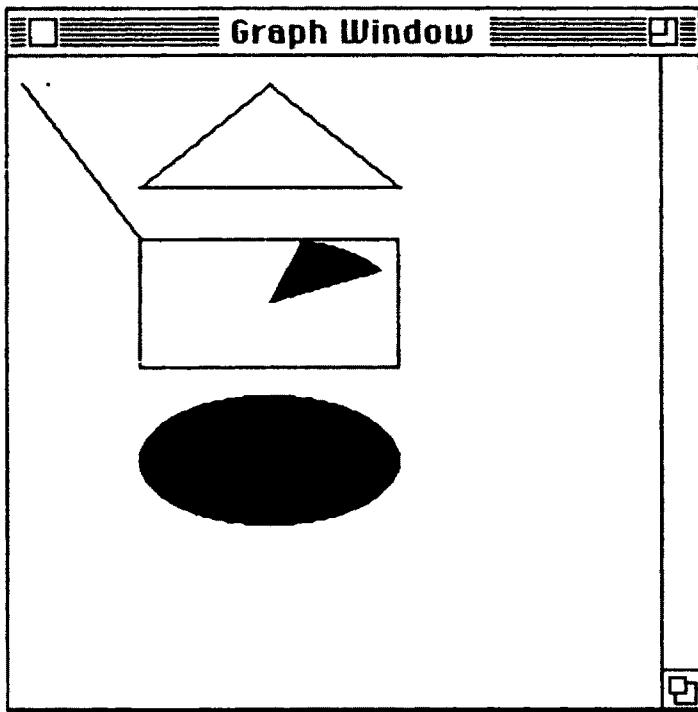


Figure 8.3: A basic graphics window.

```
(send w :frame-poly '((50 50) (100 0) (-50 -40) (-50 40)) nil)
```

The cumulative result of these drawing operations is shown in Figure 8.3.

On most systems, if you cover the window *w* with another window and then uncover it, the drawings disappear – the window has no memory of its contents. The standard plots described in Chapter 2 redraw their contents whenever a window is resized or an obscured part of a window is exposed. Section 8.4 describes how to ensure that a window is redrawn when necessary.

8.2.3 Additional Messages

The entire content of a window can be erased by sending the window the `:erase-window` message. To erase our window, we could thus use

```
(send w :erase-window)
```

Instead of drawing a line of a fixed length, you might want to draw a line from the left top corner of the window to the center of the window. To do this, you need to be able to determine the width and height of the window's canvas. The messages `:canvas-width` and `:canvas-height` return this information. The line to the center can be drawn using

```
(let ((x (round (/ (send w :canvas-width) 2)))
      (y (round (/ (send w :canvas-height) 2))))
  (send w :draw-line 0 0 x y))
```

The messages :draw-string and :draw-string-up draw a string horizontally or vertically in the window. They take as arguments the string and two integers, representing the *x* and *y* coordinates of the point at which the string is to start. The expression

```
(send w :draw-string "Hello" 100 120)
```

draws the string "Hello" starting at the point (100, 120).

To be able to position a string relative to an axis mark, for example, we need to be able to determine the string's size. The message :text-width takes a string, and returns its width in pixels. The message :text-ascent takes no arguments, and returns the maximal ascent of a letter above the base line in the window's font, measured in pixels. The message :text-descent returns the maximal descent below the base line. Thus the expression

```
(let ((ta (send w :text-ascent))
      (tw (send w :text-width "Hello"))))
  (send w :draw-string-up "World" (+ 100 ta tw) 120))
```

draws the string "World" vertically just to the right of the string "Hello".

In many cases you may be able to avoid calculating starting points for drawing strings by using the :draw-text message. The method for this message takes a string and four numbers as arguments. The first two numbers are the coordinates of a point. The third should be 0, 1, or 2, indicating whether the string should be left justified, centered, or right justified on the point, respectively. The final argument should be 0 or 1, indicating whether the string should be drawn above or below the point. The message :draw-text-up is analogous but rotated by 90 degrees.

The :draw-symbol message draws a symbol at a specified point. The symbol is specified using two arguments, a symbol name and a flag that is true if the symbol should be highlighted and nil if not. The symbol name should be one of the Lisp symbols in the list returned by the function plot-symbol-symbols. This list should include at least the symbols dot, dot1, dot2, dot3, dot4, disk, diamond, cross, square, wedge1, wedge2, and x. For example, the expressions

```
(send w :draw-symbol 'disk nil 100 100)
```

and

```
(send w :draw-symbol 'disk t 100 120)
```

draw the normal and highlighted versions of the standard point symbol used in the plots described in Chapter 2.

The message `:replace-symbol` replaces one symbol by another at the specified location. To replace the normal disk by a highlighted disk at the point (100, 100), you can use the expression

```
(send w :replace-symbol 'disk nil 'disk t 100 100)
```

This message checks whether any of the old symbol needs to be erased before replacing it with the new symbol.

Bitmaps are drawn with the `:draw-bitmap` message. The method for this message requires three arguments: the bitmap and two integers representing the coordinates at the point where the left top corner of the bitmap is to be placed. The bitmap itself should be a matrix of zeros and ones. For example,

```
(send w :draw-bitmap
      '#2a((0 1 0) (1 1 1) (0 1 0)) 100 140)
```

draws a small cross in the window. The `:draw-bitmap` message can also be given a fourth argument, another bitmap matrix to be used as a mask. This matrix should have the same dimensions as the image bitmap. The default mask is a matrix of ones.

Exercise 8.1

Write a method for a graphics window that draws a circle with an inscribed square. The method should take the coordinates of the center and the radius as arguments.

8.3 Double Buffering and Animation

We can now start to look at techniques for moving brushes across a plot or rotating a cloud of points. These techniques are called *animation techniques*. Two animation techniques are available: XOR drawing and double buffering.

Let's start by looking at XOR drawing. Taking our window `w` which was set up earlier, the expression

```
(let ((width (send w :canvas-width))
      (height (send w :canvas-height))
      (mode (send w :draw-mode)))
  (send w :draw-mode 'xor)
  (dotimes (i (min width height))
    (send w :draw-symbol 'disk t i i)
    (send w :draw-symbol 'disk t i i))
  (send w :draw-mode mode))
```

moves a highlighted symbol down the diagonal of the window. The bindings in the `let` statement determine the current canvas size and save the current drawing mode. The first expression in the body sets the drawing mode to

XOR; this remains in effect until it is reset at the end of the body. The remaining statement is a `dotimes` loop. The body of the loop contains two identical drawing expressions. The first draws the symbol. The second draws it again. Since the drawing system is in XOR mode, the effect of the second drawing command is to erase the symbol.

On some systems the animation in this example may move too quickly to be visible. If this is the case, you can insert a `pause` expression, such as

```
(pause 2)
```

between the two drawing commands in the loop.

XOR drawing is useful for moving simple objects across a plain background. It does, however, have several drawbacks that reduce its usefulness as a general animation technique:

- If the background contains many items, the XOR inversion process distorts the object being drawn as it moves across the items in the background.
- XOR drawing inherently involves a certain amount of flicker. The symbol in our example spends as much time in an erased state as it spends on the screen.
- Moving several objects results in some distortions, since only one object can be moved at a time.
- The notion of inverting the background pixels used in XOR drawing is not well-defined if the background color is not black or white.

Nevertheless, XOR drawing is still a useful technique, even on color displays, since the property of being its own inverse is preserved. The plots in Lisp-Stat use XOR drawing for two purposes: moving the brush and drawing point labels.

A second animation method is *double buffering*. In double buffering, an image is composed off screen and is then copied onto the screen. As long as the copying process is sufficiently fast, this produces a flicker-free transition from one image to the next. To support double buffering, the graphics system provides a single background buffer. You can think of the buffer as just another window that is located somewhere off the screen. By sending a window the `:start-buffering` message, you instruct the window to forward all its drawing messages to the buffer. The `:start-buffering` method also makes sure that the state of the buffer's drawing system is the same as the window's state when buffering starts. State change instructions given to the window while buffering is turned on affect both the window and the buffer. Drawing operations only affect the buffer. When you have finished drawing your picture in the buffer, you can copy the picture to the window and turn the buffering off by sending the window the `:buffer-to-screen` message.

Here is an expression to move a symbol down the window diagonal using double buffering:

```
(let ((width (send w :canvas-width))
      (height (send w :canvas-height)))
  (dotimes (i (min width height))
    (send w :start-buffering)
    (send w :erase-window)
    (send w :draw-symbol 'disk t i i)
    (send w :buffer-to-screen)))
```

The `:erase` message is needed to clear the previous symbol from the buffer. Using double buffering, the symbol moves more slowly, but it appears to move more smoothly because there is virtually no flicker.

The `:buffer-to-screen` message can also be given the coordinates of a rectangle as arguments. It then copies only the specified rectangle from the buffer to the screen. This is useful, for example, if you want to change the contents of a plot without changing a set of axes.

If you send the `:start-buffering` message twice in a row, without copying the buffer to the screen in between, the effect is to increase a buffer count. Sending the `:buffer-to-screen` message decrements the count and only copies the buffer to the screen when the count reaches zero. This is useful if you are writing a method in terms of another method that may or may not already make use of the buffer. For example, the standard scatterplot can be instructed to redraw the entire plot with the `:redraw` message, or just the content of the plot with the `:redraw-content` message. Both use the buffer, and the `:redraw` method is defined in terms of the `:redraw-content` message and code to redraw the axes.

While you are debugging a graphics algorithm, you may put yourself into a state where you don't know if a window is buffering or not. The function `reset-graphics-buffer` turns off all buffering and sets the buffering count to zero. Returning to the top level after an error or an interrupt also resets the buffer.

Only one window should use the buffer at a time.

Exercise 8.2

Write a method that takes a circle with an inscribed square and rotates it smoothly through 360 degrees. Try this with XOR drawing and double buffering.

8.4 Responding to Events

Up to now, we have controlled our graphics window directly from the interpreter. The plots described in Chapter 2, on the other hand, are able to respond on their own to mouse actions, the need to be redrawn or resized,

and so on. To allow a window to respond to these user-generated events, the graphics system sends each window object a message when an event occurs that requires a response from the window.

Events generated by resizing or exposing a window are clearly associated with one particular window. Events generated by mouse or keyboard actions, on the other hand, could be assigned to windows according to several different rules. The window receiving such events is called the *focus window*. Different user interfaces follow different conventions for determining the focus window. Some interfaces assume that the focus window is the window containing the cursor. Others require that a window be designated explicitly as the current focus window by, for example, clicking on the window. Lisp-Stat leaves the rules for determining the focus window to the native window system. The interface to the window system is responsible for sending messages corresponding to user actions to the proper window object.

8.4.1 Resize and Exposure Events

After a window is resized by the user, the system first sends the window object the `:resize` message. Then the window is sent the `:redraw` message. The `:redraw` message is also sent after an obscured part of a window is exposed. By responding to these messages, the window can maintain its picture while it is resized and other windows are raised, lowered, and moved around it.

As an example, let's take the window created earlier and try to have it show a highlighted symbol located at its center. We could do this using only the `:redraw` message, but to get some practice let's use both the `:redraw` and `:resize` messages. First, we can add two slots to hold the coordinates of the symbol location,

```
(send w :add-slot 'x (/ (send w :canvas-width) 2))
(send w :add-slot 'y (/ (send w :canvas-height) 2))
```

and define accessors for the slots as

```
(defmeth w :x (&optional (val nil set))
  (if set (setf (slot-value 'x) val))
  (slot-value 'x))
```

and

```
(defmeth w :y (&optional (val nil set))
  (if set (setf (slot-value 'y) val))
  (slot-value 'y))
```

Next, we can define a `:resize` message that updates these slots whenever the window is resized:

```
(defmeth w :resize ()
  (send self :x (/ (send self :canvas-width) 2))
  (send self :y (/ (send self :canvas-height) 2)))
```

Finally, the `:redraw` method uses the information in these slots to draw the symbol:

```
(defmeth w :redraw ()
  (let ((x (round (send self :x)))
        (y (round (send self :y))))
    (send self :erase-window)
    (send self :draw-symbol 'disk t x y)))
```

The `round` function is needed, since the coordinate arguments to `:draw-symbol` must be integers.

This is the basic approach used by all the standard plots: when the window is resized, basic information about the window's size is determined and saved. The `redraw` message then retrieves this information when it needs it.

When a new window is created, it is first sent a `:resize` and then a `:redraw` message. You can therefore rely on having a `:resize` call before the first time the window is drawn.

Exercise 8.3

Write `:resize` and `:redraw` methods for the circle and square of Exercise 8.2 that keep the circle at the center of the window when the window is resized. The `:redraw` method should be able to draw the square at different angles. The current angle can be stored in a slot in the window object.

8.4.2 Mouse Events

There are two types of mouse events: *motion events* and *click events*. When the mouse is moved and a graphics window is the focus window, the graphics system sends the window the `:do-motion` message with two integer arguments, the *x* and *y* coordinates of the new mouse location. For our example, we might like to define a `:do-motion` method that has the symbol follow the location of the mouse as it is moved. Such a method can be defined as

```
(defmeth w :do-motion (x y)
  (send self :x x)
  (send self :y y)
  (send self :redraw))
```

Instead of adjusting the symbol every time the mouse is moved, it may be preferable to adjust it only in response to a mouse click. This can be done by first removing the `:do-motion` method using the expression

```
(send w :delete-method :do-motion)
```

and then defining a `:do-click` method. Whenever a mouse click occurs, the system sends the focus window the `:do-click` message with four arguments.

The first two arguments are integers, the *x* and *y* coordinates of the location of the click. The remaining two arguments are either *t* or *nil*, to indicate the states of two modifiers. The first modifier is the *extend modifier*, and the second is the *option modifier*. The method used to signal these modifiers depends on the particular user interface. On the Macintosh they correspond to holding down the *shift* and *option* keys, respectively. For our example, a method that adjusts the symbol in response to a click can be defined as

```
(defmeth w :do-click (x y m1 m2)
  (send self :x x)
  (send self :y y)
  (send self :redraw))
```

We have just seen two different approaches for positioning an object in a window. The first is to track the mouse as it moves, and the second is to move the object to the location of a mouse click. In many cases it is useful to combine these two approaches by allowing the object to be moved continuously as the mouse is *dragged* (i.e., moved while holding down the mouse button). To allow an action to be performed while the button is held down, you can send the window object the *:while-button-down* message from the body of the *:do-click* message. The *:while-button-down* message requires one argument, a function. It also accepts an additional optional argument. If the optional argument is *t*, the default, then, while the mouse button remains down, the function argument is called each time the mouse is moved. If the optional argument is *nil*, the function is called continuously, until the mouse button is released. The function should take two arguments, the *x* and *y* coordinates of the current mouse position. The method for the *:while-button-down* message returns when the mouse button is released.

To incorporate dragging into our example, we can redefine the *:do-click* method as

```
(defmeth w :do-click (x y m1 m2)
  (flet ((set-symbol (x y)
                (send self :x x)
                (send self :y y)
                (send self :redraw)))
    (set-symbol x y)
    (send self :while-button-down #'set-symbol)))
```

The initial call to the local function *set-symbol* ensures that the initial click moves the symbol to the location of the click.

Since the *:while-button-down* message in this definition is sent without the optional argument, the *set-symbol* function is only called when the mouse is moved with the button down. If the last expression is replaced by

```
(send self :while-button-down #'set-symbol nil)
```

then the `set-symbol` function is called even when the mouse is not moved. On many systems this results in the symbol flickering rapidly while the mouse is held down. This ability to call a function continuously is useful for implementing a button that causes an action to occur while it is pressed. The rotation control buttons on the Lisp-Stat rotating plot use this approach.

Instead of dragging an entire object, it is sometimes better to drag a rectangular outline of the object. The message `:drag-grey-rect` uses XOR drawing to drag a rectangle over the window. It requires four arguments: the *x* and *y* coordinates of the current mouse position, and the width and height of the rectangle. It draws a dashed rectangle and moves it in response to mouse motions until the mouse is released. When the button is released, the method removes the dashed rectangle from the screen, and returns a list of the coordinates of the final rectangle. To use this message in our example, we can define the `:do-click` method as

```
(defmeth w :do-click (x y m1 m2)
  (let ((xy (send self :drag-grey-rect x y 5 5)))
    (send self :x (+ 3 (first xy)))
    (send self :y (+ 3 (second xy)))
    (send self :redraw)))
```

This definition uses a square that is five pixels wide. Adding 3 to each coordinate centers the symbol within the rectangle.

By default, the `:drag-grey-rect` method puts the mouse cursor at the right bottom corner of the rectangle. You can supply two additional integer arguments to the method that represent an offset, the number of pixels to shift the rectangle to the right and down relative to the mouse. Thus

```
(send self :drag-grey-rect x y 5 5 3 3)
```

would approximately center the cursor within the rectangle.

We can add one more variation to our example. Redefining the `:do-click` method as

```
(defmeth w :do-click (x y m1 m2)
  (let ((cursor (send self :cursor)))
    (send self :cursor 'finger)
    (let ((xy (send self :drag-grey-rect x y 5 5)))
      (send self :x (+ 3 (first xy)))
      (send self :y (+ 3 (second xy)))
      (send self :redraw))
    (send self :cursor cursor)))
```

causes the cursor to change to a hand with a pointing finger while the symbol is being dragged. The list of available cursors is returned by the function `cursor-symbols`. It should include `arrow`, `brush`, `hand`, and `finger`. Each graph window maintains a cursor that is used whenever the window is the

focus window and the mouse is over the window. Changing the cursor is not really necessary in this simple example. But if a plot can be in several different modes, then it is important to give the user a visual cue of the current mode. All standard Lisp-Stat plots can be in one of two modes, brushing and selecting. In addition, several examples in Chapter 9 show how to add new modes to a plot. With several available modes it is easy to become confused. Using a different cursor for each mode eliminates this confusion.

Exercise 8.4

Write a :do-click method for the circle and square of Exercise 8.3 that allows you to change the center of the circle by dragging it.

Exercise 8.5

Modify the method of Exercise 8.4 to continuously increase the radius of the circle by one pixel at a time if the mouse is clicked and held down outside of the circle. This requires using the optional argument to :while-button-down.

8.4.3 Key Events

When a key is hit and a graphics window is the focus window, then the window object is sent the :do-key message with three arguments. The first argument is a Lisp character corresponding to the key hit. The remaining two arguments are modifiers, usually indicating whether the *shift* key or the *option* or *control* key was pressed when the key was hit.

For our example, we can define a :do-key method that moves the symbol up, down, right or left in response to pressing the *u*, *d*, *r*, or *l* keys, respectively. Instead of fixing the step size in advance, it may be better to add a slot and a corresponding accessor method as

```
(send w :add-slot 'step-size 10)
```

and

```
(defmeth w :step-size (&optional (val nil set))
  (if set (setf (slot-value 'step-size) val))
  (slot-value 'step-size))
```

The method defined by

```
(defmeth w :move (x y)
  (send self :x (+ x (send self :x)))
  (send self :y (+ y (send self :y)))
  (send self :redraw))
```

moves the symbol by a specified amount. Using these two methods, the :do-key method can be defined as

```
(defmeth w :do-key (c m1 m2)
  (let ((step (send self :step-size)))
    (case c
      (#\u (send self :move 0 (- step)))
      (#\d (send self :move 0 step))
      (#\r (send self :move step 0))
      (#\l (send self :move (- step) 0)))))
```

We could allow both lower and upper case characters to be used by replacing the keyforms like #\u in the case clauses by lists of the form (#\u #\U).

8.4.4 Idle Actions

The Lisp-Stat rotating plots can be instructed to rotate continuously by pressing their controls with the extend modifier. This is implemented by instructing the system to enable *idling* for these plot objects. When idling is enabled for a graphics window, the system sends the window object the :do-idle message each time the system passes through its event loop.

The :idle-on message can be used to determine whether idling is enabled and to enable idling. With no arguments, it returns t if idling is enabled, and nil if it is not. With an argument, it enables idling if the argument is true, and disables idling if it is nil. To protect against runaway idle functions, idling is automatically disabled if an error occurs in an idle action.

To continue our example, we can use the :do-idle method to have the symbol in our window move as a random walk, selecting a step up, down, to the right or the left with probability 1/4 on each call. Using the methods for :step-size and :move just defined, the :do-idle method can be written as

```
(defmeth w :do-idle ()
  (let ((step (send self :step-size)))
    (case (random 4)
      (0 (send self :move 0 (- step)))
      (1 (send self :move 0 step))
      (2 (send self :move step 0))
      (3 (send self :move (- step) 0)))))
```

The function random takes an integer n , and returns a random integer selected uniformly from $0, \dots, n - 1$. The expression

```
(send w :idle-on t)
```

starts the random walk, and

```
(send w :idle-on nil)
```

stops the walk again.

Exercise 8.6

The random walk in the example will eventually move out of the window. Modify one or more of the methods to ensure that the walk is forced to remain in the window.

Exercise 8.7

Write a `:do-idle` method for the circle and square of Exercise 8.5 that causes the circle to rotate continuously.

8.4.5 Menus

Each graphics window provides support for associating a menu with itself. The particular action that causes the menu to be presented depends on the particular window system and user interface. In the Macintosh version of XLISP-STAT, the menu for a graphics window is installed in the menu bar when the window is the active window. In the *X11* version, the menu is popped up by pressing a menu button at the top of the window.

To associate a menu with a graph window, you can send the window object the `:menu` message with the menu as its argument. A `nil` argument removes the current menu from the window. Without an argument, the message returns the current menu for the window.

For our window `w`, which now implements a simple random walk simulation, it might be useful to have a menu that restarts the walk from the center of the window and allows the walk to be turned on and off. A method for restarting the window can be defined as

```
(defmeth w :restart ()
  (send self :x (/ (send self :canvas-width) 2))
  (send self :y (/ (send self :canvas-height) 2)))
  (send self :redraw))
```

and a menu object for sending the message is given by

```
(setf restart-item
  (send menu-item-proto :new "Restart"
    :action #'(lambda () (send w :restart))))
```

A menu item for starting and stopping the walk is given by

```
(setf run-item
  (send menu-item-proto :new "Run"
    :action #'(lambda ()
      (send w :idle-on (not (send w :idle-on))))))
```

This item toggles idling on and off. The update method

```
(defmeth run-item :update ()
  (send self :mark (send w :idle-on)))
```

ensures that the item contains a check mark when the walk is turned on. Finally, the expressions

```
(setf menu (send menu-proto :new "Random Walk"))
(send menu :append-items restart-item run-item)
(send w :menu menu)
```

construct and install the menu.

It is also possible to pop up a menu in response to a mouse click by sending the menu the :popup message. The method for this message requires two integer arguments representing the coordinates of the point at which the menu is to be displayed. By default, these coordinates are interpreted relative to the screen origin. But if a window object is supplied as an additional optional argument, then the coordinates are taken to be relative to the left top corner of the window's content. Thus

```
(defmeth w :do-click (x y m1 m2)
  (send (send self :menu) :popup x y self))
```

defines a :do-click method that pops up the window menu at the point where the click occurred.

8.5 Additional Features

8.5.1 Canvas Dimensions

A window's canvas is a rectangular area with the origin of the coordinate system at its top left corner. The width and height of the canvas can be fixed or elastic. If they are elastic, they are equal to the width or height of the window's content. This is the default for a new graphics window. To give a window canvas a fixed width, you can send the window the :has-h-scroll message with an argument that is either t or a positive integer. The integer is used as the fixed width of the canvas, and a scroll bar is installed.⁴ If the argument is t, then the width is set equal to the maximum of the screen's width and height, as determined by the screen-size function. The canvas' height can be made fixed using the :has-v-scroll message. Both :has-h-scroll and :has-v-scroll can be given a nil argument to make the width or height elastic. Called with no argument, these messages return t if the dimension is currently fixed and nil if it is elastic. The canvas width and height can also be specified with the :has-h-scroll and :has-v-scroll keywords to the :isnew method for the graphics window prototype.

⁴Most systems impose a bound on the maximal size of a canvas dimension. The bound is at least $2^{15} - 1$.

When a dimension is fixed, the plot window contains a scroll bar for that dimension. These scroll bars can be used to position the window within the canvas. You can also position the window by sending the window object the :scroll message with two arguments, the coordinates of the window's left top corner in the canvas coordinate system. The :scroll message can also be sent with no arguments to obtain a list of the current coordinates of this corner. The coordinates of the current window rectangle can be obtained using the :view-rect message.

In most user interfaces scroll bars allow two forms of scrolling, one for scrolling a line at a time and one for scrolling a page at a time. By default, line scrolling scrolls one pixel at a time and page scrolling scrolls five pixels at a time. You can change these values by using the :h-scroll-incs and :v-scroll-incs messages. Both take either no arguments or two integer arguments, the line and page increments. With no arguments, they return a list of the current increments.

Fixed canvases can be useful for examining a scatterplot or a scatterplot matrix on a small screen. They are not used in the default settings of the standard plots, except in the `name-list` plot. But all plots allow the addition of scroll bars through the **Options...** dialog in the plot menus.

8.5.2 Clipping

It is sometimes useful to be able to restrict drawing to a subrectangle of a window without having to explicitly check every drawing action's arguments directly. This can be accomplished by setting a *clip rectangle*. The message :clip-rect is used to set the rectangle. Its arguments can be the four coordinates of the rectangle, or `nil` to turn clipping off. It returns a list of the current clipping rectangle's coordinates or `nil`. It can also be called with no coordinates to return the current clipping state without changing it.

If a clip rectangle is set, it is not automatically resized when the window is resized. You have to adjust it yourself in a :resize method.

8.5.3 Saving Graphs

Lisp-Stat implementations should provide a way of saving the image in a plot to a file. In the Macintosh version of XLISP-STAT the **Copy** command in the **Edit** menu can be used. It sends the window the :copy-to-clip message, which is intended to copy the image to the clip board. In the *SunView* and *X11* versions the :save-image message can be sent to the window object. The methods for these messages may use the :redraw method of the window to construct the image to be saved, and may thus not work properly if the :redraw method does not reconstruct the current image.

8.5.4 Adding New Colors

Colors available to the Lisp-Stat system are identified by symbols, such as `red` and `green`. A color system typically supports at least the colors `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta`, and `yellow`. You can get a list of the available colors by using the function `color-symbols`. You can also ask the system to make a new color available by using the function `make-color`. This function takes four arguments, a symbol naming the new color and three real numbers from the range $[0, 1]$ representing the red, green, and blue contributions in the color's RGB representation.

On some of the color workstations available today, colors are a scarce resource. The number of different colors that can be displayed at different times is very large, perhaps several million, but the number of colors that can be displayed at one time is often only 256, or perhaps even as low as 16. As a result, the system may not be able to allocate a color you request. In this case it may signal an error, or it may allocate a color that is slightly different from the one requested.

When you no longer need a color that you have allocated with `make-color`, you can release it with `free-color`. This function takes a single argument, the symbol of the color to be released. Attempting to use a color after it has been released is an error. You can redefine a color with `make-color` without first freeing the previously defined color.

8.5.5 Adding New Cursors

You can also add new cursors to the ones provided in Lisp-Stat. The `make-cursor` function requires two arguments and has three optional arguments. The first argument is a symbol used to name the cursor. The next two arguments should be bitmaps, two-dimensional arrays of zeros and ones, of equal size. The first bitmap is the cursor image; the second is the cursor mask. Each one in the image is drawn in a foreground color, usually black, and each zero is drawn in a background color, usually white, provided the corresponding elements have ones in the mask bitmap. Pixels with zeros in the mask are not affected. If no mask is supplied, the mask is taken to be all ones. The final two optional arguments are integers describing the coordinates of the cursor's *hot spot*, the point in the image associated with the position of the mouse. The default hot spot is the left top corner.

Most workstations in use today use cursors constructed from 16×16 bitmaps. Some workstations may allow cursors of other sizes. The function `best-cursor-size` returns a list of the width and height for the system's preferred cursor size. This function can also be given two integer arguments, and will then return the best cursor size close to the width and height represented by these integers. The system may truncate or expand a supplied cursor that is not the optimal size.

The function `cursor-symbols` returns a list of the available cursors.

free-cursor takes a cursor symbol and releases the associated cursor. Attempting to use a cursor after it has been released is an error. You can redefine a cursor with **make-cursor** without first freeing the previously defined cursor.

8.6 An Example

As an illustration, we can use the graphics window prototype as the basis for a simple bitmap constructor. The idea is to represent the bitmap by black and white squares or rectangles, black for ones and white for zeros. Clicking the mouse in a square changes the corresponding entry in the bitmap array from zero to one, or from one to zero, and updates the screen.

To start off, lets define a prototype containing a slot for the bitmap and two slots for holding the horizontal and vertical coordinates of the rectangle grid to be placed on the window.

```
(defproto bitmap-edit-proto
  '(bitmap h v) nil graph-window-proto)
```

The **:isnew** method should require two integer arguments, the width and height of the bitmap:

```
(defmeth bitmap-edit-proto :isnew (width height)
  (call-next-method)
  (setf (slot-value 'bitmap)
    (make-array (list height width) :initial-element 0)))
```

We also need readers for the three slots, defined by

```
(defmeth bitmap-edit-proto :bitmap () (slot-value 'bitmap))
(defmeth bitmap-edit-proto :v () (slot-value 'v))
```

and

```
(defmeth bitmap-edit-proto :h () (slot-value 'h))
```

The rectangle coordinates can be set in the **:resize** method:

```
(defmeth bitmap-edit-proto :resize ()
  (let ((m (array-dimension (send self :bitmap) 0))
    (n (array-dimension (send self :bitmap) 1))
    (height (send self :canvas-height))
    (width (send self :canvas-width)))
    (setf (slot-value 'v)
      (coerce (floor (* (iseq 0 m) (/ height m)))
        'vector))
    (setf (slot-value 'h)
      (coerce (floor (* (iseq 0 n) (/ width n)))
        'vector))))
```

The coordinates are stored as vectors to allow for fast random access to their elements.

A method for drawing the rectangle for a particular bitmap pixel is given by

```
(defmeth bitmap-editproto :draw-pixel (i j)
  (let* ((b (send self :bitmap))
         (v (send self :v))
         (h (send self :h))
         (left (aref h j))
         (right (aref h (+ j 1)))
         (top (aref v i))
         (bottom (aref v (+ i 1))))
    (send self (if (= 1 (aref b i j)) :paint-rect :erase-rect)
          left top (- right left) (- bottom top))))
```

The if expression in this method selects the appropriate message selector symbol to use, :paint-rect for drawing the pixel and :erase-rect for erasing it. The :redraw method can then just erase the window and send the :draw-pixel message for each element of the bitmap array:

```
(defmeth bitmap-editproto :redraw ()
  (let* ((b (send self :bitmap))
         (m (array-dimension b 0))
         (n (array-dimension b 1))
         (width (send self :canvas-width))
         (height (send self :canvas-height)))
    (send self :start-buffering)
    (send self :erase-rect 0 0 width height)
    (dotimes (i m)
      (dotimes (j n)
        (send self :draw-pixel i j)))
    (send self :buffer-to-screen)))
```

To support the mouse click method, we can define a method that takes a pair of coordinates, locates the corresponding pixel, reverses its value in the bitmap array, and redraws its image:

```
(defmeth bitmap-editproto :set-pixel (x y)
  (let* ((b (send self :bitmap))
         (m (array-dimension b 0))
         (n (array-dimension b 1))
         (width (send self :canvas-width))
         (height (send self :canvas-height))
         (i (min (floor (* y (/ m height))) (- m 1)))
         (j (min (floor (* x (/ n width))) (- n 1))))
    (setf (aref b i j) (if (= (aref b i j) 1) 0 1))
    (send self :draw-pixel i j)))
```

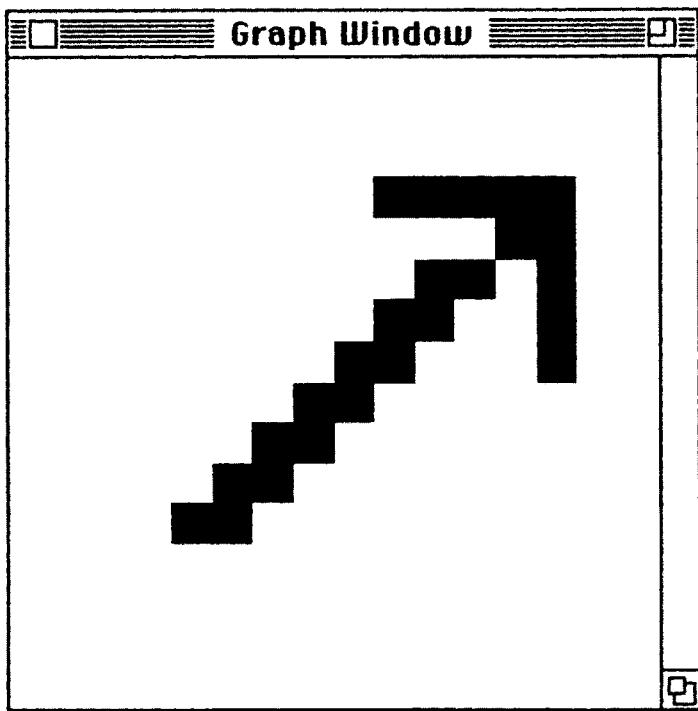


Figure 8.4: A simple bitmap editor.

The `:do-click` method is then simply

```
(defmeth bitmap-editproto :do-click (x y m1 m2)
  (send self :set-pixel x y))
```

We can now construct a bitmap editor for a 16×16 bitmap as

```
(setf w (send bitmap-editproto :new 16 16))
```

After a few mouse clicks the result might look like Figure 8.4.

Numerous variations on this example are possible. We could add a method for assigning the bitmap to a global variable:

```
(defmeth bitmap-editproto :name-bitmap ()
  (let ((str (get-string-dialog "Symbol for the bitmap:")))
    (if str
        (let ((name (with-input-from-string (s str) (read s))))
          (setf (symbol-value name) (send self :bitmap))))))
```

If the **OK** button is clicked and `get-string-dialog` returns a string, then `with-input-from-string` allows the `read` function to extract the first entry

using the standard reading conventions. In particular, this converts the name into upper case letters before constructing the symbol.

Another useful method might be one that installs the current bitmap as the window's cursor:

```
(defmeth bitmap-edit-proto :bitmap-as-cursor (yes)
  (if yes (make-cursor 'temp-cursor (send self :bitmap)))
  (send self :cursor (if yes 'temp-cursor 'arrow)))
```

Both of these methods will be easier to use if we provide access to them through a menu. We can define a menu for our window as

```
(setf bitmenu (send menu-proto :new "Bitmap"))
```

and two items by

```
(setf name-item
  (send menu-item-proto :new "Name Bitmap..."
    :action #'(lambda () (send w :name-bitmap))))
```

and

```
(setf cursor-item
  (send menu-item-proto :new "Use as Cursor"
    :action
    #'(lambda ()
      (let ((mark (send cursor-item :mark)))
        (send w :bitmap-as-cursor (not mark))
        (send cursor-item :mark (not mark)))))
```

The action function for `cursor-item` uses the item's check mark to keep track of whether the bitmap or the arrow cursor is currently in use. The expressions

```
(send bitmenu :append-items name-item cursor-item)
```

and

```
(send w :menu bitmenu)
```

install the items in the menu and the menu in our window.

Exercise 8.8

Modify this example to also display the bitmap in its actual size in the window, say, in the left top corner of the window.

Exercise 8.9

Modify this example to change bits as the mouse is dragged.

Chapter 9

Statistical Graphics Windows

In addition to the basic drawing tools provided by the graph window prototype outlined in the previous chapter, the statistical graphs in Lisp-Stat need a set of tools for managing data and translating that data into images on the screen. These tools are provided by the graph prototype `graph-proto`. More specialized graphs, such as histograms and scatterplot matrices, are based on prototypes that inherit from `graph-proto`. The first section of this chapter describes the graph prototype, and the second section outlines the more specialized prototypes. Examples illustrating how to develop new types of graphs from these prototypes are presented in the following chapter.

9.1 The Graph Prototype

The prototype `graph-proto` implements a scatterplot for displaying two-dimensional views of points and lines in m -dimensional space. The views are constructed by first centering and scaling the data, then applying a linear transformation, such as a rotation, and finally producing a scatterplot of two of the dimensions in the image of the transformation. The `:resize` and `:redraw` methods for the prototype ensure that the image is properly redrawn when the window is exposed or resized. The click and motion methods support the notion that the plot can be used in the selecting or brushing modes described in Section 2.5. The prototype also provides a basic menu for interacting with the plot.

To give a detailed illustration of the facilities provided by this prototype, this section uses a plot for examining the stack loss data introduced in Section 5.6.2.

9.1.1 Constructing a New Graph

The prototype `graph-proto` inherits from the prototype `graph-window-proto`. The `:isnew` method for `graph-proto` requires one argument, an integer m representing the dimension of the space to be viewed. The stack loss data consists of four variables, airflow (`air`), temperature (`temp`), concentration (`conc`), and ammonia loss (`loss`). A graph for viewing these data can be constructed as

```
(setf w (send graph-proto :new 4))
```

The number of variables in a graph can be retrieved with the `:num-variables` message,

```
> (send w :num-variables)
4
```

but it can not be changed once the graph is created.

A graph includes a label string for each dimension. These strings can be set and retrieved with the `:variable-label` message. Initially, these strings are empty:

```
> (send w :variable-label 0)
""
```

They can be changed by supplying a string as a second argument:

```
> (send w :variable-label 0 "Air")
"Air"
```

The method for this message is vectorized. Labels for the remaining three dimensions can thus be specified with the expression

```
(send w :variable-label '(1 2 3) (list "Temp." "Conc." "Loss"))
```

and all four labels can be retrieved by

```
> (send w :variable-label '(0 1 2 3))
("Air" "Temp." "Conc." "Loss")
```

The `:isnew` method for `graph-proto` accepts all keyword arguments accepted by the `:isnew` method for `graph-window-proto`. In addition, the keyword `:variable-labels` can be used to specify a list of m initial variable label strings. Another keyword argument, `:scale-type`, is described below in Section 9.1.3.

9.1.2 Adding Data and Axes

Points

A graph can contain two kinds of data: points and lines. Points consist of locations in the m -dimensional space, together with additional information, such as the color and symbol used to draw the point. Initially, a graph contains no points:

```
> (send w :num-points)  
0
```

Points can be added using the `:add-points` message. The method for this message requires one argument, a list of m lists representing the coordinates of the points to be added. The expression

```
(send w :add-points (list air temp conc loss))
```

adds the stack loss data to the graph. The data set contains 21 points:

```
> (send w :num-points)  
21
```

The `:add-points` method draws the new points on the screen unless the `:draw` keyword is supplied with value `nil`. The method also allows a list of point label strings to be supplied with the keyword `:point-labels`.

Even though the plot now contains the data, it will not yet show any points in its window. The reason is that the plot initially views a range consisting of the unit interval in each variable. To adjust the range viewed by the plot to the data, you can send the plot the `:adjust-to-data` message. You can send this message using the expression

```
(send w :adjust-to-data)
```

or by selecting the **Rescale Plot** item from the plot menu. The method for this message adjusts the range viewed by the plot to correspond exactly to the range spanned by the data. After sending this message, the plot should show a scatterplot of the first two variables in the data set. The resulting plot is shown in Figure 9.1.

When new points are added to a plot, each is given a default symbol, color, and label. For the first point,

```
> (send w :point-symbol 0)  
DISK  
> (send w :point-color 0)  
NIL  
> (send w :point-label 0)  
"0"
```

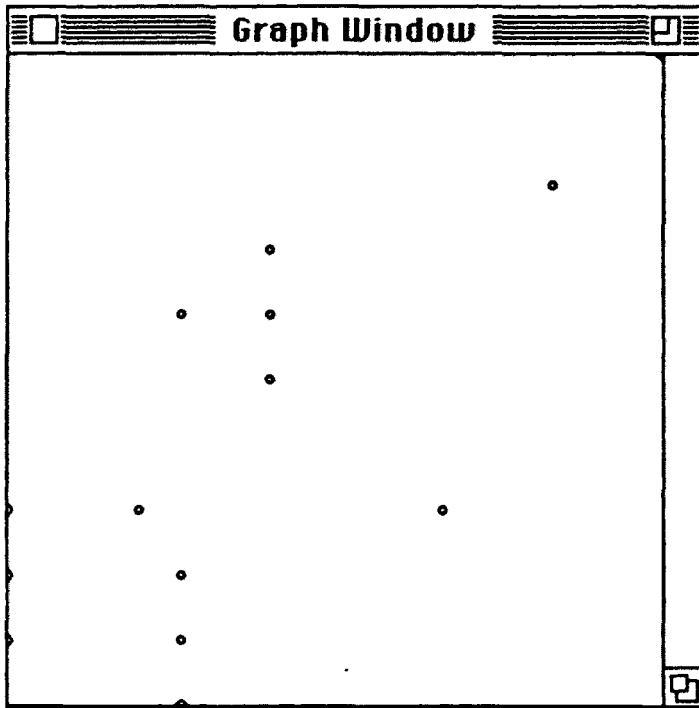


Figure 9.1: Plot of the airflow and temperature variables in the stack loss data set.

The default symbol is the symbol named `disk`. The default color is `nil`, which means the point is drawn with the current drawing color of the window. The default label is a string of the point index. You can specify new values for these properties by giving a second argument to these messages. Symbols should be taken from the list returned by the function `plot-symbol-symbols`. Colors should be `nil` or one of the values in the list returned by `color-symbols`. The methods for `:point-symbol`, `:point-color`, and `:point-label` are vectorized. Thus

```
(send w :point-symbol (iseq 21) 'diamond)
```

sets the symbol for all 21 points to `diamond`. None of these three messages causes the plot to be redrawn; to see the effect of the changes on the screen, you have to send the plot a `redraw` message.

Each point also has a state value that can be set and retrieved using the `:point-state` message. The state values can be one of the four symbols `invisible`, `normal`, `hilited`, and `selected`. Point states are used as part of the linking mechanism and are described in detail in Section 9.1.5.

The `:point-coordinate` message can be used to retrieve and set the value of an individual variable coordinate for a particular point. The method for this message requires two arguments, the variable index and the point index. Thus the airflow, temperature, concentration, and ammonia loss values for the first observation are

```
> (send w :point-coordinate 0 0)
80
> (send w :point-coordinate 1 0)
27
> (send w :point-coordinate 2 0)
89
> (send w :point-coordinate 3 0)
42
```

The method for the `:point-coordinate` message is vectorized as well. A new coordinate value can be specified as a third argument. Once again, the method does not redraw the plot when a new value is supplied.

Exercise 9.1

Write a method for a plot that takes one argument, a point symbol, and sets the point symbols for all points in the plot to the specified symbol.

Lines

Lines are represented as directed line segments starting at locations in m -dimensional space called *linestarts*. Each linestart includes additional information, such as the width and type of line to use in drawing the line segment, and the index of another linestart to use as the end of the segment. An index of `nil` indicates that the linestart is used only as the end of a segment starting elsewhere. Circular definitions are allowed: they cause no problems since the drawing routines only pass through the set of linestarts once.

When a plot is created it contains no linestarts:

```
> (send w :num-lines)
0
```

The stack loss data are in fact collected over time. It may be useful to represent the temporal relationship by drawing a line from the first observation to the second, from the second to the third, and so on. The message `:add-lines` adds such a series of line segments. Its method requires one argument, a list of m lists of coordinates for the linestarts. Thus the expression

```
(send w :add-lines (list air temp conc loss))
```

adds the series of connected segments for the stack loss data to our plot. The lines help to bring out a downward drift in the airflow and temperature in the first ten observations of the data set.

The method for the `:add-lines` message also allows a line type to be supplied with the `:type` keyword. The lines are drawn on the plot unless the `:draw` keyword is supplied with value `nil`.

Each linestart has a width, type, and color for drawing the line segment to the next linestart. For the first linestart, corresponding to the first data point,

```
> (send w :linestart-width 0)
1
> (send w :linestart-type 0)
SOLID
> (send w :linestart-color 0)
NIL
```

Again the methods are vectorized, and can be used to change the values of linestart properties by supplying new values as a second argument. Value changes do not cause any drawing to occur.

Each linestart also contains the index of the next linestart in the sequence, to be used as the end point in drawing a line segment. The `:add-lines` method connects the linestarts in series, so

```
> (send w :linestart-next 0)
1
> (send w :linestart-next 1)
2
```

The last linestart in the series has no next index:

```
> (send w :linestart-next 20)
NIL
```

You can remove one of the segments by setting the next segment value for the linestart at the beginning of the segment to `nil`. For example,

```
(send w :linestart-next 7 nil)
```

removes the line between the points with indices 7 and 8. To make this change visible on the screen, you can send the plot the `:redraw` message.

As for points, you can access and change the coordinates of linestarts. The first two coordinates of the first linestart, corresponding to the airflow and temperature variables, are given by

```
> (send w :linestart-coordinate 0 0)
80
> (send w :linestart-coordinate 1 0)
27
```

The method for the `:linestart-coordinate` message is vectorized and can be used to change the values of coordinates. It does not redraw the plot if coordinates are changed.

Exercise 9.2

Write a method for a message `:add-segments` that takes two lists of m coordinate lists and adds single line segments from points in the first list to the corresponding points in the second list.

Axes and Current Variables

The scatterplot implemented by the `graph-proto` prototype can be asked to show x and y coordinate axes with the `:x-axis` and `:y-axis` messages. With no arguments, these messages return the current axis state. For example,

```
> (send w :x-axis)
(NIL NIL 0)
```

The three elements of the list indicate whether the axis is showing, whether it contains a label, and the number of tick marks it uses. Sending the message with the single argument `t` redraws the plot with an axis. By default, no label and 4 tick marks are used:

```
> (send w :x-axis t)
(T NIL 4)
```

You can specify alternate choices with optional second and third arguments. The `:y-axis` method is analogous. Both methods send the plot the `:resize` and `:redraw` messages when the axis state is changed, unless the keyword argument `:draw` is supplied with value `nil`. To supply this keyword, you have to give all three optional arguments. After adding x and y axes, our plot appears as shown in Figure 9.2.

The `:adjust-to-data` method sets the range viewed by a graph to the range of the data. This often does not lead to nice axis labels. You can access and change the range for each variable with the `:range` message. For the airflow and temperature variables,

```
> (send w :range 0)
(50 80)
> (send w :range 1)
(17 27)
```

The elements of the result lists represent the lower and upper bounds on the ranges. To change the range of a variable, you need to supply two additional values, the new lower and upper bounds. The expression

```
(send w :range 1 15 30)
```

sets the range for the temperature variable to the interval [15, 30]. When the range of a variable is changed, the plot is redrawn unless the `:draw` keyword is supplied with value `nil`. The method for the `:range` message is vectorized.

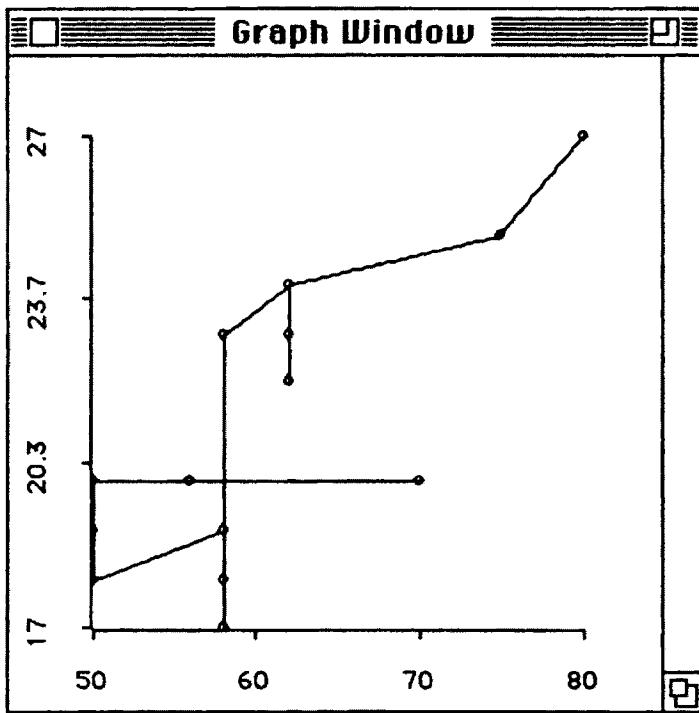


Figure 9.2: Plot with x and y axes of the airflow and temperature variables in the stack loss data set. Consecutive observations are connected by lines.

The function `get-nice-range` can be used to help find a good combination of range and tick marks. The function takes three arguments, the lower and upper end points of an interval and an integer number of tick marks. It returns a list of three values representing the end points of an interval containing the original interval, and a number of tick marks close to the number specified. The new values should produce reasonable axes. For example, for the range of the temperature variable with approximately four tick marks,

```
> (get-nice-range 17 27 4)
(16 28 7)
```

This suggests setting the range for this variable to the interval [16, 28] with

```
(send w :range 1 16 28)
```

and setting the y axis to use seven tick marks:

```
(send w :y-axis t t 7)
```

The resulting axis marks are 16, 18, 20, ..., 28.

Up to now, our plot has shown only the first two variables from the four variable data set. The message `:current-variables` can be used to set and retrieve the two variables currently used to construct the plot. The default is to show the first two variables:

```
> (send w :current-variables)  
(0 1)
```

The expression

```
(send w :current-variables 2 3)
```

instructs the plot to switch to using the other two variables. The method sends the plot the `:redraw` message unless the `:draw` keyword is supplied with value `nil`.

Exercise 9.3

Write an `:adjust-to-data` method that sets the range and axes to nice values for the current variables.

Clearing Plot Data

The data currently in a plot can be cleared using the messages `:clear`, `:clear-points`, and `:clear-lines`. The `:clear-points` message removes the internal point data and sets the number of points to zero. It redraws the plot unless the `:draw` keyword is supplied with value `nil`. The `:clear-lines` message for clearing linestart data is analogous. The `:clear` message removes both points and lines.

These messages are particularly useful for animations in which the data shown in a plot are changed rapidly. Since the redraw methods described below in Section 9.1.6 use double buffering, this leads to a smooth animation.

9.1.3 Scaling and Transformations

A major feature of the `graph-proto` prototype is its ability to allow linear transformations, in particular rotations, to be applied to the data. The transformation process can be decomposed into two stages. The first stage consists of centering and scaling the data. The second consists of applying a transformation matrix to the scaled and centered data. The plot then shows a scatterplot of two of the coordinates of the result of the transformation.¹

¹This decomposition is motivated by the viewing pipeline model of Buja et al. [16].

Scaling and Centering

The scaling and centering stage introduces a new coordinate system, called the *scaled coordinate system*. For reference, the coordinate system in which the data are originally specified is called the *real coordinate system*. An arbitrary scaling scheme can be used, but two standard scaling schemes, called *fixed scaling* and *variable scaling*, are sufficient for most situations. These scaling types are implemented by the :adjust-to-data method and the :scale-type method. Before proceeding, we can use the expressions

```
(send w :x-axis nil)
```

and

```
(send w :y-axis nil)
```

to remove the axes from our plot. The axes are not meaningful for transformed data.

When a new plot is constructed, its scale type is `nil`:

```
> (send w :scale-type)  
NIL
```

This means there is no scaling or centering, and the viewing range in each dimension is set to the range of the data in the corresponding dimension of the real coordinate system. The range in the scaled coordinate system can be retrieved by the :scaled-range message:

```
> (send w :range 0)  
(50 80)  
> (send w :scaled-range 0)  
(50 80)
```

Since there is initially no scaling, the scaled and real ranges are identical.

To examine the new scaling types, we can start by setting the scale type for our plot to the symbol `variable` with the expression

```
(send w :scale-type 'variable)
```

The method for the :scale-type message sends the :adjust-to-data message and redraws the plot, unless the :draw keyword is supplied with value `nil`. For a plot using variable scaling, the :adjust-to-data method centers the data at the midranges for each variable, scales each centered variable to the interval $[-1, 1]$, and sets the scaled ranges to $[-\sqrt{m}, \sqrt{m}]$. This ensures that any rotation of the data will fit in the scaled range. After setting the scale type to `variable`, the range and scaled range of the first dimension of the plot are

```
> (send w :range 0)
(35 95)
> (send w :scaled-range 0)
(-2 2)
```

Variable scaling is appropriate when the variables are not measured on comparable scales. If the variables are measured on comparable scales, we may want to preserve angles in the data by using the same scale factor for each dimension. This is incorporated in the fixed scaling strategy. If the scale type is set to the symbol `fixed`, then the `:adjust-to-data` method centers the data at the midranges, chooses the scale factor 1 for each dimension, and sets the scaled range for each variable to \sqrt{m} times the maximal range of the centered variables.

If the variable and fixed scaling schemes are not adequate, you can define your own scaling scheme by defining a new `:adjust-to-data` method. This method can use the `:scaled-range` message to set a new scaled range, and it can use the `:center` and `:scale` messages to set or retrieve the centers and scale factors. For our plot, with variable scaling, the scale factor and center for the first variable are

```
> (send w :scale 0)
15
> (send w :center 0)
65
```

When the scaled range is set with the `:scaled-range` message, the range in the original coordinate system is adjusted. The messages `:scale`, `:center`, and `:scaled-range` are vectorized. When used to set new values, they redraw the plot unless they are given the `:draw` keyword with value `nil`.

Another message that is useful for defining custom scaling schemes is the `:visible-range` message. The method for this message takes a dimension index, and returns the range of all linestarts and visible points in that dimension. For example, for the first dimension of our plot

```
> (send w :visible-range 0)
(50 80)
```

The initial scale type of a plot can be specified with the `:scale-type` keyword to the `graph-proto :isnew` method.

Exercise 9.4

Modify the `:adjust-to-data` method defined in Exercise 9.3 to use the default method when the scale type of the plot is not `nil`.

Transformations

Having chosen a scaling scheme, such as variable scaling, we can now apply a transformation. Initially, there is no transformation:

```
> (send w :transformation)
NIL
```

We can apply a transformation matrix to each point and linestart in the data by giving it as an argument to the `:transformation` message. This transforms each data location, viewed as a column vector, by multiplying the vector on the left by the transformation matrix. Unless the keyword `:draw` is supplied with value `nil`, the `:transformation` method redraws the plot when it is given a new transformation. As an example, if the current variables are given by

```
> (send w :current-variables)
(0 1)
```

then we can use the expression

```
(send w :transformation
  '#2A((0 0 -1 0)
        (0 0 0 -1)
        (1 0 0 0)
        (0 1 0 0)))
```

to apply a rotation that replaces the airflow variable by concentration and temperature by ammonia loss. The expression

```
(send w :transformation nil)
```

returns the plot to the untransformed state.

Several other methods are available for applying a transformation. To make it easier to understand the correspondence between points in the original plot of temperature versus airflow and the transformed plot of ammonia loss versus concentration, it is useful to rotate smoothly from the first plot to the second. Such a rotation from one scatterplot to another is called a *plot interpolation* [16]. You can perform this rotation by setting the transformation matrix to a series of different intermediate rotations. Another approach is to use the `:apply-transformation` message. The method for this message takes an incremental transformation matrix, and constructs a new plot transformation by multiplying the current transformation on the left by the incremental transformation. By default, this method redraws the plot. Since this redraw uses the buffer, the expression

```
(let* ((c (cos (/ pi 20)))
      (s (sin (/ pi 20)))
      (m (+ (* c (identity-matrix 4))
            (* s '#2A((0 0 -1 0)
                      (0 0 0 -1)
                      (1 0 0 0)
                      (0 1 0 0))))))
  (dotimes (i 10) (send w :apply-transformation m)))
```

rotates smoothly from the original plot to the transformed plot using ten steps.

The message `:rotate-2` can be used to apply a two-dimensional rotation in a plane defined by two variable indices. A rotation angle is required as a third argument. For example, the expression

```
(send w :rotate-2 0 2 (/ pi 20))
```

rotates the dimensions 0 and 2 of the current transformed data by the angle $\pi/20$. After resetting the transformation in our plot to `nil`, we can use `:rotate-2` in the expression

```
(dotimes (i 10)
  (send w :rotate-2 0 2 (/ pi 20) :draw nil)
  (send w :rotate-2 1 3 (/ pi 20)))
```

to rotate smoothly from the temperature versus airflow plot to the ammonia loss versus concentration plot. Each step consists of two two-dimensional rotations. The plot is only redrawn by the second rotation.

The `:transformation` and `:apply-transformation` methods can also be given square matrices of dimension $k < m$ as arguments. In this case the transformations are applied to the first k dimensions of the data space. This is useful for certain prototypes, such as the histogram prototype, that inherit from `graph-proto`. The `:apply-transformation` message can also be given a sequence of the same dimension as the matrix argument with the `:basis` keyword. The sequence should contain `nil` and non-`nil` elements. The transformation system ignores elements of the transformation matrix corresponding to `nil` entries in the basis. This feature allows low-dimensional rotations to be applied efficiently to higher-dimensional plots.

Exercise 9.5

Write a method that takes two variable indices as arguments, and rotates smoothly from the current transformation to a plot showing the variables specified by the two indices. Thus calling the method with the arguments 0 and 1 should rotate to a plot of temperature versus airflow.

Accessing Transformed Data

The `graph-proto` prototype provides messages for retrieving the current value of a transformed point or linestart coordinate. For example, the first two transformed coordinates of the first point and linestart are given by

```
> (send w :point-transformed-coordinate 0 0)
-0.619048
> (send w :point-transformed-coordinate 1 0)
-1
```

and

```
> (send w :linestart-transformed-coordinate 0 0)
-0.619048
> (send w :linestart-transformed-coordinate 1 0)
-1
```

9.1.4 Mouse Events and Mouse Modes

The `:do-click` and `:do-motion` methods for the `graph-proto` prototype are defined to support the notion that a plot can be in different *mouse modes*. New modes can be added. You can switch among the available modes by using a dialog brought up by the **Mouse Mode** menu item, or by sending a message to the plot. Mouse modes are identified by a Lisp symbol. Each mouse mode also has a title string, used in the dialog for switching modes, a cursor to provide a visual cue of the current mode, and its own click and motion actions. To customize a plot or define a new prototype, it is usually not necessary to override the `:do-click` and `:do-motion` methods. It is sufficient to add new mouse modes or override messages used by the standard mouse modes.

Adding New Mouse Modes

Initially a plot has two mouse modes, `selecting` and `brushing`. The message `:mouse-modes` returns a list of the symbols identifying the available modes:

```
> (send w :mouse-modes)
(SELECTING BRUSHING)
```

The `:add-mouse-mode` message adds a new mouse mode or redefines an existing one. The message requires one argument, the mouse mode symbol, and takes several keyword arguments. These keywords include `:title`, for specifying a title string, and `:cursor`, for specifying a cursor symbol. The keywords `:click` and `:motion` can be used to specify message selectors to use for handling click and motion events when the plot is in the new mode.

As an example, we can add a new mode to our plot that simply shows the coordinates of a mouse click on the window. The mode can be added with the expression

```
(send w :add-mouse-mode 'show-coordinates
      :title "Show Coordinates"
      :click :do-show-coordinates
      :cursor 'finger)
```

After adding this mode, the plot has three modes available:

```
> (send w :mouse-modes)
(SELECTING BRUSHING SHOW-COORDINATES)
```

When the plot is in the **show-coordinates** mode, the cursor is set to **finger**. When a mouse click occurs in this mode, the plot is sent the **:do-show-coordinates** message with the four arguments to the **:do-click** message. Since no motion action is specified, motion events are ignored.

Before using the new mode we need to define the **:do-show-coordinates** method. To avoid having to redraw the plot, we can use XOR drawing to draw a string showing the location of the click while the button is held down. When the button is released, we can draw the string once more to remove the image of the string:

```
(defmeth w :do-show-coordinates (x y m1 m2)
  (let ((s (format nil "~s" (list x y)))
        (mode (send self :draw-mode)))
    (send self :draw-mode 'xor)
    (send self :draw-string s x y)
    (send self :while-button-down #'(lambda (x y) nil)
          (send self :draw-string s x y)
          (send self :draw-mode mode)))
```

The **:while-button-down** action just waits for the mouse button to be released.

We can now switch to the new mode by using the dialog provided by the plot menu, or by sending the plot the **:mouse-mode** message with the mode symbol as its argument:

```
(send w :mouse-mode 'show-coordinates)
```

Sent without an argument, this message returns the current mode symbol:

```
> (send w :mouse-mode)
SHOW-COORDINATES
```

Instead of showing the canvas coordinates of the mouse click, we can show the corresponding coordinates of the current variables in the scaled coordinate system. The message **:canvas-to-scaled** takes two integer arguments representing a point on the canvas, and returns two real numbers representing the corresponding point in the two current variables of the scaled coordinate system:

```
> (send w :canvas-to-scaled 100 150)
(-0.4 -0.4)
```

The message **:canvas-to-real** attempts to convert back to the real coordinate system, but it only works properly if the plot transformation is **nil**. To allow you to choose between showing canvas, scaled, or real coordinates we can define the **:do-show-coordinates** message to check the modifiers and show the real coordinates when the click occurs with the **extend** modifier and the scaled coordinates when the **option** modifier is used. With no modifiers, it shows the canvas coordinates:

```
(defmeth w :do-show-coordinates (x y m1 m2)
  (let* ((xy (cond (m1 (send self :canvas-to-real x y))
                  (m2 (send self :canvas-to-scaled x y))
                  (t (list x y))))
         (s (format nil "~s" xy))
         (mode (send self :draw-mode)))
    (send self :draw-mode 'xor)
    (send self :draw-string s x y)
    (send self :while-button-down #'(lambda (x y) nil)
          (send self :draw-string s x y)
          (send self :draw-mode mode)))
```

The messages :scaled-to-canvas and :real-to-canvas take two real numbers representing coordinates of the current variables in the scaled or real coordinate system, and return a list of the corresponding canvas coordinates.

Several other messages are available to help in developing new mouse modes. To illustrate some of these messages, we can construct a mouse mode for identifying points by placing a label next to a point while the mouse is clicked and held down near the point. This is an alternative to the standard option of having all highlighted or selected points show their labels. First, we can remove the mode just defined by using the expression

```
(send w :delete-mouse-mode 'show-coordinates)
```

This does not remove the click or motion messages associated with the mode. This message requires that each plot have at least one mouse mode. Next, we can specify a new mode for identifying points as

```
(send w :add-mouse-mode 'identify
      :title "Identify"
      :click :do-identify
      :cursor 'finger)
```

The click method for the new mode needs to set a tolerance for determining when a click is close enough to a point. Each plot maintains a tolerance range that can be set and retrieved with the :click-range message. This message returns a list of two integers, the width and height of the tolerance range in pixels. The default range is

```
> (send w :click-range)
(4 4)
```

This range is used by the standard selecting mode to determine the points to select when the mouse is clicked. A new click range can be specified by sending this message with two integer arguments, the width and height of the new range.

Using this range and the coordinates of a mouse click, we can send the plot the :points-in-rect message to obtain a list of the indices of all points with images falling in the specified rectangle. For example,

```
(send w :points-in-rect 100 150 10 15)
```

returns a list of the indices of all points with images located in the rectangle that is 10 pixels wide and 15 pixels high and has its left top corner at the canvas location (100, 150). If no points are contained in this rectangle, `nil` is returned.

Using these two messages, the `:do-identify` message can be defined as

```
(defmeth w :do-identify (x y m1 m2)
  (let* ((cr (send self :click-range))
         (p (first (send self :points-in-rect
                           (- x (round (/ (first cr) 2)))
                           (- y (round (/ (second cr) 2)))
                           (first cr)
                           (second cr))))))
    (if p
        (let ((mode (send self :draw-mode))
              (label (send self :point-label p)))
          (send self :draw-mode 'xor)
          (send self :draw-string label x y)
          (send self :while-button-down #'(lambda (x y) nil)
                (send self :draw-string label x y)
                (send self :draw-mode mode))))
```

The `:points-in-rect` message uses a rectangle centered at the click with width and height equal to the values specified in the click range. The function `first` returns the first index in the list returned by `:points-in-rect`, or `nil` if the list is empty. Thus the variable `p` is an integer if a point is found close to the click, or `nil` if no point is found.

To illustrate another message, `:drag-point`, we can define a mouse mode for editing our data by moving points in the plot. We can set up the new mode with

```
(send w :add-mouse-mode 'point-moving
      :title "Point Moving"
      :cursor 'hand
      :click :do-point-moving)
```

The message `:drag-point` can be used to drag a point in a plot. It takes the canvas coordinates of a mouse click and checks if a point is close to the click. If there is such a point, a grey rectangle is dragged around the window until the mouse button is released. At that time the point's coordinates are changed and the plot is redrawn, unless the keyword `:draw` is given with value `nil`. The new coordinates are calculated using the conversion messages described above. For a transformed plot `:drag-point` attempts to move the point in the plane associated with the current variables, keeping the orthogonal component fixed. This only works properly if the transformation

is orthogonal. The method for :drag-point returns the index of the dragged point, or nil if no point is close enough to the click. Using :drag-point, we can define a method for :do-point-moving as

```
(defmeth w :do-point-moving (x y m1 m2)
  (let ((p (send self :drag-point x y)))
    (if p (format t "Point ~d has been moved.~%" p))))
```

Exercise 9.6

The label drawn by the :do-identify method may be hard to read if it is drawn over one or more points. Modify the :do-identify method to allow the label to be dragged to a free area of the window to make it more readable.

Standard Mouse Modes

A plot constructed from the graph-proto prototype initially contains two modes, selecting and brushing. The selecting mode uses the arrow cursor and only needs a click method, :do-select-click. The brushing mode uses the brush cursor, and its click and motion messages are :do-brush-click and :do-brush-motion.

The click and motion methods for selecting and brushing are designed to implement the basic user interface for these mouse modes. The actual selection and highlighting operations are carried out by two additional messages, :unselect-all-points and :adjust-points-in-rect. The :unselect-all-points message takes no arguments. The :adjust-points-in-rect message requires five arguments. The first four are the integer coordinates of a rectangle, the two coordinates of the left top corner, and the width and the height. The fifth argument is one of the point state symbols selected or hilited. When a click occurs and the plot is in selecting mode, the :do-select-click method uses these messages in the following way:

- If the click does not include the extend modifier, the :unselect-all-points message is sent to unselect any currently selected points.
- Next, the :adjust-points-in-rect message is sent with the coordinates of a rectangle centered at the click and width and height specified by the current click range. The fifth argument is selected. The method for this message should select all visible points in the specified rectangle.
- While the mouse button is down, a dashed rectangle with one corner fixed at the click location is dragged out over the plot. When the button is released, the :adjust-points-in-rect message is sent with the coordinates of the final rectangle and the symbol selected as arguments.

When a plot is in brushing mode, the `:do-brush-click` method first unselects all points, unless the `extend` modifier is given. Then it sends the `:adjust-points-in-rect` message with the current brush coordinates and the symbol `selected` as arguments. It continues to send this message each time the mouse is moved with the button down. The `:do-brush-motion` method sends the `:adjust-points-in-rect` message with the current brush coordinates and the symbol `hilited` as arguments. When the fifth argument is `hilited`, the method for this message is expected to highlight all visible points in the rectangle and to unhighlight all highlighted points outside the rectangle.

The current brush location and size can be retrieved with the `:brush` message. The brush is specified by four integers, the `x` and `y` canvas coordinates of the right bottom corner, the corner attached to the mouse, and the width and height of the brush. New brush coordinates can be specified by sending the `:brush` message with four integer arguments, the new coordinates. The `:resize-brush` message provides a dialog for setting the new brush size interactively. This message is sent to a plot by the **Resize Brush** menu item in the plot menu.

The protocol of using the `:unselect-all-points` and `:adjust-points-in-rect` messages allows new plots to be developed without modifying the brushing or selecting methods themselves. For example, on a system with a color display we can modify the methods for these two messages to ensure that selected points are colored red and unselected points are colored in the drawing color. After sending the message

```
(send w :use-color t)
```

to ensure that the window is using color drawing, we can define the message for `:unselect-all-points` as

```
(defmeth w :unselect-all-points ()
  (send self :point-color (iseq (send self :num-points)) nil)
  (call-next-method))
```

This method sets the color of all points to `nil`, and then calls the method inherited from `graph-proto`. The new `:adjust-points-in-rect` method can be defined as

```
(defmeth w :adjust-points-in-rect (x y w h s)
  (if (eq s 'selected)
      (let ((p (send self :points-in-rect x y w h)))
        (if p (send self :point-color p 'red))))
  (call-next-method x y w h s))
```

The variable `p` contains a list of the indices of the points in the specified rectangle. The `:point-color` message is only sent if this list is not `nil`.

9.1.5 Linking

Points in a plot can be in several different states. The selecting and brushing operations provide an easy way to change the state of a point in a plot. To help explore the relationship among several views of a set of data, the Lisp-Stat graphics system allows plots to be linked. When two or more plots are linked, the system attempts to keep the states of points in linked plots identical.

The default Lisp-Stat linking system is based on a loose association model in which points in different linked plots are related by their indices. Thus selecting a point with index 3 in one plot selects the point with index 3 in all other linked plots. This association is loose in the sense that no other properties of the points are matched. Points in linked plots can have different labels, colors, or symbols. Other linking models are possible. One alternative is outlined in Section 10.6.

Determining Which Plots Are Linked

The system for determining which plots are linked to other plots is based on the two messages `:linked` and `:links`. When the `:links` message is sent to a plot, it should return `nil` if the plot is not linked. If the plot is linked, it should return a list containing all plots linked to the plot receiving the message. The plot itself may be an element of this list. The `:linked` message can be sent with no arguments to determine whether the plot is linked. If the plot is linked, it returns `t`. Otherwise, it returns `nil`. This message can also be given an optional argument, `t` to link the plot and `nil` to unlink it.

The `:links` and `:linked` methods for `graph-proto` maintain a single list of linked plots. This list is returned by the `:links` message for plots that are linked. The list can also be obtained by using the `linked-plots` function. When a plot is linked, the states of its points are passed on to all other linked plots, as determined by the result returned by the `:links` message.

Since the determination of linked plots is based entirely on these two messages, it is quite simple to define alternate strategies. For example, if we want to ensure that all plots related to the stack loss data are linked to one another but not to other plots, we can set up a list of stack loss plots as

```
(setf *stack-plots* nil)
```

and then give each stack plot its own `:links` and `:linked` methods, which are defined as

```
(defmeth w :links ()
  (if (member self *stack-plots*) *stack-plots*))
```

and

```
(defmeth w :linked (&optional (link nil set))
  (when set
    (setf *stack-plots*
      (if link
        (cons self *stack-plots*)
        (remove self *stack-plots*)))
    (call-next-method link))
  (call-next-method))
```

One way to avoid having to add these methods to each new plot would be to define a separate stack loss plot prototype.

Point States and Linking

Each point in a plot is in one of four possible states, corresponding to the symbols **invisible**, **normal**, **hilited**, and **selected**. The **graph-proto** drawing methods draw both highlighted and selected points with the highlighted version of a point's symbol. Points in the **normal** state are drawn with the **normal** symbol, and **invisible** points are not drawn. The state of a point can be determined and set with the **:point-state** message. For the first point in our plot of the stack loss data,

```
> (send w :point-state 0)
NORMAL
```

The method for the **:point-state** message is vectorized, so it can be used to determine the states of all points in the plot as

```
> (send w :point-state (iseq 21))
(NORMAL NORMAL NORMAL NORMAL ...)
```

To change a point's state, the **:point-state** message can be given a new state value as a second argument. The expression

```
(send w :point-state 0 'selected)
```

changes the state of the first point to **selected**. When a point's state is changed in one plot, the state of the corresponding point is changed in all linked plots. In addition, each plot is sent the **:adjust-screen-point** method with one argument, the index of the point to change. This method is responsible for changing the image of the point on the screen.

The action taken by the **:adjust-screen-point** method depends on both the new state and the previous state of a point. When the **:adjust-screen-point** message is sent, the value returned by the **:point-state** message reflects the new state. The previous state can be retrieved by sending the plot the **:last-point-state** message with the point index as its argument.

To give the best sense of correspondence between points in linked plots, the **:adjust-screen-point** method tries to redraw points as quickly as possible to reflect any changes in their states. For some state changes this is

quite simple. If the point state is changed among the `normal`, `hilited`, and `selected` values, then the `:adjust-screen-point` method uses the `:replace-symbol` method introduced in the previous chapter to change the symbol shown on the screen. Unfortunately, if the new state of a point is `invisible`, it is difficult to remove the point cleanly from the screen without redrawing the plot. Since it would be wasteful to redraw the plot each time a point is made invisible, the `:adjust-screen-point` method takes an indirect approach: it sets a flag to indicate that the plot needs to be adjusted. The `:needs-adjusting` message can be used to set and retrieve the value of this flag. After setting the state of a point to `invisible`, the value of this flag is `t`:

```
> (send w :needs-adjusting)
NIL
> (send w :point-state 0 'invisible)
INVISIBLE
> (send w :needs-adjusting)
T
```

At a convenient time the plot can then be sent the `:adjust-screen` message. The method for this message checks the flag, resets it, and redraws the plot if necessary. Thus after setting a point's state to `invisible`,

```
> (send w :adjust-screen)
NIL
> (send w :needs-adjusting)
NIL
```

The `:adjust-points-in-rect` method sends the `:adjust-screen` message to all linked plots after adjusting any point states that need to be changed.

As a simple illustration of these ideas, we can define a new `:adjust-screen-point` method for our plot. This method sets the color of all highlighted and selected points to `red`, and the color of all unhighlighted points to `nil`:

```
(defmeth w :adjust-screen-point (i)
  (let* ((state (send self :point-state i))
         (color (if (member state '(selected hilited)) 'red)))
    (send self :point-color i color))
  (call-next-method i))
```

This approach differs from the approach used earlier in which the method for `:adjust-points-in-rect` was modified. With the previous approach, points are only colored when their states are changed by brushing or selecting in the plot itself. The present approach ensures that the color of a point is adjusted even if its state is changed by selecting or brushing in a linked plot.

Exercise 9.7

If a plot contains a large data set, it may be useful to have the plot show only selected points. One way to accomplish this is to set the color for all normal points to the background color. Write a new `:adjust-screen-point` method that implements this idea.

Higher-Level State Messages

Several other messages are available for accessing and changing point states. The messages `:point-showing`, `:point-hilited`, and `:point-selected` require a point index and return `t` or `nil` to indicate whether the point is visible, highlighted, or selected, respectively. Given a second argument of `t` or `nil`, the methods for these messages make the appropriate change to a point's state. All three methods are vectorized, and all three send the `:adjust-screen` message before returning.

The `:selection` message returns a list of the points currently selected. Given a list of indices as an optional argument, it unselects all selected points and selects all points in the argument list that are visible. It sends the `:adjust-screen` message before returning. The method for the `:points-selected` message is identical to the `:selection` method. The messages for `:points-hilited` and `:points-showing` are analogous.

The `:erase-selection` method changes the states of all currently selected points to invisible, and the `:show-all-points` method changes all points states to `normal`. The `:focus-on-selection` method sets the states of all unselected points to invisible. All three methods send the `:adjust-screen` message before returning.

Two useful predicate messages are available. The method for the message `:any-points-selected-p` returns `t` if any of the points in the plot are selected, and `nil` otherwise. The `:all-points-showing-p` message returns `t` if all points are visible, and `nil` if any are invisible.

9.1.6 Window Layout, Resizing, and Redrawing

The Resize Method

The `graph-proto` `:resize` method is based on certain assumptions about the layout of the plot. The content of the plot, the image of the points and lines, is restricted to a rectangle called the *content rectangle*. If the plot contains axes, they are drawn immediately outside the content rectangle. The entire plot, consisting of the content and the axes, is surrounded by a *margin*. This margin can be used to hold controls for a plot, such as the rotation controls of the standard rotating plot. The margin reserves a fixed number of pixels along the left, top, right, and bottom edges of the window canvas. When a plot is resized, the `:resize` message keeps the margin fixed and changes the size of the content rectangle to fill the rest of the canvas. The exact size and

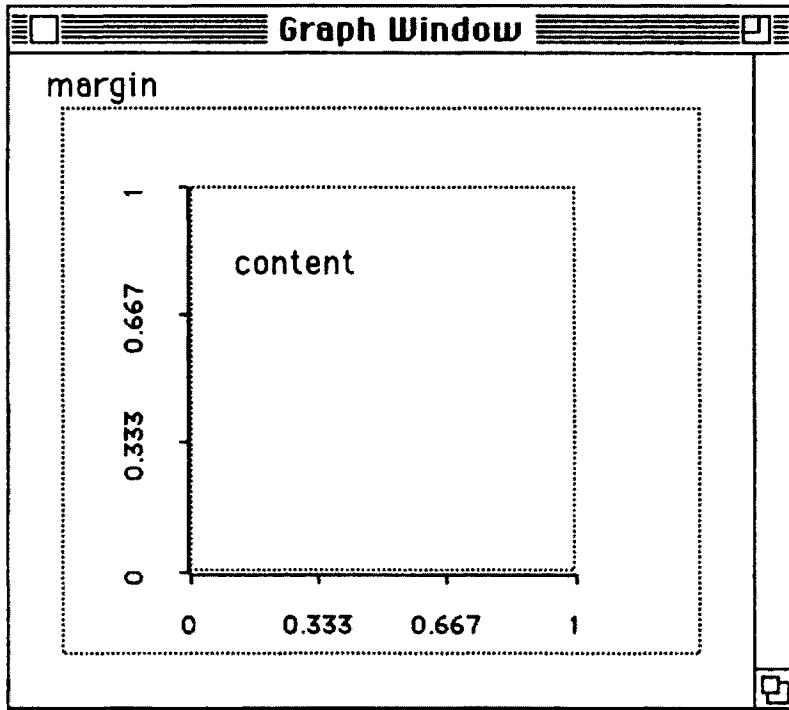


Figure 9.3: Content and margin for a plot with axes and a fixed aspect ratio.

shape of the content rectangle depend on whether space is needed for x or y axes, whether the axes have labels, and whether the content rectangle is to maintain a *fixed aspect ratio*. For a fixed aspect ratio, the content rectangle is the largest square that fits inside the margins and leaves room for the axes. If the plot allows the aspect ratio to vary, the content rectangle is the largest rectangle satisfying these constraints. Figure 9.3 shows the margin and content rectangle for a plot with axes and a fixed aspect ratio.

The content rectangle can be set and retrieved by using the `:content-rect` message. With no arguments, this message returns a list of the coordinates of the content rectangle (i.e., a list of the two coordinates of the left top corner, the width and the height):

```
> (send w :content-rect)
(0 0 250 250)
```

If the message is called with four integer arguments, it sets the content rectangle to the specified coordinates.

The margin for a plot can be set and retrieved with the `:margin` message. Called with no arguments, this message returns a list of four integers repre-

senting the margin sizes in pixels for the left, top, right, and bottom edges of the canvas. Initially, the margins of a plot created from the `graph-proto` prototype are all zero:

```
> (send w :margin)
(0 0 0 0)
```

To set a new margin, the `:margin` message can be given four integer arguments.

The aspect ratio of a plot can be determined and set with the message `:fixed-aspect`. The result is `non-nil` for a fixed aspect ratio and `nil` for a variable aspect ratio. For our example the aspect ratio is initially variable:

```
> (send w :fixed-aspect)
NIL
```

To change the aspect ratio, you can send the message with a `nil` or `non-nil` argument. The expression

```
(send w :fixed-aspect t)
```

gives our plot a fixed aspect ratio.

If any of the layout features are changed using the `:margin`, `:fixed-aspect`, `:x-axis`, or `:y-axis` messages, then the methods for these messages send the `:resize` and `:redraw` messages to the graph, unless the `:draw` keyword argument is given with value `nil`.

Before returning, the `:resize` method sends the plot the `:redraw-overlays` message. Overlays, described in detail in Section 9.1.7, provide a convenient method for implementing plot controls, such as the rotation controls for a rotating plot.

The Redraw Method

The `:redraw` method for the graph prototype breaks the redrawing process into several stages. First, it sends the `:redraw-background` message. The method for this message erases the canvas and draws any axes that are needed for the current plot settings. Next, the `:redraw` message sends the plot the `:redraw-overlays` message. Finally, the `:redraw-content` message is sent to the plot. The method for this message is responsible for drawing the points and lines themselves. The methods for the `:redraw`, `:redraw-content`, and `:redraw-background` messages all use the buffer.

The reason for separating the background drawing from the content drawing is that drawing the axes, and overlays, may be rather time-consuming and is not necessary if only the content is changed. As a result, methods for messages like `:adjust-screen` and `:rotate-2` only need to send the `:redraw-content` message to change the content of a plot.

In developing a new plot, it is rarely necessary to override the `graph-proto` `:redraw` method. It is usually sufficient to write a new `:redraw-content` method.

Lower-Level Messages

To aid its drawing methods, the graph-proto transformation system maintains integer versions of the transformed point and linestart coordinates. A range of integers, called the *canvas range*, is associated with each variable. The system scales the transformed coordinate values from the scaled range to the canvas range. Thus if a variable has a canvas range from 0 to 200, then a scaled coordinate of 0.5 in a scaled range of -1.0 to 1.0 corresponds to a canvas coordinate of 150.

The canvas range of a variable can be retrieved and set with the :*canvas-range* message. For the first variable in our example plot, this range is

```
> (send w :canvas-range 0)
(0 232)
```

The :*resize* method sets the canvas range for the first of the current variables to the range from 0 to the width of the content rectangle. The range for all other variables is set to the range from 0 to the height of the content rectangle.

The canvas coordinates of points and linestarts can be retrieved using the :*point-canvas-coordinate* and :*linestart-canvas-coordinate* messages. These messages are used like the other coordinate messages introduced above. The canvas coordinates of the first two variables for the first point are obtained as

```
> (send w :point-canvas-coordinate 0 0)
80
> (send w :point-canvas-coordinate 1 0)
58
```

The drawing system also uses a point in the content rectangle called the *content origin*. This origin can be set and retrieved using the :*content-origin* message:

```
> (send w :content-origin)
(9 250)
```

The :*resize* method places this origin at the lower left corner of the content rectangle, and the :*redraw-content* method uses the canvas coordinates as offsets from this origin.

A final message used in this context is :*content-variables*. The method for this message is identical to the :*current-variables* method, except that it does not redraw the plot when the variables are changed.

9.1.7 Plot Overlays

Overlays can be viewed as transparent sheets that are placed over the plot background before the content is drawn. These sheets are maintained in

order and are drawn from bottom to top. In order not to be covered by the content of a plot, overlays usually only draw in the margin of a plot.

A key feature of plot overlays is their ability to intercept or filter mouse clicks. The `:do-click` method for the `graph-proto` prototype allows each overlay, starting from the top and working down, to accept or pass on a mouse click. The click is only passed on to the current mouse mode if it is not accepted by any overlay. This feature makes it possible to use overlays to implement simple plot controls, such as buttons for having a plot carry out a particular action. The rotation controls for a rotating plot are implemented as an overlay.

Plot overlays are objects that inherit from the `graph-overlay-proto` prototype. An overlay can be added to a plot by sending the plot the `:add-overlay` message with one argument, the overlay object. This places the new overlay on top of any overlays already in the plot. An overlay can be removed by sending the plot the `:delete-overlay` message with the overlay object as its argument.

When a plot is resized, the `:resize` method sends the plot the `:resize-overlays` message before returning. The method for this message sends each overlay in the plot the `:resize` message. When a plot is redrawn, the `:redraw` method sends the plot the `:redraw-overlays` message. The method for this message sends each overlay the `:redraw` message, starting with the bottom overlay and working up to the top of the overlay stack.

The `graph-proto` `:do-click` method sends overlays the `:do-click` message with the arguments it received. It starts with the top overlay, and continues through the overlays until one of the overlays returns a non-nil value as the result of the click. If all overlays return `nil` in response to the click, then the click is passed on to the click method for the current mouse mode.

As an example, we can add an overlay to our stack loss plot that contains a “button,” a small square followed by a label. When a click occurs in the square, the plot is instructed to run a smooth interpolation from the temperature versus airflow plot to the ammonia loss versus concentration plot. To avoid the need to reposition the button when the plot is resized, we can place it at the left top corner of the plot.

The expression

```
(let ((h (+ (send w :text-ascent) (send w :text-descent))))
  (send w :margin 0 (round (* 1.5 h)) 0 0))
```

sets the top margin to leave enough room for one line of text in the window's font. We can construct the overlay object by sending the `:new` message to the overlay prototype:

```
(setf interp-overlay (send graph-overlay-proto :new))
```

Next, we can give the overlay a slot for holding information describing the location of the button box and the label string.

```
(let* ((ascent (send w :text-ascent))
      (x ascent)
      (y (round (* 1.5 ascent)))
      (box ascent))
  (send interp-overlay :add-slot 'location
    (list x y box (round (+ x (* 1.5 box))))))
```

and we can define a method for a reader message for the slot as

```
(defmeth interp-overlay :location () (slot-value 'location))
```

The :redraw method for the overlay can now be defined to obtain positioning information with the :location message, and draw a box and a label at the specified location:

```
(defmeth interp-overlay :redraw ()
  (let* ((loc (send self :location))
         (x (first loc))
         (y (second loc))
         (box (third loc))
         (string-x (fourth loc))
         (graph (send self :graph)))
    (send graph :frame-rect x (- y box) box box)
    (send graph :draw-string "Interpolate" string-x y)))
```

This method sends the overlay the :graph message to determine the plot containing the overlay. The overlay system ensures that this message returns the proper result when the overlay is installed in a plot.

The :do-click method returns nil unless the click falls inside the button box. If the click is in the box, the method sends the plot the :interpolate message and returns t, to ensure that the click is not processed further:

```
(defmeth interp-overlay :do-click (x y m1 m2)
  (let* ((loc (send self :location))
         (box (third loc))
         (left (first loc))
         (top (- (second loc) box))
         (right (+ left box))
         (bottom (+ top box))
         (graph (send self :graph)))
    (when (and (< left x right) (< top y bottom))
      (send graph :interpolate)
      t)))
```

To complete the example, we need to define an :interpolate method for our plot. A simple definition can be based on the smooth rotation loops shown in Section 9.1.3. For example,

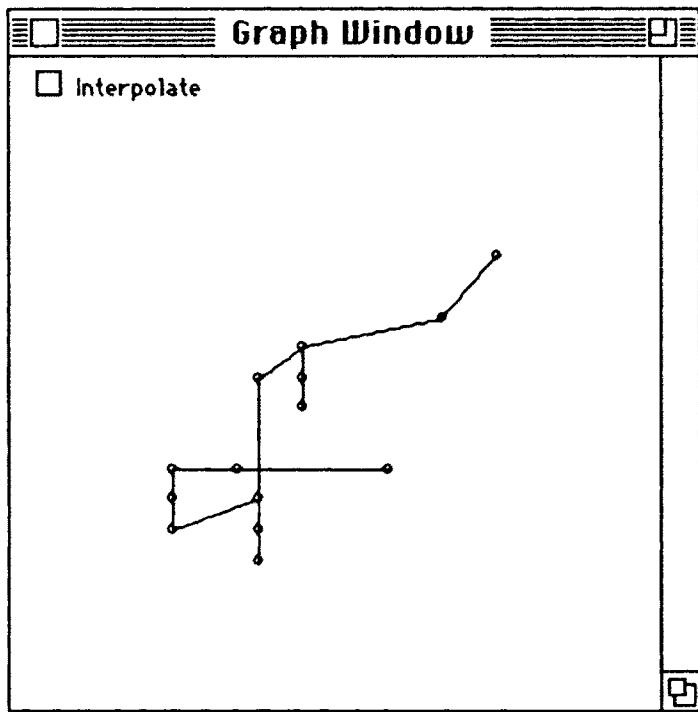


Figure 9.4: Plot of the stack loss data with an interpolation button in an overlay.

```
(defmeth w :interpolate ()
  (send self :transformation nil)
  (dotimes (i 10)
    (send self :rotate-2 0 2 (/ pi 20) :draw nil)
    (send self :rotate-2 1 3 (/ pi 20))))
```

Finally, we can install the overlay in our plot by using the expression

```
(send w :add-overlay interp-overlay)
```

The result is shown in Figure 9.4. A click in the content of the plot is processed as before, but a click in the button causes the plot to run the interpolation.

After experimenting with this overlay, we can remove it with the expression

```
(send w :delete-overlay interp-overlay)
```

Exercise 9.8

The `:do-click` method for the interpolation overlay runs the interpolation as soon as a click occurs. In most user interfaces, push buttons are highlighted when they are pressed, but don't execute their actions unless the mouse is released over the button. Modify the `:do-click` to incorporate this behavior.

Exercise 9.9

The interpolation in this example is always run in the same direction. Modify the example to run the interpolation forward or backward, depending on the current state of the plot's transformation.

Exercise 9.10

Revise the interpolation overlay to put the button at the left bottom corner of the plot.

9.1.8 Menus and Menu Items

The `graph-proto :isnew` method gives a new plot a menu by sending the plot object the `:new-menu` message. The method for this message in turn sends the object the `:menu-title` and `:menu-template` messages. The `:menu-title` message returns a title string:

```
> (send w :menu-title)  
"Plot"
```

It can also be used to install a new menu title for use in constructing a new plot menu. This is primarily useful for giving a new prototype a more suitable menu title.

The `:menu-template` method is expected to return a list that is used for constructing the menu items. The list can contain menu item objects, which are simply installed in the menu, or it can contain symbols for specifying standard menu items. The `graph-proto` method for this message returns a list of symbols:

```
> (send w :menu-template)  
(LINK SHOWING-LABELS MOUSE RESIZE-BRUSH ...)
```

The `:new-menu` method constructs an appropriate menu item for each of these symbols. The following symbols can be used to specify standard menu items:

- `color` – sends the `:set-selection-color` message, which presents a dialog for changing the color of selected points.
- `dash` – a disabled dash item for separation.

- **focus-on-selection** – sends the `:focus-on-selection` message.
- **link** – toggles linking of the graph with the `:linked` message.
- **mouse** – sends the `:choose-mouse-mode` message, which presents a dialog for changing the mouse mode.
- **options** – sends the plot the `:set-options` message, which presents a dialog for setting several options.
- **redraw** – sends the `:redraw` message.
- **erase-selection** – sends the `:erase-selection` message.
- **rescale** – sends the `:adjust-to-data` message.
- **save-image** – sends the `:ask-save-image` message, which presents a dialog for saving the image to a file.
- **selection** – sends the `:selection-dialog` message.
- **show-all** – sends the `:show-all-points` message.
- **showing-labels** – toggles showing labels with the `:showing-labels` message.
- **symbol** – sends the `:set-selection-symbol` message, which presents a dialog for changing the symbol of selected points.

Several of these menu items use the messages `:any-points-selected-p` or `:all-points-showing-p` to determine whether they should be enabled or not.

The `:new-menu` message can also be given an optional title string argument, which is used instead of the result of the `:menu-title` message. An alternative list of items can be supplied with the `:items` keyword. The item list can contain any of the symbols listed above to specify standard menu items. For example, the expression

```
(send w :new-menu "Stack Loss" :items '(link dash rescale))
```

gives the stack loss plot a menu containing only the `link` and `rescale` items, which are separated by a line. An alternative approach is to define a new `:menu-template` method and then to send the `:new-menu` message without the `:items` keyword. This is useful primarily for developing new graph prototypes.

9.1.9 Miscellaneous Messages

Several additional messages are available in the `graph-proto` prototype. The `:showing-labels` message sets and retrieves the state of the labeling option. If it is non-nil, the drawing methods place labels near highlighted and selected points.

Several methods are designed for plots with $m = 2$ dimensions. The `:add-function` message takes a function of one real argument, two real numbers representing lower and upper bounds on an interval, and a keyword argument specifying a number of points. The method for this message constructs a grid, evaluates the function on the grid, and adds the corresponding set of lines to the plot. It also accepts the keyword arguments `:color` and `:type` for specifying the color and line type to use. The plot is sent the `:redraw-content` message, unless the `:draw` keyword is supplied with the value `nil`.

The message `:abline` takes an intercept and a slope for a linear function and adds it to the graph. The range for the independent variable is the current range of the x axis.

The `:adjust-depth-cuing` message is used primarily by rotating plots, but can be used by other plots as well. It takes an integer dimension index as its argument, and uses the canvas coordinates for that dimension, viewed as an out-of-screen axis, to apply depth cuing to points and lines in the plot. Points are depth cued by changing their symbols to one of the symbols `dot1`, `dot2`, `dot3` and `dot4`. The symbol `dot1` is used for points farthest from the viewer, and `dot4` is used for points closest to the viewer. Lines are depth cued by changing their width.

9.2 Some Standard Graph Prototypes

The five standard graph types introduced in Chapter 2 are constructed using prototypes that inherit from the `graph-proto` prototype. Even though these graphs vary considerably in appearance, they only require overriding and adding a small number of methods to the ones provided by `graph-proto`.

9.2.1 Scatterplots

The basic scatterplot prototype `scatterplot-proto` is used for plots constructed by the functions `plot-points` and `plot-lines`. It requires at least two-dimensional data. For two-dimensional scatterplots the methods for `:add-points` and `:add-lines` allow new data to be specified using two separate lists instead of requiring a list of lists. Thus if `s` is a two-dimensional scatterplot and `x` and `y` are lists of numbers of equal length, then

```
(send s :add-points x y)
```

and

```
(send s :add-points (list x y))
```

are equivalent.

The `:adjust-to-data` method adds axes to a graph if its scale type is `nil`. It also sets the ranges to produce nice axis marks.

9.2.2 Scatterplot Matrices

The scatterplot matrix prototype `scatmat-proto` requires at least two-dimensional data. It uses a fixed aspect ratio. Its content origin is at the lower left corner of the content rectangle, and the canvas range for each variable is set to the left and right limits of its column of subplots, measured from the origin. The `:do-click` and `:do-motion` methods are modified to set the content variables to the indices of the subplot containing the cursor. This allows many inherited methods to work properly without changes.

The methods for the four messages `:add-lines`, `:add-points`, `:adjust-screen-point`, and `:redraw-content` are modified to properly draw all subplots in the matrix. If the scale type of the graph is `nil`, the default, the `:redraw-content` method also draws variable labels and limits in the diagonal subplots.

9.2.3 Rotating Plots

The prototype for rotating plots produced by the `spin-plot` function is `spin-proto`. This prototype requires at least three-dimensional data. It uses a fixed aspect ratio, and its default scale type is `variable`. The `:content-variables` method uses three content variable indices. The first two are used in the usual way, and the third is the index of the depth cuing variable. The `:resize` method places the content origin in the center of the content rectangle.

The `:redraw-content` method adjusts depth cuing before performing a standard content redraw. If the plot is showing the axes, the axes are redrawn by the `:redraw-content` method by sending the plot object the `:draw-axes` message. If the scale type is `nil`, the `:adjust-to-data` method adjusts the plot range to ensure that all points are visible in a rotation around the origin. The messages `:depth-cuing` and `:showing-axes` can be used to determine whether a graph is using depth cuing or is showing the axes. Sent with a non-`nil` or `nil` argument, they can be used to change the corresponding state of the graph.

The controls at the bottom of a rotating plot are implemented as an overlay added by the `:isnew` method. These buttons set the type of rotation to `pitching`, `rolling`, or `yawing`, adjust the sign of the angle, and send the `:rotate` message while the mouse button is held down. The `:idle` method sends this message as well. The current angle in a rotating plot can be set and retrieved using the `:angle` message. It is measured in radians. The rotation type is set and retrieved using the `:rotation-type`

message. Its value can be one of the symbols `pitching`, `rolling`, or `yawing`, or it can be a rotation matrix to be applied as an incremental rotation. If the rotation type is a symbol, the `:rotate` method uses `:rotate-2` with the appropriate variables and angle. If the type is a matrix, `:rotate` uses `:apply-transformation`.

In addition to adding the control overlay, the `:isnew` method for this prototype sets the aspect ratio of the graph to fixed, and sets an appropriate margin for the overlay.

The `:add-function` method for the rotating plot prototype is designed for a three-dimensional graph and requires a function of two arguments followed by four real numbers, the limits on the x range and the limits on the y range. The message `:abcplane` takes three real arguments a , b , and c , and adds line segments to represent a section of the function $f(x, y) = a + bx + cy$.

9.2.4 Histograms

The histogram prototype is `histogram-proto`. Histograms display m -dimensional point data in an $m + 1$ -dimensional plot. The additional dimension is used for the y axis of the graph, and the first dimension is binned and displayed as the histogram. For example, the expression

```
(setf hs (histogram (list air temp conc loss)))
```

constructs a histogram for the stack loss data. The number of variables for this histogram is 5,

```
> (send hs :num-variables)
5
```

and the content variables are

```
> (send hs :content-variables)
(0 4)
```

Thus the plot initially shows a histogram of the first variable in the data set, the airflow. If a transformation is applied to the plot `hs`, it will show a histogram of the first dimension of the transformed data. For example, after setting its scale type to `variable` by

```
(send hs :scale-type 'variable)
```

the expression

```
(dotimes (i 10) (send hs :rotate-2 0 1 (/ pi 20)))
```

rotates in 10 steps to a histogram of the temperature variable.

For a histogram of one-dimensional data, the `:add-points` method can be given a sequence of numbers instead of a list of one sequence of numbers. For a histogram of m -dimensional data, the `:add-points` method requires a list of m sequences. The stack loss data could thus have been installed with the expression

```
(send hs :add-points (list air temp conc loss))
```

The method for `:add-lines`, which might be used to add a density to a histogram, requires a list of $m + 1$ sequences. The final sequence is used as the y coordinate. For example, if `hp` is a histogram of the one-dimensional precipitation data of Section 2.2.1 constructed as

```
(setf hp (histogram precipitation))
```

then the expression

```
(let* ((mu (mean precipitation))
      (s (standard-deviation precipitation))
      (x (rseq (min precipitation) (max precipitation) 30)))
      (y (/ (normal-dens (/ (- x mu) s)) s)))
  (send hp :add-lines (list x y)))
```

adds the graph of a normal density to the histogram.

Since the methods for `:transformation` and `:apply-transformation` can be given matrices of dimension lower than the plot dimension, you can use transformations of the same dimension as the point data in a histogram.

The histogram prototype provides its own methods for the messages `:adjust-screen`, `:adjust-screen-point`, `:redraw-content`, and `:adjust-points-in-rect`. The `:adjust-to-data` method adds an x -axis if the scale type is `nil`. This method also recomputes the bins used in the histogram. The methods for `:clear-points` and `:resize` are also modified to adjust the bins.

Two messages provide access to information about the bins. The `:bin-counts` message returns a list of the counts in each bin. The `:num-bins` message with no argument returns the number of bins currently used. Called with a positive integer argument, it changes the number of bins to the specified value. When a new number of bins is specified, the graph object is sent the `:redraw` message, unless the keyword argument `:draw` is given with value `nil`.

9.2.5 Name Lists

The final standard graph prototype is `name-list-proto`. Its purpose is to provide a linkable list of point labels. It does not use any numerical data.

A name list can be constructed using $m = 0$ as the number of dimensions. The `:add-points` method for this prototype can be given a number specifying the number of points to add instead of a list of lists of coordinate values. For example, a name list for the stack loss data can be constructed as

```
(setf n (send name-list-proto :new 0))
```

and a set of 21 labels can be added by

```
(send n :add-points 21)
```

The `:add-lines` method is redefined to do nothing, since the name list does not display line data. The `:adjust-screen`, `:adjust-screen-point`, `:redraw-content`, and `:adjust-points-in-rect` methods are redefined as well. The `:isnew` method ensures that a name list has a vertical scroll bar.

Chapter 10

Some Dynamic Graphics Examples

Research on the effective use of dynamic graphical methods in statistics is just beginning. The Lisp-Stat graphics system is designed to support this research effort by supporting the exploration of variations on standard methods as well as the development of new methods. This chapter presents a number of examples to illustrate the use of the system along these lines. The examples are chosen both to introduce new statistical ideas proposed in the recent literature and to illustrate programming techniques that are useful in implementing these ideas in Lisp-Stat. Some of the examples are quite short and straightforward, while others are more extensive.

10.1 Some Animations

One of the earliest examples of the use of dynamic graphics in statistics is a system developed by Fowlkes in the late 1960s to examine the effect of power transformations on a probability plot [30]. The plot was drawn on a CRT display, and the parameters of the transformation were controlled by dials on the display. This form of animation, in which a plot depends on one or more continuous parameters that can be adjusted using mechanical or graphical controls, is useful in a wide variety of situations. It is very easy to implement in Lisp-Stat. A first example, a version of Fowlkes' power transformation plot, was given in Section 2.7.3. This section reexamines the power transformation example and presents two additional examples.

10.1.1 Another Look at Power Transformations

Section 2.7.3 gave an example of an animated power transformation plot for the precipitation data introduced in Section 2.2.1. The plot was set up by

first sorting the precipitation data, and then plotting the sorted data against the corresponding normal scores. After a change in the power, the plot was redrawn by sending the `:clear` message followed by an `:add-points` message to add the new data.

This approach has two drawbacks. First, by sorting the data the correspondence between the original indexing of the precipitation data and the data in the plot is lost. This means that linking the power transformation plot to another plot of the precipitation data will not produce the right result. A second problem is that point characteristics, such as color, symbol, and state, are not preserved when the power is changed.

We can overcome both these problems by taking advantage of some of the messages introduced in the previous chapter. To avoid having to sort our data, we can use the ranks to construct the normal scores as

```
(let ((ranks (rank precipitation)))
  (setf nq (normal-quant (/ (+ ranks 1) 31))))
```

Adding 1 to each rank is necessary since the `rank` function returns zero-based ranks. As in Section 2.7.3, we can define a transformation function `bc` as

```
(defun bc (x p)
  (let* ((bcx (if (< (abs p) .0001) (log x) (/ (^ x p) p)))
         (min (min bcx))
         (max (max bcx)))
    (/ (- bcx min) (- max min))))
```

The initial plot for a power of 1 is then set up by

```
(setf w (plot-points nq (bc precipitation 1)))
```

Since the power used is 1, the function `bc` just rescales the data.

To change the power used in the plot, we can define a function `change-power`. Instead of clearing the point data, we can use the `:point-coordinate` message to change the *y* coordinates of the points in the plot. All other characteristics of the points remain unchanged. Since the `:point-coordinate` message requires a list of the indices of all points, it is useful to set up a variable containing these indices:

```
(setf indices (iseq 30))
```

This avoids having to construct the list each time the power is changed. Using this variable, the `change-power` function is defined as

```
(defun change-power (p)
  (send w :point-coordinate 1 indices (bc precipitation p))
  (send w :redraw-content))
```

Only one :point-coordinate message is needed since the method for this message is vectorized. The :redraw-content message is needed since the :point-coordinate method does not redraw the plot.

We can use the change power function from the interpreter, or we can use it as the action function of a slider:

```
(setf slider
      (interval-slider-dialog '(-1 2) :action #'change-power))
```

The default initial value in the slider is the lower end point of the interval. The expression

```
(send slider :value 1)
```

sets the value in the slider to the power used in constructing the plot. Since the slider is designed specifically to control the plot w, the slider should be removed from the screen when the plot is closed. We can ensure that this happens by registering the slider as a subordinate of the plot with the message

```
(send w :add-subordinate slider)
```

For larger data sets, or slow computers, an animation like this may move rather slowly. If an animation depends on only one parameter, it is often possible to precompute all the values that might be needed. For our example, we can use

```
(setf powers (rseq -1 2 31))
```

to set up a list of powers and calculate the transformed data as a list of lists:

```
(setf data (mapcar #'(lambda (p) (bc precipitation p)) powers))
```

A sequence slider can then be used to scroll through this data list:

```
(flet ((change (x)
              (send w :point-coordinate 1 indices x)
              (send w :redraw-content)))
  (sequence-slider-dialog data
    :display powers
    :action #'change))
```

The argument to the action function in this slider is a list of the transformed coordinate values for the current power. The list of powers is used as the display sequence for the slider.

Precomputation often leads to significant speed improvements in animations. But it usually requires additional programming effort. Because of the large amount of space needed to hold precomputed results, precomputation is usually not feasible if there is more than one animation parameter.

Exercise 10.1

Write a function that takes a data sequence and constructs an animated power transformation plot for the sequence.

10.1.2 Plot Interpolation

One of the major objectives of dynamic graphics is to provide ways of visualizing data in more than three dimensions. One approach for examining four-dimensional data is to split the variables into pairs (x_1, y_1) and (x_2, y_2) , and use animation to move continuously between the two-dimensional scatterplots of the pairs. This technique is called *plot interpolation* [16].

Several approaches can be used to interpolate between two plots. A natural first choice is convex interpolation. That is, set

$$\begin{aligned} x_p &= (1 - p)x_1 + px_2 \\ y_p &= (1 - p)y_1 + py_2 \end{aligned}$$

and show a plot of y_p against x_p as p varies continuously from 0 to 1. Unfortunately, there is a problem with this approach. When it is used on uncorrelated data, the point cloud shown by the interpolation shrinks as p moves from 0 to 0.5, and expands again as p moves from 0.5 to 1.

To understand the source of the problem, suppose we use the interpolation to view a sample from a four-dimensional spherical normal distribution. The components are independent standard normal random variables, so the initial and final plots in the interpolation show plots of two independent standard normal variables. But for $p = 0.5$, the variables x_p and y_p are averages of two independent standard normal variables, and therefore have standard deviation $1/\sqrt{2}$.

To avoid this problem, Buja et al. [16] recommend using trigonometric interpolation. This sets

$$\begin{aligned} x_p &= \cos(p\pi/2)x_1 + \sin(p\pi/2)x_2 \\ y_p &= \cos(p\pi/2)y_1 + \sin(p\pi/2)y_2 \end{aligned}$$

and plots y_p against x_p as p varies continuously from 0 to 1. This preserves component variances if all variables are scaled to have the same variances. It is equivalent to the rotations used in Section 9.1.3.

Trigonometric plot interpolations can be implemented in several ways. One approach is to use a two-dimensional plot, and change the data in the plot with the :point-coordinate message.

To start, we can define a function that standardizes the data as

```
(defun standardize (x)
  (let ((x-bar (mean x))
        (s (standard-deviation x)))
    (/ (- x x-bar) s)))
```

Using the stack loss data of Section 5.6.2, we can construct four standardized variables as

```
(setf std-air (standardize air))
(setf std-temp (standardize temp))
(setf std-conc (standardize conc))
(setf std-loss (standardize loss))
```

The `plot-points` function can be used to set up a plot of the first two variables:

```
(setf w (plot-points std-air std-temp))
```

We need to set the ranges of the two variables to ensure that they are large enough to show any rotation of the data. For the standardized data, a range of $[-3, 3]$ in both variables is sufficient. Since the method for the `:range` message is vectorized, both ranges can be set with one expression:

```
(send w :range '(0 1) -3 3)
```

After storing a list of the point indices in the variable `indices` with

```
(setf indices (iseq (length std-air)))
```

the function `interpolate` defined by

```
(defun interpolate (p)
  (let* ((alpha (* (/ pi 2) p))
         (s (sin alpha))
         (c (cos alpha))
         (x (+ (* c std-air) (* s std-temp))))
    (y (+ (* c std-conc) (* s std-loss))))
  (send w :point-coordinate 0 indices x)
  (send w :point-coordinate 1 indices y)
  (send w :redraw-content)))
```

takes a point in the range $[0, 1]$, calculates the corresponding angle, and sets the variables in the plot to the interpolation specified by the angle. The interpolation can now be run with a loop like

```
(dolist (p (rseq 0 1 30)) (interpolate p))
```

or a slider constructed by

```
(interval-slider-dialog '(0 1) :action #'interpolate)
```

As in the power transformation example, the slider should be registered as a subordinate with the plot window to ensure that it is closed when the window is closed.

A second approach to constructing a trigonometric plot interpolation is to use a four-dimensional scatterplot, and take advantage of the transformation

system provided by the `graph-proto` prototype. The `plot-points` function can again be used to set up the plot. It can be given a list of the four variables, and it also allows the initial scale type to be specified with the `:scale-type` keyword:

```
(setf w (plot-points (list std-air std-conc std-loss std-temp)
                         :scale-type 'fixed))
```

The fixed scale type is appropriate since we are using standardized variables.

The `interpolate` function can now be defined as

```
(defun interpolate (p)
  (let* ((alpha (* (/ pi 2) p))
         (s (sin alpha))
         (c (cos alpha))
         (m (make-array '(4 4) :initial-element 0)))
    (setf (aref m 0 0) c)
    (setf (aref m 0 2) s)
    (setf (aref m 1 1) c)
    (setf (aref m 1 3) s)
    (send w :transformation m)))
```

The transformation matrix in this definition is not an orthogonal transformation, since it sets the third and fourth transformed coordinates to zero. We could use an orthogonal matrix, but it is not necessary to do so.

The `interpolate` function can be used as before to run the interpolation from a loop or a slider dialog. Figure 10.1 shows four views of the interpolation.

There are a number of similarities between the power transformation animation and the plot interpolation animation, but there is also one major difference. In the power transformation example, the objective is to find a reasonable power and explore values around it. In using the animation, we would typically start with a wide range, and then gradually narrow that range down. In contrast, in the plot interpolation we would almost always want to run the animation completely from the initial pair to the final pair. Intermediate plots do have interpretations as rotations of the data, but in the interpolation they are not of primary interest. The purpose of the intermediate plots is to allow the viewer to easily follow individual points or groups of points as they move from one scatterplot to the other. The objective of the interpolation is thus similar to the objective of linking two scatterplots. As a result, a button control, such as the button overlay constructed in the example in Section 9.1.7, may be a better graphical control for running a plot interpolation than a scroll bar.

Exercise 10.2

Write a function to set up a plot interpolation for a data set with four or

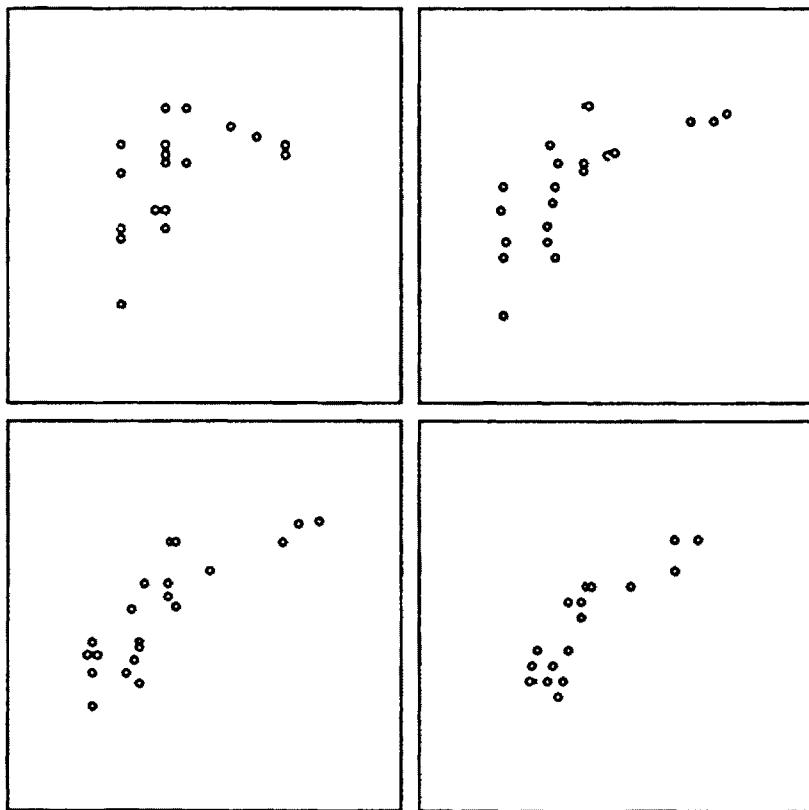


Figure 10.1: Four interpolated views of the stack loss data.

more variables. Provide a slider control for the interpolation, and a menu item for choosing the variable pairings to use in the interpolation.

10.1.3 Choosing a Smoothing Parameter

The two examples discussed so far used animation to show changes in point data. Animation can also be used with line data. As an example, we can use animation to provide a graphical means for selecting a smoothing parameter for a kernel density estimate for the precipitation data. The range of the precipitation data is given by

```
> (min precipitation)
0.32
> (max precipitation)
4.75
```

The `kernel-dens` function accepts a window width supplied with the `:width` keyword. For the range of the precipitation data, an initial window of size 1 may be reasonable. The expression

```
(setf w (plot-lines (kernel-dens precipitation :width 1)))
```

sets up a plot using the default bisquare kernel and a window of width 1.

Since we might like to change both the kernel width and the kernel type, we can install slots to hold the current kernel specification in the plot with

```
(send w :add-slot 'kernel-width 1)
```

and

```
(send w :add-slot 'kernel-type 'b)
```

Assuming that the graph of the density estimate can be reset with a message `:set-lines`, accessor methods for these slots can be defined as

```
(defmeth w :kernel-width (&optional width)
  (when width
    (setf (slot-value 'kernel-width) width)
    (send self :set-lines))
  (slot-value 'kernel-width))
```

and

```
(defmeth w :kernel-type (&optional type)
  (when type
    (setf (slot-value 'kernel-type) type)
    (send self :set-lines))
  (slot-value 'kernel-type))
```

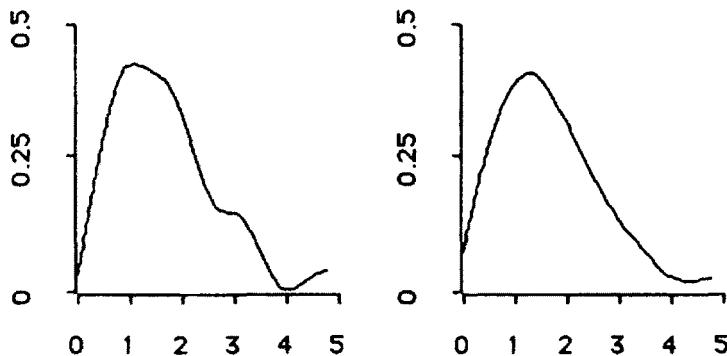


Figure 10.2: Kernel density estimate for the precipitation data using a bisquare kernel and two different smoothing parameter values.

The `:set-lines` method can be defined using the `:clear-lines` and `:add-lines` messages:

```
(defmeth w :set-lines ()
  (let ((width (send self :kernel-width))
        (type (send self :kernel-type)))
    (send self :clear-lines :draw nil)
    (send self :add-lines
          (kernel-dens precipitation
                         :width width :type type))))
```

To avoid losing any color, width, or line type information in the linestarts, we could use the `:linestart-coordinate` message instead. But maintaining the linestart properties is usually less important than maintaining point properties.

Again, a slider can be used to change the kernel window. For this data set a reasonable range to explore is the interval [0.25 1.5]:

```
(interval-slider-dialog '(.25 1.5)
                        :action
                        #'(lambda (s)
                            (send w :kernel-width s)))
```

The slider value should be set to the current kernel value of the plot, and the slider should be registered as a subordinate of the window. By default, the display field of the slider shows the current window width. An alternative would be to have it display the value of one of the criteria for selecting smoothing parameters available in the literature. Figure 10.2 shows the density estimate for two different window widths.

To simplify changing the kernel type, we can define a `:choose-kernel` method that puts up a dialog for selecting the kernel:

```
(defmeth w :choose-kernel ()
  (let* ((types '("Bisquare" "Gaussian" "Triangle" "Uniform"))
         (i (choose-item-dialog "Kernel Type" types)))
    (if i (send w :kernel-type (select '(b g t u) i)))))
```

This message can be sent from the keyboard, or we can construct a menu item

```
(setf kernel-item
  (send menu-item-proto :new "Kernel Type"
    :action #'(lambda () (send w :choose-kernel))))
```

and install the item in the plot menu:

```
(send (send w :menu) :append-items kernel-item)
```

Exercise 10.3

Construct a prototype that implements a graphical kernel density estimator, and define a constructor function that takes a data sequence and returns a kernel density plot object.

Exercise 10.4

Modify your kernel density plot prototype to use point data and adjust the kernel when points are made invisible from a linked plot.

10.2 Using New Mouse Modes

How a plot responds to user actions depends on its mouse modes. A wide variety of effects can be achieved by adding new mouse modes to a plot. This section presents three examples that illustrate the range of possibilities.

10.2.1 Sensitivity in Simple Linear Regression

The first example is an interactive plot for illustrating the effect of leverage on a least squares regression line [9]. A scatterplot shows a set of (x, y) pairs along with a least squares regression line. A new mouse mode allows the points in the plot to be moved by clicking near a point and dragging. The regression line is redrawn when the mouse button is released. This example is primarily useful as an instructional tool, but variations on the basic idea may also be useful as exploratory tools.

To construct the example, we can start with some simulated data given by

```
(setf x (append (iseq 1 18) (list 30 40)))
(setf y (+ x (* 2 (normal-rand 20))))
```

The *y* values follow a normal linear regression with unit slope, and two of the points have *x* values giving them large leverages. The expression

```
(setf w (plot-points x y))
```

constructs a scatterplot of these data.

A new mouse mode, *point-moving*, is defined by

```
(send w :add-mouse-mode 'point-moving
      :title "Point Moving"
      :cursor 'finger
      :click :do-point-moving)
```

The method for the click message *:do-point-moving* can be defined using the *:drag-point* message as

```
(defmeth w :do-point-moving (x y a b)
  (let ((p (send self :drag-point x y :draw nil)))
    (if p (send self :set-regression-line)))
```

The message *:set-regression-line* is responsible for adjusting the regression line to the point data in the plot. It can be defined as

```
(defmeth w :set-regression-line ()
  (let ((coefs (send self :calculate-coefficients)))
    (send self :clear-lines :draw nil)
    (send self :abline (select coefs 0) (select coefs 1))))
```

This definition assumes that the *:calculate-coefficients* message can be used to determine the regression coefficients for the current data. For the least squares fitting method, the method for this message can be defined as

```
(defmeth w :calculate-coefficients ()
  (let* ((i (iseq 0 (- (send self :num-points) 1)))
         (x (send self :point-coordinate 0 i))
         (y (send self :point-coordinate 1 i))
         (m (regression-model x y :print nil)))
    (send m :coef-estimates)))
```

The advantage of separating the fitting process from the drawing process is that a new fitting method can be introduced by simply redefining the *:calculate-coefficients* method.

To complete the example, we can add the initial regression line,

```
(send w :set-regression-line)
```

and put the plot in the *point-moving* mode:

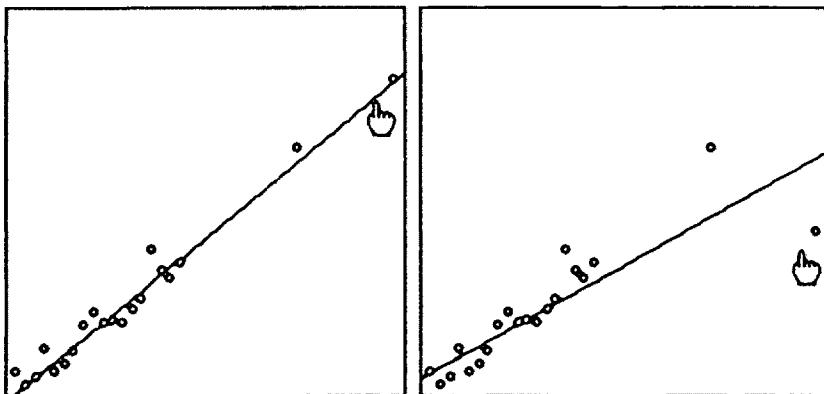


Figure 10.3: Data and line before (*left*) and after (*right*) moving the rightmost point.

```
(send w :mouse-mode 'point-moving)
```

Figure 10.3 shows the plot before and after moving one of the points.

This example can be improved in a number of ways. For instance, we can redefine the `:calculate-coefficients` method to use only the visible points in the plot:

```
(defmeth w :calculate-coefficients ()
  (let* ((i (send self :points-showing))
         (x (send self :point-coordinate 0 i))
         (y (send self :point-coordinate 1 i))
         (m (regression-model x y :print nil)))
    (send m :coef-estimates)))
```

Since the system for maintaining point states in a scatterplot always sets the `:needs-adjusting` flag when a point's state is changed to or from the `invisible` level, overriding the `:adjust-screen` method by

```
(defmeth w :adjust-screen ()
  (if (send self :needs-adjusting)
      (send self :set-regression-line))
  (call-next-method))
```

ensures that the regression line is recalculated when points are made invisible in the plot `w` or any linked plots.

Exercise 10.5

The second definition of `:calculate-coefficients` will not work properly if all points are invisible. Modify the method to produce a reasonable result in this case.

Exercise 10.6

Modify the `:calculate-coefficients` method to compute the regression coefficients by using a robust fitting method.

10.2.2 Hand Rotation

The rotation controls on a rotating plot allow the plot to be rotated around the screen *x* and *y* axes and the out-of-screen axis. Several other control strategies could be used. One strategy is based on imagining the data are contained in a “globe.” The globe can be rotated by “grabbing” and moving it with the mouse. When the mouse button is released, the rotation can stop or continue; it continues if the extend modifier is supplied with the click that grabs the globe. This control strategy can be implemented by defining a new mouse mode. Since this mouse mode is useful for any rotating plot, we can install it in the prototype `spin-proto`.

The new mouse mode can be defined as

```
(send spin-proto :add-mouse-mode 'hand-rotate
  :title "Hand Rotate"
  :cursor 'hand
  :click :do-hand-rotate)
```

The `hand` cursor seems a natural choice for this mode.

To develop the click method for the new mode, we need a method that converts a click in a rotating plot window to a point on the globe:

```
(defmeth spin-proto :canvas-to-sphere (x y rad)
  (let* ((p (send self :canvas-to-scaled x y)))
    (x (first p))
    (y (second p))
    (norm-2 (+ (* x x) (* y y))))
    (rad-2 (^ rad 2))
    (z (sqrt (max (- rad-2 norm-2) 0))))
  (if (< norm-2 rad-2)
    (list x y z)
    (let ((r (sqrt (/ norm-2 rad-2))))
      (list (/ x r) (/ y r) (/ z r))))))
```

The arguments to this method are the canvas coordinates of a click and the radius of a sphere in the scaled coordinate system. The sphere intersects the plane of the screen in a circle of the specified radius. A click inside this circle is converted to the point on the sphere above the click. A click outside the circle is projected onto the circle.

Using the `:canvas-to-sphere` message, the `:do-hand-rotate` method is

```
(defmeth spinproto :do-hand-rotate (x y m1 m2)
  (let* ((m (send self :num-variables))
         (range (send self :scaled-range 0))
         (rad (/ (apply #'- range) 2))
         (oldp (send self :canvas-to-sphere x y rad))
         (p oldp)
         (vars (send self :content-variables))
         (trans (identity-matrix m)))
    (flet ((spin-sphere (x y)
              (setf oldp p)
              (setf p (send self :canvas-to-sphere x y rad))
              (setf (select trans vars vars)
                    (make-rotation oldp p))
              (when m1
                  (send self :rotation-type trans)
                  (send self :idle-on t))
              (send self :apply-transformation trans)))
           (send self :idle-on nil)
           (send self :while-button-down #'spin-sphere))))
  
```

The local function `spin-sphere` is used as the `:while-button-down` action. It uses three variables defined in its surrounding environment. The variables `p` and `oldp` hold the locations of the current and previous clicks, translated to the globe. The variable `trans` holds a transformation matrix that is an identity matrix except in the rows and columns corresponding to the content variables. This definition is necessary since rotating plots can be used with more than three variables. The `spin-sphere` function updates the values of `oldp` and `p`, and then uses the new values as arguments to `make-rotation`. This function returns a rotation matrix that rotates the plot in the plane defined by the two points, keeping the orthogonal complement fixed. Before applying the rotation, `spin-sphere` checks the extend modifier. If the modifier value is non-`nil`, idling is turned on and the rotation type is set to the current incremental transformation. The `:do-idle` method for the rotating plot applies this matrix with the `:apply-transformation` method each time it is sent.

10.2.3 Graphical Function Input

There are a number of situations in which a procedure needs a positive-valued function as input. One example is the elicitation of a prior density on a real-valued quantity. In some cases it is sufficient to require that this function be specified as a procedure or an expression, but in other cases it may be more convenient to allow the function to be specified graphically. A tool for graphically specifying a function should

- allow the current function to be set and retrieved,

- force the input to produce a genuine function, that is, prevent multiple y values for a single x value.

It may also be useful to force the function to be continuous.

To construct a graph for specifying a positive-valued function on the unit interval, we can start by constructing a plot that contains a series of 50 connected linestarts, with x values spaced equally in the unit interval and y values equal to zero:

```
(setf p (plot-lines (rseq 0 1 50) (repeat 0 50)))
```

If we are primarily interested in the shape of the function, we can remove the y axis with the expression

```
(send p :y-axis nil)
```

The initial linestarts in the graph represent a function that is identically zero. To allow this function to be changed with the mouse, we can specify a new mouse mode as

```
(send p :add-mouse-mode 'drawing
      :title "Drawing"
      :cursor 'finger
      :click :mouse-drawing)
```

The expression

```
(send p :mouse-mode 'drawing)
```

puts the plot in the new mode.

The method for the click message `:mouse-drawing` has to take the x coordinate of a mouse click, use it to identify the linestart with the closest x coordinate, and change the y value of that linestart to the y value of the click. As the mouse is dragged, linestarts with x values crossed by the mouse need to have their y values adjusted as well. A simple first implementation of the `:mouse-drawing` method is given by

```
(defmeth p :mouse-drawing (x y m1 m2)
  (flet ((adjust (x y)
              (let* ((n (send self :num-lines))
                     (reals (send self :canvas-to-real x y))
                     (i (x-index (first reals) n))
                     (y (second reals)))
                (send self :linestart-coordinate 1 i y)
                (send self :redraw-content))))
    (adjust x y)
    (send self :while-button-down #'adjust)))
```

with the function `x-index` defined as

```
(defun x-index (x n)
  (max 0 (min (- n 1) (floor (* n x)))))
```

This definition works if the mouse is clicked and dragged slowly. But if the mouse is moved quickly, it may skip over several linestarts, resulting in a function with spikes. To avoid this problem, we can use a slightly more elaborate :mouse-drawing method that interpolates linearly between points passed to the button down action function. The function

```
(defun interpolate (x a b ya yb)
  (let* ((range (if-else (/= a b) (- b a) 1))
         (p (pmax 0 (pmin 1 (abs (/ (- x a) range))))))
    (+ (* p yb) (* (- 1 p) ya))))
```

linearly interpolates the y value at the x argument between the points (a, ya) and (b, yb) . The value a can be greater than the value b , and all arguments can be compound. The `if-else` expression is needed to avoid a division by zero if a is equal to b . Using the `interpolate` function, we can define the :mouse-drawing method as

```
(defmeth p :mouse-drawing (x y m1 m2)
  (let* ((n (send self :num-lines))
         (reals (send self :canvas-to-real x y))
         (old-i (x-index (first reals) n))
         (old-y (second reals)))
    (flet ((adjust (x y)
              (let* ((reals (send self :canvas-to-real x y))
                     (new-i (x-index (first reals) n))
                     (new-y (second reals))
                     (i (iseq old-i new-i))
                     (yvals (interpolate
                             i old-i new-i old-y new-y)))
                (send self :linestart-coordinate 1 i yvals)
                (send self :redraw-content)
                (setf old-i new-i)
                (setf old-y new-y))))
      (adjust x y)
      (send self :while-button-down #'adjust))))
```

The local function `adjust` uses the variables `old-i` and `old-y` in the surrounding environment to hold the values corresponding to the last call to `adjust`. Linestarts between the previous and current mouse locations are interpolated linearly.

The linestarts of the plot `p` contain the values of its function at a grid of x values. The message :lines, with a method defined by

```
(defmeth p :lines ()
  (let ((i (iseq (send self :num-lines))))
    (list (send self :linestart-coordinate 0 i)
          (send self :linestart-coordinate 1 i))))
```

can be used to retrieve the current function values as a list of lists of x and y values.

Exercise 10.7

Set up a menu item that presents a dialog for saving the current lines in the plot in a global variable.

Exercise 10.8

Use the :lines message to allow the most recent mouse action to be undone.

Exercise 10.9

Define a method for changing the number of points in the x axis grid. Try to preserve the current function by interpolating new y values from the current ones.

10.3 Plot Control Overlays

Plot actions can be controlled from menus, dialogs, or controls placed in overlays on the plot itself. This section describes two simple overlay control prototypes, and shows how they can be used to give additional functionality to a rotating plot.

10.3.1 A Press Button Control

A press button consists of a small square, representing a button, and a label string drawn to the right of the square. When the mouse is clicked in the square, thus pressing the button, the square is highlighted, and an action is called continuously until the button is released.

The Press Button Prototype

The press button prototype is defined by

```
(defproto button-overlay-proto
  '(location title)
  nil
  graph-overlay-proto)
```

Accessors for the two slots are defined as

```
(defmeth button-overlay-proto :location (&optional new)
  (if new (setf (slot-value 'location) new))
  (slot-value 'location))
```

and

```
(defmeth button-overlay-proto :title (&optional new)
  (if new (setf (slot-value 'title) new))
  (slot-value 'title))
```

Neither accessor method does any error checking, and neither attempts to redraw the control if its slot value is modified.

The accessors can be used to give reasonable initial values to the prototype's location slot

```
(send button-overlay-proto :location '(0 0))
```

and title slot

```
(send button-overlay-proto :title "Button")
```

The location can be changed by a :resize method.

To aid in positioning the button, we need to be able to determine the size of a rectangle surrounding the button. The location slot is assumed to contain the coordinates of the left top corner of this rectangle. The size of the rectangle depends on the text font of the graph containing the button overlay. The button itself is a square with sides equal to the ascent of the font. A gap half the size of the font ascent is placed between the button and the title string. A margin of the same size as the gap is placed around the string and button. Using this layout, a :size method that returns a list of the width and height of the rectangle can be defined as

```
(defmeth button-overlay-proto :size ()
  (let* ((graph (send self :graph))
         (title (send self :title))
         (text-width (send graph :text-width title))
         (side (send graph :text-ascent))
         (gap (floor (/ side 2))))
    (descent (send graph :text-descent))
    (height (+ side descent (* 2 gap))))
  (list (+ side (* 3 gap) text-width) height)))
```

Based on the layout just described, a method to return a list of the rectangle coordinates of the button square is given by

```
(defmeth button-overlay-proto :button-box ()
  (let* ((graph (send self :graph))
         (loc (send self :location))
         (side (send graph :text-ascent))
         (gap (floor (/ side 2))))
    (list (+ gap (first loc)) (+ gap (second loc)) side side)))
```

The method defined as

```
(defmeth button-overlay-proto :title-start ()
  (let* ((graph (send self :graph))
         (loc (send self :location))
         (title (send self :title))
         (side (send graph :text-ascent))
         (gap (floor (/ side 2))))
    (list (+ (* 2 gap) side (first loc))
          (+ gap side (second loc)))))
```

calculates the location at which to draw the title string.

A method to draw the button is given by

```
(defmeth button-overlay-proto :draw-button (&optional paint)
  (let ((box (send self :button-box))
        (graph (send self :graph)))
    (apply #'send graph :erase-rect box)
    (if paint
        (apply #'send graph :paint-rect box)
        (apply #'send graph :frame-rect box))))
```

This method takes an optional argument. If the argument is non-nil, the button is drawn in a highlighted state by painting it. Otherwise, it is simply framed. The apply function is needed since the box variable contains a list of the arguments needed by the rectangle drawing methods.

A method to draw the title can be defined as

```
(defmeth button-overlay-proto :draw-title ()
  (let ((graph (send self :graph))
        (title (send self :title))
        (title-xy (send self :title-start)))
    (apply #'send graph :draw-string title title-xy)))
```

Using these two methods, the :redraw method is

```
(defmeth button-overlay-proto :redraw ()
  (send self :draw-title)
  (send self :draw-button))
```

To determine whether the overlay should respond to a click, we need to determine whether the click location is contained in the button square. This can be done with a method defined as

```
(defmeth button-overlay-proto :point-in-button (x y)
  (let* ((box (send self :button-box))
         (left (first box))
         (top (second box))
         (side (third box)))
    (and (< left x (+ left side)) (< top y (+ top side)))))
```

A button overlay is assumed to have a :do-action method that carries out the button's action. The action may want to make use of the modifiers for the click. The action may also want to act differently on the initial click than on subsequent calls while the button is held down. To allow for this possibility, we can adopt the convention that the :do-action message is sent with one argument. On the first call after a click, the argument is a list of the two modifiers. On subsequent calls, the argument is nil. The :do-click method can thus be defined as

```
(defmeth button-overlay-proto :do-click (x y m1 m2)
  (let ((graph (send self :graph)))
    (when (send self :point-in-button x y)
      (send self :draw-button t)
      (send self :do-action (list m1 m2))
      (send graph :while-button-down
        #'(lambda (x y) (send self :do-action nil))
        nil)
      (send self :draw-button nil)
      t)))
```

Since the :while-button-down message is sent with a second argument with value nil, its action function is called continuously while the button is held down.

To complete the definition of the push button prototype, we can give it a default :do-action method that does nothing:

```
(defmeth button-overlay-proto :do-action (x) nil)
```

An Application: Rocking a Rotatable Plot

A problem with rotating plots on a computer is that the depth illusion created by motion is lost when the rotation stops. On the other hand, as the plot rotates, it is hard to focus on one particular view. One solution to this problem is to allow the plot to be rocked back and forth, rotating it by a small amount in each direction around the vertical screen axis. The rocking motion provides the depth illusion, but the view of the data remains essentially unchanged. A method for rocking a rotatable plot can be defined as

```
(defmeth spin-proto :rock-plot (&optional (a .15))
  (let* ((angle (send self :angle))
         (k (round (/ a angle))))
    (dotimes (i k) (send self :rotate-2 0 2 angle))
    (dotimes (i (* 2 k)) (send self :rotate-2 0 2 (- angle)))
    (dotimes (i k) (send self :rotate-2 0 2 angle))))
```

This method rotates by a specified angle in one direction, rotates by twice the angle in the opposite direction, and then rotates back to the original position. The default angle used is 0.15 radians.

A press button provides a convenient way to send the :rock-plot message to a rotating plot. Using the press button prototype, we can define a new prototype for rocking a rotating plot as

```
(defproto spin-rock-control-proto () () button-overlay-proto)
```

The title for the button can be set by

```
(send spin-rock-control-proto :title "Rock Plot")
```

and the action method for the rocker button sends its plot the :rock-plot message.

```
(defmeth spin-rock-control-proto :do-action (first)
  (send (send self :graph) :rock-plot))
```

A natural location for the button is along the bottom of the plot, together with the standard rotation controls. The :resize method

```
(defmeth spin-rock-control-proto :resize ()
  (let* ((graph (send self :graph))
         (size (send self :size))
         (width (send graph :canvas-width))
         (height (send graph :canvas-height)))
    (send self :location (- (list width (+ 3 height)) size))))
```

sets the button's location to place the button at the right lower corner of the plot. The 3 pixel height adjustment is needed to align the button with the standard controls on the Macintosh.

As an example, using the abrasion loss data introduced in Section 2.5.1, the expression

```
(let ((w (spin-plot
            (list hardness tensile-strength abrasion-loss)))
      (b (send spin-rock-control-proto :new)))
  (send w :add-overlay b)
  (send b :resize)
  (send b :redraw))
```

constructs a rotating plot and installs a rocking button in the plot.

10.3.2 Two-Button Controls

The Two-Button Prototype

The standard controls on a rotating plot are two-button controls, with one button used for positive angles and one for negative angles. A prototype for a two-button control can be constructed from scratch, like the press button prototype. But there is considerable similarity between a press button and a two-button control, so we can save some effort by defining the new prototype to inherit from the press button control:

```
(defproto twobutton-control-proto () () button-overlay-proto)
```

This use of inheritance is similar to defining nonlinear regression models in terms of linear models.

The layout of a two-button control is similar to the layout of a press button. It merely adds an additional button. The :size method can thus use the inherited method and add room for another button and a space between the buttons:

```
(defmeth twobutton-control-proto :size ()
  (let* ((graph (send self :graph))
         (size (call-next-method))
         (side (send graph :text-ascent))
         (gap (floor (/ side 2))))
    (list (+ gap side (first size)) (second size))))
```

The :title-start method could also use the inherited method, but it is easy enough to define from scratch:

```
(defmeth twobutton-control-proto :title-start ()
  (let* ((graph (send self :graph))
         (loc (send self :location))
         (title (send self :title))
         (side (send graph :text-ascent))
         (gap (floor (/ side 2))))
    (list (+ (* 3 gap) (* 2 side) (first loc))
          (+ gap side (second loc)))))
```

Since there are now two buttons, we need a way to distinguish between them. The symbols - and + will be used for the left and right buttons, respectively. The :button-box method now takes one argument, the button symbol:

```
(defmeth twobutton-control-proto :button-box (which)
  (let* ((graph (send self :graph))
         (loc (send self :location))
         (side (send graph :text-ascent))
         (gap (floor (/ side 2)))
         (left (case which
                  (+ (+ gap (first loc)))
                  (- (+ (* 2 gap) side (first loc))))))
    (list left (+ gap (second loc)) side side)))
```

The :draw-button method defined as

```
(defmeth twobutton-control-proto :draw-button
  (which &optional paint)
  (let ((box (send self :button-box which)))
```

```
(graph (send self :graph)))
(cond (paint (apply #'send graph :paint-rect box))
      (t (apply #'send graph :erase-rect box)
           (apply #'send graph :frame-rect box))))
```

also requires a button symbol argument, and takes an optional argument to specify whether the button is to be highlighted. The :redraw method is now given by

```
(defmeth twocontrol-proto :redraw ()
  (send self :draw-title)
  (send self :draw-button '-')
  (send self :draw-button '+))
```

The :point-in-button method returns `nil` if the click coordinates do not fall in one of the buttons. If they do fall in a button, the button symbol is returned:

```
(defmeth twocontrol-proto :point-in-button (x y)
  (let* ((box1 (send self :button-box '-))
         (box2 (send self :button-box '+))
         (left1 (first box1))
         (top (second box1))
         (side (third box1))
         (left2 (first box2)))
    (cond
      ((and (< left1 x (+ left1 side)) (< top y (+ top side)))
       '-')
      ((and (< left2 x (+ left1 side)) (< top y (+ top side)))
       '+))))
```

The click method can be defined as

```
(defmeth twocontrol-proto :do-click (x y m1 m2)
  (let ((graph (send self :graph))
        (which (send self :point-in-button x y)))
    (when which
      (send self :draw-button which t)
      (send self :do-action which (list m1 m2))
      (send graph :while-button-down
            #'(lambda (x y)
                (send self :do-action which nil))
            nil)
      (send self :draw-button which nil)
      t)))
```

The :do-action message is sent with a button symbol and a second argument that consists of a list of the modifiers on the first call and is `nil` on subsequent calls. A default :do-action method is given by

```
(defmeth twobutton-control-proto :do-action (which mods) nil)
```

An Application: Rotation around Coordinate Axes

The standard controls for a rotating plot allow the plot to be rotated around the screen axes. At times it is useful to rotate around one of the coordinate axes. This is only meaningful in three dimensions, since an axis and an angle do not uniquely specify a rotation in higher dimensions.

The method

```
(defmeth spin-proto :set-axis-rotation (v)
  (let* ((m (send self :num-variables))
         (v1 (if (= v 0) 1 0))
         (v2 (if (= v 2) 1 2))
         (trans (send self :transformation))
         (cols (column-list
                  (if trans trans (identity-matrix m))))
         (x1 (select cols v1))
         (x2 (select cols v2))
         (angle (send self :angle)))
    (send self :rotation-type (make-rotation x1 x2 angle))))
```

sets the `:rotation-type` of a plot to a matrix that rotates by the current angle around the axis specified by the index argument `v`. A two-button prototype for rotating around a specified axis is defined by

```
(defproto spin-rotate-control-proto
  '(v) () twobutton-control-proto)
```

The slot `v` holds the index of the rotation axis, and an `:isnew` method defined as

```
(defmeth spin-rotate-control-proto :isnew (v &rest args)
  (apply #'call-next-method :v v args))
```

takes an index as its argument and installs the index in the slot `v`, using the inherited `:isnew` method.

The title string for an axis rotation button can be read from the graph by using the `:variable-label` message:

```
(defmeth spin-rotate-control-proto :title ()
  (send (send self :graph) :variable-label (slot-value 'v)))
```

The `:do-action` method defined by

```
(defmeth spin-rotate-control-proto :do-action (sign mods)
  (let ((graph (send self :graph)))
    (if mods
        (let ((v (slot-value 'v)))
```

```

        (angle (abs (send graph :angle)))
        (send graph :idle-on (first mods))
        (send graph :angle
              (if (eq sign '+) angle (- angle)))
        (send graph :set-axis-rotation v)))
    (send graph :rotate)))

```

sets the axis rotation on a click, and sends the :rotate message on subsequent calls. The sign of the angle is adjusted according to the button pressed, and idling is turned on if the extend modifier is provided with the click.

Using the abrasion loss data once more, the expression

```

(flet ((width (c) (first (send c :size))))
  (let* ((w (spin-plot
              (list hardness tensile-strength abrasion-loss)))
         (c0 (send spin-rotate-control-proto :new 0))
         (c1 (send spin-rotate-control-proto :new 1))
         (c2 (send spin-rotate-control-proto :new 2)))
    (send w :add-overlay c0)
    (send w :add-overlay c1)
    (send w :add-overlay c2)
    (let ((width (max (mapcar #'width (list c0 c1 c2))))
          (height (second (send c0 :size)))
          (margin (send w :margin)))
      (send c1 :location (list 0 height))
      (send c2 :location (list 0 (* 2 height)))
      (send w :margin width 0 0 (fourth margin))))))

```

sets up a rotating plot and gives it three axis rotation controls, one for each data axis. The new controls are placed in the left margin of the plot. The :margin method sends the plot the :resize and :redraw messages. The methods for these messages send the corresponding messages to the overlays. The resulting plot is shown in Figure 10.4.

10.4 Grand Tours

A method recently proposed for exploring multi-dimensional data is the *grand tour* [4,16]. The basic idea is to find a sequence of one-, two-, or three-dimensional projections of the data that rapidly becomes dense among all possible projections. This sequence can then be examined for particularly “unusual” views of the data, such as views that exhibit clustering or other structure. The examination can be done numerically by calculating a statistic for the projection, or graphically by displaying the image of the projection. For the numerical approach, the sequence of projections does not need to have any additional structure. For the graphical approach, it should be continuous

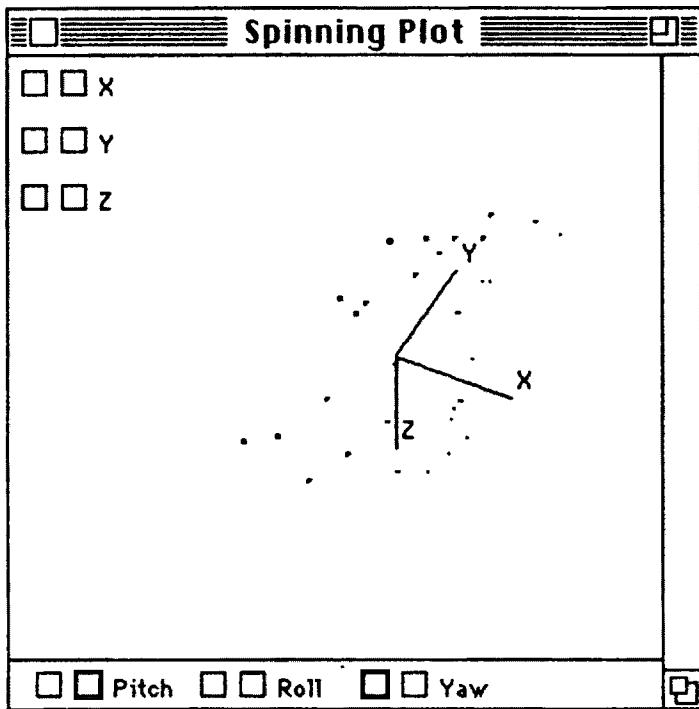


Figure 10.4: Rotating plot of the abrasion loss data with axis rotation controls.

in order to allow the viewer to easily follow individual points and groups of points.

One way to implement an m -dimensional graphical grand tour in Lisp-Stat is to choose a series of rotations and watch the plot as these rotations are applied. A scheme closely related to the geodesic grand tour [4,16] works as follows: Choose two points on the m -dimensional unit sphere. Use these two points to define a plane, and construct a series of rotations that takes the first point into the second while keeping the orthogonal complement of the plane fixed. When the second point is reached, repeat the process with another pair of points. A slight variation on this scheme uses the two points to determine the rotation plane. A rotation angle α is used in constructing the incremental rotation, and this rotation is applied a random number of times before switching to a new rotation. The number of times to apply the rotation is chosen uniformly from the nonnegative integers in the interval $[0, \pi/2\alpha]$.

This scheme can be implemented using several of the basic prototypes. The rotating plot already provides controls for speed and a system for applying incremental rotations. The histogram, on the other hand, gives a better

view of the density of points in a one-dimensional projection and may thus be more useful for detecting departures from normality. Instead of implementing all features of the tour separately for tours based on the rotating plot and on the histogram, we can take advantage of the multiple inheritance capability of the object system by defining a tour *mixin* to hold the basic methods and slot specifications needed for a tour. A tour prototype based on the rotating plot can then be constructed to have the mixin and *spin-proto* as its parents.

To implement the strategy outlined above, a tour plot uses the *make-rotation* function to construct a matrix for rotating from one point to another. The rotation uses an angle determined by the current rotation speed. The rotation needs to be applied a certain number of times to map the first point on the sphere into the second. When the second point is reached, a new incremental rotation is generated and the number of applications of the rotation is calculated. The tour mixin prototype can thus be defined as

```
(defproto tour-mixin '(tour-count tour-trans))
```

The two slots hold the incremental rotation matrix and the number of applications required. The mixin has no parents, since it is intended to be used in conjunction with other plot prototypes.

The touring process is to run continuously, so it can be implemented by defining the *:do-idle* method as

```
(defmeth tour-mixin :do-idle () (send self :tour-step))
```

The *:tour-step* method is the main part of the system:

```
(defmeth tour-mixin :tour-step ()
  (when (< (slot-value 'tour-count) 0)
    (flet ((sphere-rand (m)
      (let* ((x (normal-rand m))
             (nx2 (sum (^ x 2))))
        (if (< 0 nx2)
            (/ x (sqrt nx2))
            (/ (repeat 1 m) (sqrt m)))))))
      (let* ((m (send self :num-variables))
             (angle (send self :angle))
             (max (+ 1 (abs (floor (/ pi (* 2 angle)))))))
        (setf (slot-value 'tour-count) (random max))
        (setf (slot-value 'tour-trans)
              (make-rotation (sphere-rand m)
                            (sphere-rand m)
                            angle))))
      (send self :apply-transformation (slot-value 'tour-trans))
      (setf (slot-value 'tour-count)
            (- (slot-value 'tour-count) 1))))
```

This method checks the `tour-count` slot to see if it is below zero. If it is not, its value is decremented by one and the current value of the `tour-trans` slot is applied. If the `tour-count` slot value is below zero, the method first calculates a new transformation and count. The local function `sphere-rand` is used to generate uniform points on the m -dimensional unit sphere by normalizing a list of m independent standard normal random variables. To be safe, it checks for a division by zero. Two points are generated using `sphere-rand`, an angle is obtained by sending the plot the `:angle` message, and the results are passed to the `make-rotation` function to construct the new incremental rotation matrix. The new count is constructed with the `random` function.

For this method to work properly, the plot must have an `:angle` method. Since the `spin-proto` plot already has such a method, it is not included in the `tour-mixin`. Tour plots based on other prototypes need to add their own `:angle` methods.

To ensure that a new transformation and count are calculated the first time the `:tour-step` method is used, we can set the value of the `tour-count` slot to a negative number, such as

```
(send tour-mixin :slot-value 'tour-count -1)
```

To turn touring on and off, or to determine if touring is turned on, we can define a `:tour-on` method. The simplest version of this method passes its arguments, if any, to the `:idle-on` message:

```
(defmeth tour-mixin :tour-on (&rest args)
  (apply #'send self :idle-on args))
```

At a later time we may wish to use a more elaborate definition.

To allow us to add an item for turning the tour on and off to the plot menu, we can define a `tour-item-proto` prototype as

```
(defproto tour-item-proto '(graph) () menu-item-proto)
```

The `:isnew` method for this prototype requires a `graph` object as its argument:

```
(defmeth tour-item-proto :isnew (graph)
  (call-next-method "Touring")
  (setf (slot-value 'graph) graph))
```

The `graph` can be retrieved with the `:graph` message:

```
(defmeth tour-item-proto :graph () (slot-value 'graph))
```

The `:update` method for the item places a check mark in front of the item if touring is turned on,

```
(defmeth tour-item-proto :update ()
  (let ((graph (send self :graph)))
    (send self :mark (send graph :tour-on))))
```

and the :do-action method toggles the touring action:

```
(defmeth tour-item-proto :do-action ()
  (let* ((graph (send self :graph))
         (is-on (send graph :tour-on)))
    (send graph :tour-on (not is-on))))
```

Finally, we can redefine the :menu-template method for the tour mixin to add a tour item to the end of the menu template produced by the inherited method:

```
(defmeth tour-mixin :menu-template ()
  (append (call-next-method)
          (list (send tour-item-proto :new self))))
```

Using the tour mixin, a tour plot prototype based on the rotating plot can be defined as

```
(defproto spin-tour-proto () () (list tour-mixin spin-proto))
```

This prototype has two parents, the tour mixin and `spin-proto`. The mixin is placed ahead of `spin-proto`, so its methods appear before methods obtained from the rotating plot in the precedence list. We can give the new prototype more suitable window and menu titles with the expressions

```
(send spin-tour-proto :title "Grand Tour")
```

and

```
(send spin-tour-proto :menu-title "Tour")
```

A constructor function for producing a tour plot for a set of data is defined by

```
(defun tour-plot (data &rest args &key point-labels)
  (let ((graph (apply #'send spin-tour-proto :new
                        (length data) args)))
    (if point-labels
        (send graph :add-points
              data :point-labels point-labels :draw nil)
        (send graph :add-points data :draw nil))
    (send graph :adjust-to-data :draw nil)
    graph))
```

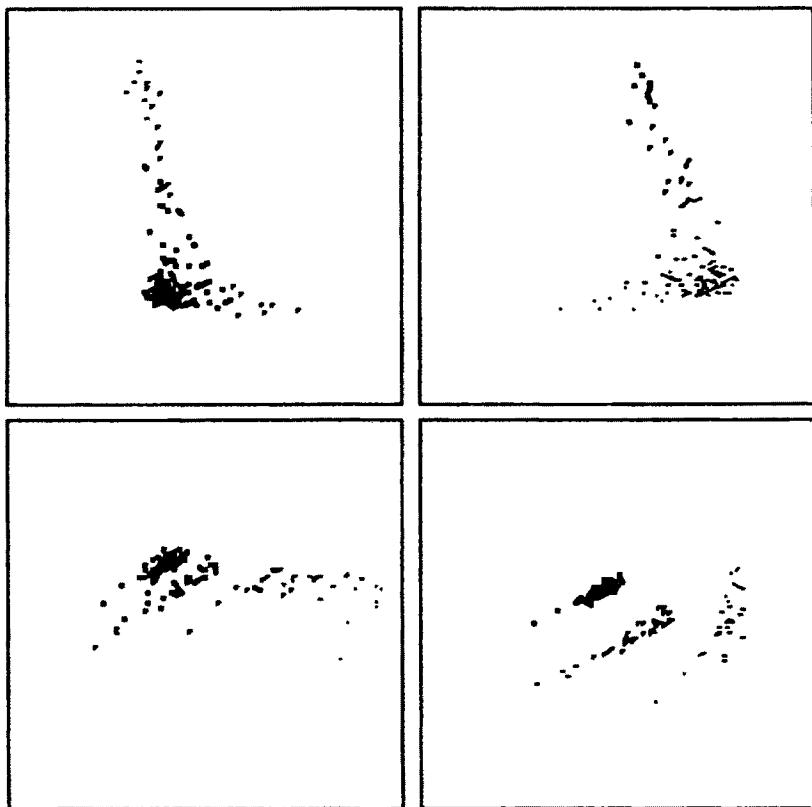


Figure 10.5: Four views from a grand tour of the diabetes data.

This function produces a new graph by sending the prototype the :new message, adds the data to the plot, and scales the plot to the data with the :adjust-to-data message. Since the tour mixin does not have an :isnew method, the :isnew method inherited from spin-proto is used.

Grand tours appear to be very useful for detecting clusters in multi-dimensional space. As an artificial illustration, Figure 10.5 shows four views of a grand tour of a data set from Reaven and Miller [28,52]. The data used in the plot include three continuous measurements on each of 150 patients, two glucose measurements and a measure of insulin tolerance. The fourth variable is a categorical variable with three levels to indicate whether the patient is classified as normal, having “chemical” diabetes, or having “overt” diabetes. The first two views in Figure 10.5 show the “boomerang” shape of the point cloud in the three continuous variables. The two remaining views show the separation of the cloud into three clusters as the fourth categorical variable is brought into the tour.

Using the tour plot mixin, we can also construct a histogram tour prototype. The definition

```
(defproto hist-tour-proto
  '(angle) () (list tour-mixin histogram-proto))
```

adds an angle slot. An accessor method for the slot is given by

```
(defmeth hist-tour-proto :angle (&optional new)
  (if new (setf (slot-value 'angle) new))
  (slot-value 'angle))
```

and an initial value can be installed as

```
(send hist-tour-proto :angle .1)
```

The default scale option for this prototype should be variable:

```
(send hist-tour-proto :scale-type 'variable)
```

After giving the prototype a new window title

```
(send hist-tour-proto :title "Histogram Tour")
```

and a new menu title,

```
(send hist-tour-proto :menu-title "Tour")
```

we can define a constructor function as

```
(defun histogram-tour (data &rest args &key point-labels)
  (let ((graph (apply #'send hist-tour-proto :new
                           (length data) :draw nil args)))
    (if point-labels
        (send graph :add-points
```

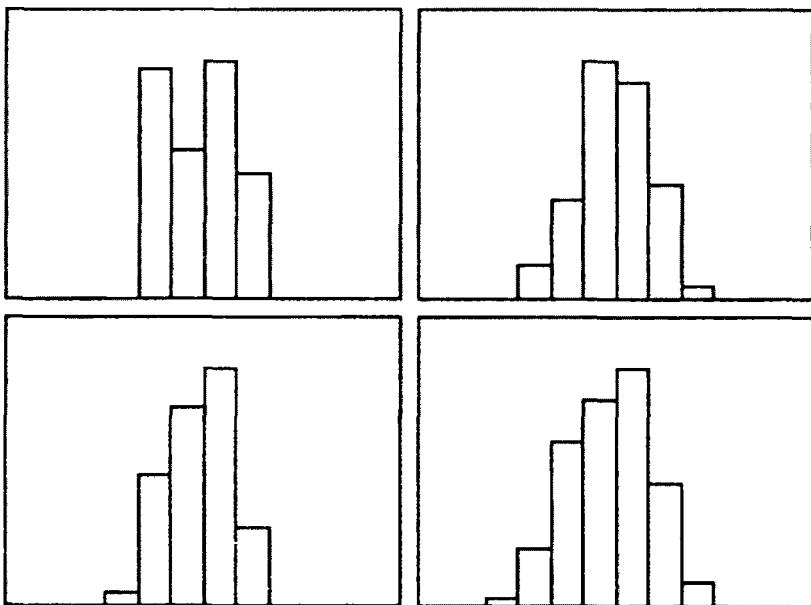


Figure 10.6: Four views from a histogram grand tour of 100 points distributed uniformly in the six-dimensional unit cube.

```

data :point-labels point-labels :draw nil)
  (send graph :add-points data :draw nil))
  (send graph :adjust-to-data :draw nil)
graph))

```

An interesting point to explore with a histogram tour is the theoretical observation that most projections of uniform data in m -dimensional space “look” normal for large m [27]. The expression

```
(histogram-tour (uniform-rand (repeat 100 6)))
```

constructs a histogram tour of 100 points distributed uniformly in the six-dimensional unit cube. Figure 10.6 shows four views of the tour. The first view shows a histogram of the first coordinate and looks reasonably uniform. The other views are taken at various points in the tour and look more normal than uniform.

A similar approach can be used to construct a tour plot based on the scatterplot matrix. A scatterplot matrix tour provides simultaneous views of all coordinate pairs of the current rotation.

Exercise 10.10

Implement the scatterplot matrix tour prototype.

Exercise 10.11

Instead of using a menu item to start and stop a tour, it may be more convenient to use a button in an overlay. Give the tour mixin an `:isnew` method that adjusts the plot margin and installs such a button overlay.

Exercise 10.12

As you watch a tour run, you may want to stop and “rewind” it to show a view that looked interesting. Write a method that reverses the current rotation one step at a time, and design an overlay with a button for stopping the tour and reversing the current rotation while the button is held down.

Exercise 10.13

Using the plot developed in Exercise 10.4, design a tour plot that shows a kernel density estimate of the first coordinate of the transformed data.

Exercise 10.14

Buja et al. [16] describe another touring strategy called a *correlation tour*. Implement a correlation tour mixin, and use it to construct several correlation tour prototypes.

10.5 Parallel Coordinate Plots

Another method for displaying data in three or more dimensions is to use a parallel coordinate plot [13.35]. A parallel coordinate plot is constructed by placing equally spaced parallel vertical axes on a plot, marking off the values of each point on each axis, and connecting the marks for each point. As an illustration, Figure 10.7 shows a parallel coordinate plot of the stack loss data. The four axes, which are not drawn, are at the kinks in the lines. Points in the plot can be selected by selecting or brushing their point symbols. The point symbols are initially located at the first axis, but can be positioned at any one of the axes with a dialog brought up by the plot menu. This layout is based on a similar plot used by Andreas Buja and Paul Tukey.

A prototype for a parallel coordinate plot can be developed, starting with the `graph-proto` prototype. The new prototype needs one additional slot to hold the index of the current axis, the axis containing the point symbols:

```
(defproto parallel-plot-proto '(v) () graph-proto)
```

A new title for a parallel plot window is installed by

```
(send parallel-plot-proto :title "Parallel Plot")
```

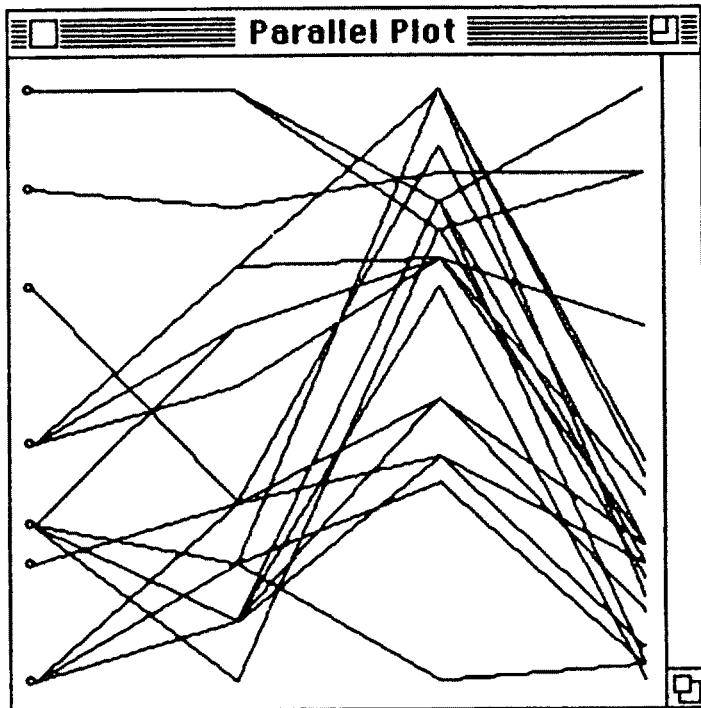


Figure 10.7: A parallel coordinate plot of the stack loss data.

Like a histogram, a parallel plot adds an additional dimension to the data. This dimension is used to position the point symbols along the horizontal axis. The :isnew method

```
(defmeth parallel-plot-proto :isnew (m &rest args)
  (setf (slot-value 'v) 0)
  (apply #'call-next-method (+ 1 m) args)
  (send self :content-variables m 0))
```

adds 1 to its dimension argument, sets the current axis slot to 0, and sets the initial content variables to be the last and first dimensions. As long as appropriate values are installed as the coordinates for the last dimension, this setup ensures that the graph-proto methods draw the point symbols in the appropriate locations and handle the standard mouse actions. The remaining methods for the parallel plot are needed to ensure that the proper data are installed, and to take care of drawing the lines.

Installing the right point coordinate values is the responsibility of the method for accessing the current axis holding the point symbols. Called without an argument, this method returns the current axis index. Called with an index i specifying a new axis, it sets the value of the point coordinates

of the last variable to *i*, and sets the content variables to the last variable and variable *i*:

```
(defmeth parallel-plotproto :current-axis
  (&optional (i nil set) &key (draw t))
  (when set
    (setf (slot-value 'v) i)
    (let* ((n (send self :num-points))
           (m (- (send self :num-variables) 1))
           (i (max 0 (min i (- m 1)))))
      (if (< 0 n)
          (send self :point-coordinate m (iseq n) i))
      (send self :content-variables m i)))
    (if draw (send self :redraw)))
  (slot-value 'v))
```

The :draw keyword argument can be used to prevent the method from redrawing the plot. A dialog for switching the current axis is presented by

```
(defmeth parallel-plotproto :choose-current-axis ()
  (let* ((choices
            (mapcar #'(lambda (x) (format nil "~d" x))
                    (iseq (- (send self :num-variables) 1))))
         (v (choose-item-dialog
               "Current Axis:"
               choices
               :initial (send self :current-axis))))
    (if v (send self :current-axis v))))
```

A menu item for presenting the dialog can be added to the standard menu by modifying the :menu-template method:

```
(defmeth parallel-plotproto :menu-template ()
  (flet ((action () (send self :choose-current-axis)))
    (let ((item (send menu-itemproto :new
                      "Current Variable"
                      :action #'action)))
      (append (call-next-method) (list item)))))
```

The :adjust-to-data method uses the inherited method, and then adjusts the range of the last variable to leave 0.1 units of space to either side of the first and last axes. If the scale type is **nil**, the ranges of the data variables are expanded by 10%:

```
(defmeth parallel-plotproto :adjust-to-data (&key (draw t))
  (call-next-method :draw nil)
  (let ((m (- (send self :num-variables) 1)))
    (if (null (send self :scale-type))
```

```
(flet ((expand-range (i)
  (let* ((range (send self :range i))
    (mid (mean range))
    (half (- (second range) (first range)))
    (low (- mid (* .55 half)))
    (high (+ mid (* .55 half))))
  (send self :range i low high :draw nil))))
  (dotimes (i m) (expand-range i))))
  (send self :scale m 1 :draw nil)
  (send self :center m 0 :draw nil)
  (send self :range m -.1 (- m .9) :draw draw)))
```

The `:add-points` method needs to add an additional coordinate to new point data before calling the inherited method. As long as the inherited method does not attempt to draw the points, the actual values do not matter and can be set to the proper value with the `:current-axis` message:

```
(defmeth parallel-plotproto :add-points (data &key (draw t))
  (let ((n (length (first data))))
    (call-next-method (append data (list (repeat 0 n)))
      :draw nil))
  (send self :current-axis
    (send self :current-axis) :draw draw)))
```

The `:add-lines` message can be overridden by

```
(defmeth parallel-plotproto :add-lines (&rest args)
  (error "Lines are not meaningful for this plot"))
```

since line data cannot be displayed reasonably in a parallel coordinate plot.

The `:redraw-content` method can be defined several ways. The most efficient approach is to take advantage of the canvas coordinates used by the `graph-proto` transformation system. Defining the `:resize` method as

```
(defmeth parallel-plotproto :resize ()
  (call-next-method)
  (let ((height (fourth (send self :content-rect)))
    (m (- (send self :num-variables) 1)))
  (send self :canvas-range (iseq m) 0 height)))
```

ensures that the canvas ranges of all data variables are set to the range from 0 to the height of the content rectangle. The inherited method, called at the beginning of the new method, ensures that the current `x` variable has its range set to the range from 0 to the width of the content rectangle.

The lines for the parallel representation of a point can be drawn as a polygon. The `x` coordinates are computed as offsets to the content origin based on the convention used in setting the current axis. The `y` coordinates are the point canvas coordinates viewed as offsets to the content origin. On a

color display, the lines are drawn in the point's color. A method for drawing the lines for one or more points can thus be defined as

```
(defmeth parallel-plotproto :draw-parallel-point (i)
  (let* ((points (if (numberp i) (list i) i))
         (width (third (send self :content-rect)))
         (origin (send self :content-origin))
         (x-origin (first origin))
         (y-origin (second origin))
         (m (- (send self :num-variables) 1))
         (gap (/ width (+ (- m 1) .2))))
    (xvals (+ x-origin
               (round (* gap (+ .1 (iseq 0 (- m 1)))))))
    (indices (iseq 0 (- m 1)))
    (oldcolor (send self :draw-color)))
  (dolist (i points)
    (if (send self :point-showing i)
        (let* ((color (send self :point-color i))
               (yvals (- y-origin
                           (send self
                                 :point-canvas-coordinate
                                 indices
                                 i)))
               (poly (transpose (list xvals yvals))))
          (if color (send self :draw-color color))
          (send self :frame-poly poly)
          (if color (send self :draw-color oldcolor)))))))
```

The argument to the method can be a single index or a list of indices. The :redraw-content method can now be defined as

```
(defmeth parallel-plotproto :redraw-content ()
  (let ((indices (iseq (send self :num-points))))
    (send self :start-buffering)
    (call-next-method)
    (send self :draw-parallel-point indices)
    (send self :buffer-to-screen)))
```

Finally, we can define a constructor function as

```
(defun parallel-plot (data &rest args &key point-labels)
  (let ((graph (apply #'send parallel-plotproto :new
                        (length data) :draw nil args)))
    (if point-labels
        (send graph :add-points
              data :point-labels point-labels :draw nil)
        (send graph :add-points data :draw nil)))
```

```
(send graph :adjust-to-data :draw nil)
graph))
```

The expression

```
(parallel-plot (list air temp conc loss))
```

creates the plot shown in Figure 10.7.

Since the parallel plot prototype is built on the `graph-proto` prototype, it has access to the full transformation system. It is therefore possible to apply rotations, and to define a parallel version of the grand tour.

Exercise 10.15

Construct a prototype that implements a grand tour in parallel coordinates.

Exercise 10.16

The `:isnew` and `:choose-current-axis` methods defined above do not handle variable label strings properly. Improve the methods in this respect.

Exercise 10.17

Selecting and highlighting in the parallel plot only changes the point symbols. An alternative approach is to draw points in the `normal` state with dashed lines and points in the `hilited` or `selected` states with solid lines. Modify the `:draw-parallel-point` method, and write an `:adjust-screen-point` method to implement this idea.

10.6 An Alternative Linking Strategy

The default linking strategy used in Lisp-Stat creates a loose correspondence between points in different plots with a common index. Point states are matched across linked plots, but other characteristics, such as symbols or colors, are not. An alternative approach, used by McDonald [44] and Stuetzle [58], views observations as objects that can be viewed using different graphs. Properties like the point state, color, or symbol are properties of an observation. Any change in any of these properties, or in the values of variables in an observation, should be reflected in all plots viewing the observation. In this approach the order in which observations are put in a plot is not important, since observations have their own identity as objects and do not need to be identified by an index. This implies that different plots can show different sets of observations.

This section describes a simple implementation of this linking strategy within the Lisp-Stat graphics system. The implementation is based on the data representation described in Section 6.8.2. The basic idea is to represent

observations as objects, allow plots to be used to change characteristics of an observation object, and make sure that changes to an observation object are passed on to all plots that contain the observation.

A basic observation prototype is defined as

```
(defproto observation-proto '(label state symbol color views))
```

This augments the definition in Section 6.8.2 with slots for holding characteristics needed for plotting an observation. Reader methods for these slots are given by

```
(defmeth observation-proto :label () (slot-value 'label))
(defmeth observation-proto :state () (slot-value 'state))
(defmeth observation-proto :symbol () (slot-value 'symbol))
(defmeth observation-proto :color () (slot-value 'color))
```

Default state and symbol values are installed by

```
(send observation-proto :slot-value 'state 'normal)
(send observation-proto :slot-value 'symbol 'disk)
```

Observations for a particular data set can be constructed by adding slots for holding variable values to objects inheriting from this observation prototype.

The `views` slot in an observation is used to record which plots contain an observation, along with the index of the observation in a particular plot. A new entry is added to this list by sending the observation the `:add-view` message with two arguments, the plot object and the observation's index in the plot:

```
(defmeth observation-proto :add-view (graph key)
  (setf (slot-value 'views)
    (cons (list graph key) (slot-value 'views))))
```

The method

```
(defmeth observation-proto :delete-view (graph)
  (flet ((test (x y) (eq x (first y))))
    (let ((views (slot-value 'views)))
      (if (member graph views :test #'test)
        (setf (slot-value 'views)
          (delete graph views :test #'test))))))
```

deletes the entry for a particular graph from the `views` list. For efficiency, the destructive function `delete` is used. The current `views` list is returned by the reader method defined as

```
(defmeth observation-proto :views () (slot-value 'views))
```

To support the new linking system, slots in an observation object should only be changed by sending the observation the `:change` message with the slot symbol and the new slot value as arguments. The method defined as

```
(defmeth observation Proto :change (slot value)
  (setf (slot-value slot) value)
  (dolist (view (send self :views))
    (send (first view) :changed (second view) slot value)))
```

sends each view containing the object the :changed message with the observation index, the slot symbol and the new value as arguments.

The :changed message is part of the protocol for communication between observations and plots needed to support the new linking strategy. The method for this message and several other messages can be incorporated in a mixin:

```
(defproto observation-plot-mixin '(observations variables))
```

The mixin has two slots. The **observations** slot holds a vector of the observation objects in the plot. As in Section 6.8.2, variables shown in a plot are represented by message selector keywords. The **variables** slot holds a list of these keywords. Readers for the two slots are defined by

```
(defmeth observation-plot-mixin :observations ()
  (slot-value 'observations))
```

and

```
(defmeth observation-plot-mixin :variables ()
  (slot-value 'variables))
```

The :isnew method can be defined to require a list of variable keywords instead of a dimension number as its argument:

```
(defmeth observation-plot-mixin :isnew (vars &rest args)
  (apply #'call-next-method
    (length vars)
    :variable-labels (mapcar #'string vars)
    args)
  (setf (slot-value 'variables) vars))
```

Since this mixin is to be used with graph prototypes, the inherited :isnew method will require an integer argument specifying the dimension of the plot.

Observations can be added to a plot with the :add-observation message. This message requires a list of observations, and accepts the :draw keyword to specify whether the plot should be redrawn:

```
(defmeth observation-plot-mixin :add-observations
  (new-obs &key (draw t))
  (let* ((obs (send self :observations))
    (n (length obs))
    (m (length new-obs))
    (new-obs (coerce new-obs 'vector))))
```

```
(setf (slot-value 'observations)
      (concatenate 'vector obs new-obs))
(dotimes (i m)
  (send (aref new-obs i) :add-view self (+ i n)))
(send self :needs-adjusting t)
(if draw (send self :adjust-screen)))
```

When a graph receives the :remove message, for example, as a result of closing the window, the graph should remove itself as a view from its observations:

```
(defmeth observation-plot-mixin :remove ()
  (call-next-method)
  (let ((obs (send self :observations)))
    (dotimes (i (length obs))
      (send (aref obs i) :delete-view self))))
```

The adjust screen method checks if the plot needs to be adjusted. If it does, the current points are cleared, and new points are installed by referring to the observations in the plot:

```
(defmeth observation-plot-mixin :adjust-screen ()
  (if (send self :needs-adjusting)
      (let ((vars (send self :variables))
            (obs (send self :observations)))
        (send self :clear-points :draw nil)
        (when (< 0 (length obs))
          (flet ((variable (v)
                  (map-elements #'(lambda (x) (send x v))
                                obs)))
            (send self :add-points
                  (mapcar #'variable vars) :draw nil)))
        (dotimes (i (length obs))
          (let ((x (aref obs i)))
            (send self :point-label i (send x :label))
            (send self :point-state i (send x :state))
            (send self :point-color i (send x :color))
            (send self :point-symbol
                  i (send x :symbol))))))
      (send self :needs-adjusting nil)
      (send self :redraw-content)))
```

The method for the :changed message can be defined as

```
(defmeth observation-plot-mixin :changed (key what value)
  (case what
    (state (send self :point-state key value))
    (t (send self :needs-adjusting t))))
```

Since most plots can have their point states adjusted quickly, the `:point-state` message is used to respond to state changes. Other changes are handled by marking the plot for adjustment.

Several methods used by the standard plot menus need to be redefined to change the observation objects, instead of changing the plots, and then ensure that all plots viewing the objects are adjusted. The function

```
(defun synchronize-graphs ()
  (dolist (g (active-windows))
    (if (kind-of-p g observation-plot-mixin)
        (send g :adjust-screen))))
```

can be used to adjust all plots. Three of the methods used by the menus are the `:erase-selection` method

```
(defmeth observation-plot-mixin :erase-selection ()
  (let ((obs (send self :observations)))
    (dolist (i (send self :selection))
      (send (aref obs i) :change 'state 'invisible)))
  (synchronize-graphs))
```

for making selected points invisible, the `:show-all-points` method

```
(defmeth observation-plot-mixin :show-all-points ()
  (let ((obs (send self :observations)))
    (dotimes (i (length obs))
      (send (aref obs i) :change 'state 'normal)))
  (synchronize-graphs))
```

for setting the states of any invisible observations to normal, and the `:focus-on-selection` method

```
(defmeth observation-plot-mixin :focus-on-selection ()
  (let* ((obs (send self :observations))
         (showing (send self :points-showing))
         (selection (send self :selection)))
    (dolist (i (set-difference showing selection))
      (send (aref obs i) :change 'state 'invisible)))
  (synchronize-graphs))
```

for making all unselected points invisible.

Since the original linking system is no longer used, it is also a good idea to get rid of the linking items in the standard menus by defining the `:menu-template` method in the mixin as

```
(defmeth observation-plot-mixin :menu-template ()
  (remove 'link (call-next-method)))
```

The messages used by the standard mouse methods need to be modified as well. The `:unselect-all-points` method becomes

```
(defmeth observation-plot-mixin :unselect-all-points ()
  (let ((obs (send self :observations)))
    (dolist (i (send self :selection))
      (send (aref obs i) :change 'state 'normal))
    (send self :adjust-screen)))
```

The new :adjust-points-in-rect method can be defined in terms of the :points-in-rect message as

```
(defmeth observation-plot-mixin :adjust-points-in-rect
  (left top width height state)
  (let ((points (send self :points-in-rect
                        left top width height))
        (selection (send self :selection))
        (obs (send self :observations)))
    (case state
      (selected
       (dolist (i (set-difference points selection))
         (send (aref obs i) :change 'state 'selected)))
      (hilited
       (let* ((points (set-difference points selection))
              (hilited (send self :points-hilited))
              (new (set-difference points hilited))
              (old (set-difference hilited points)))
         (dolist (i new)
           (send (aref obs i) :change 'state 'hilited))
         (dolist (i old)
           (send (aref obs i) :change 'state 'normal))))))
    (synchronize-graphs)))
```

Both methods change the states in the observations, not the plots, and rely on the communication protocol to pass the changes from the observations back to their views.

Using the observation plot mixin, we can set up an observation scatterplot prototype as

```
(defproto obs-scatterplot-proto
  () () (list observation-plot-mixin
                scatterplot-proto))
```

A simple constructor function is given by

```
(defun plot-observations (obs vars)
  (let ((graph (send obs-scatterplot-proto :new vars)))
    (send graph :new-menu)
    (send graph :add-observations obs)
    (send graph :adjust-to-data)
    graph))
```

Other types of plots can be constructed along the same lines.

To illustrate the new linking system, we can turn once more to the stack loss data. As in Section 6.8.2, an observation prototype for this data set is given by

```
(defproto stack-obs
  '(air temp conc loss) () observation-proto)
```

Reader methods for the four variable slots are defined as

```
(defmeth stack-obs :air () (slot-value 'air))
(defmeth stack-obs :temp () (slot-value 'temp))
(defmeth stack-obs :conc () (slot-value 'conc))
(defmeth stack-obs :loss () (slot-value 'loss))
```

Derived variables, such as the log loss, can again be defined as

```
(defmeth stack-obs :log-loss () (log (send self :loss)))
```

The expression

```
(flet ((make-obs (air temp conc loss index)
  (let ((label (format nil "~d" index)))
    (send stack-obs :new
      :air air
      :temp temp
      :conc conc
      :loss loss
      :label label))))
  (setf stack-list (mapcar #'make-obs
    air temp conc loss (iseq 0 20))))
```

sets up a list of observation objects, and the expressions

```
(plot-observations stack-list '(:air :log-loss))
```

and

```
(plot-observations (select stack-list (iseq 10 20))
  '(:temp :conc))
```

produce two plots of the observations. One of the plots shows all the observations; the other shows only a subset. Since points are identified by their observation objects rather than their indices, this causes no problems.

Exercise 10.18

The new linking system forces two plots of the same data set to be linked at all times. It may be useful to be able to freeze a plot in its current state and have it no longer change as the observations change. Modify the linking system to support this notion.

Bibliography

- [1] ABELSON, H., AND SUSSMAN, G. J. (1985), *Structure and Interpretation of Computer Programs*, Cambridge, MA: MIT Press.
- [2] ADELANA, B. O. (1976), "Effects of plant density on tomato yields in western Nigeria," *Exper. Ag.* 12, 43-47.
- [3] ANSCOMBE, F. J. (1981), *Computing in Statistical Science through APL*, New York, NY: Springer.
- [4] ASIMOV, D. (1985), "The grand tour: a tool for viewing multidimensional data," *SIAM J. Scient. Stat. Comp.* 6, 128-143.
- [5] BATES, D. M., AND WATTS, D. G. (1988), *Nonlinear Regression Analysis and Its Applications*, New York, NY: Wiley.
- [6] BECKER, R. A., AND CHAMBERS, J. M. (1984), *S: An Interactive Environment for Data Analysis and Graphics*, Belmont, CA: Wadsworth.
- [7] BECKER, R. A., CHAMBERS, J. M., AND WILKS, A. R. (1988), *The new S language: A Programming Environment for Data Analysis and Graphics*, Pacific Grove, CA: Wadsworth.
- [8] BECKER, R. A., AND CLEVELAND, W. S. (1987), "Brushing scatterplots," *Technometrics* 29, 127-142.
- [9] BECKER, R. A., AND MCGILL, R. (1985), "Dynamic displays of data (video tape)," AT&T Bell Laboratories, Murray Hill, NJ.
- [10] BETZ, D. (1985), "An XLISP Tutorial," *BYTE*, 221.
- [11] BETZ, D. (1988), "XLISP: An experimental object-oriented programming language," Reference manual for Version 2.0.
- [12] BOBROW, D. G., DEMICHEL, L. G., GABRIEL, R. P., KEENE, S. E., KICZALES, G., AND MOON, D. A. (1988), *Common Lisp Object System Specification*, X3J13 Document 88-002R.

- [13] BOLORFORUSH, M., AND WEGMAN, E. J. (1988), "On some graphical representations of multivariate data," *Computing Science and Statistics: Proceedings of the 20th Symposium on the Interface*, E. J. Wegman, D. T. Ganz, and J. J. Miller, editors, Alexandria, VA: ASA, 121-126.
- [14] BOX, G. E. P., HUNTER, W. G., AND HUNTER, J. S. (1978), *Statistics for Experimenters*, New York, NY: Wiley.
- [15] BROWNLEE, K. A. (1965). *Statistical Theory and Methodology in Science and Engineering*, second edition, New York, NY: Wiley.
- [16] BUJA, A., ASIMOV, D., HURLEY, C. AND McDONALD, J. A. (1988), "Elements of a viewing pipeline for data analysis," in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill, editors, Belmont, CA: Wadsworth, 227-308.
- [17] CHALONER, K., AND BRANT, R. (1988) "A Bayesian approach to outlier detection and residual analysis," *Biometrika* 75, 651-660.
- [18] CHURCH, A. (1941), *The Calculi of Lambda-Conversion*, Princeton, NJ: Princeton University Press.
- [19] CLEVELAND, W. S. (1979), "Robust locally weighted regression and smoothing scatterplots," *J. Amer. Statist. Assoc.* 74, 829-836.
- [20] CLEVELAND, W. S., AND MCGILL, M. E. (1988) *Dynamic Graphics for Statistics*, Belmont, CA: Wadsworth.
- [21] COLEMAN, D., HOLLAND, P., KADEN, N., KLEMA, V., AND PETERS, S. C. (1908), "A system of subroutines for iteratively re-weighted least-squares computations," *ACM Trans. Math. Soft.* 6, 327-336.
- [22] COX, B. (1986), *Object-Oriented Programming: An Evolutionary Approach*, Reading, MA: Addison-Wesley.
- [23] COX, D. R., AND SNELL, E. J. (1981) *Applied Statistics: Principles and Examples*, London: Chapman and Hall.
- [24] DAVIES, O. L., AND GOLDSMITH, P. L. (1972), *Statistical Methods in Research and Production*, fourth revised edition, New York, NY: Hafner.
- [25] DENNIS, J. E., AND SCHNABEL, R. B. (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall.
- [26] DEVORE, J., AND PECK, R. (1986), *Statistics, the Exploration and Analysis of Data*, St. Paul, MN: West Publishing Co.
- [27] DIACONIS, P., AND FREEDMAN, D. (1984), "Asymptotics of graphical projection pursuit," *Ann. Statist.* 12, 793-815.

- [28] DONOHO, A. W., DONOHO, D. L., AND GASKO, M. (1988), "MAC-SPIN: Dynamic graphics on a desktop computer," in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill, editors, Belmont, CA: Wadsworth., 331-352.
- [29] FEIGL, P., AND ZELEN, M. (1965), "Estimation of exponential survival probabilities with concomitant information," *Biometrics* 21, 826-838.
- [30] FOWLKES, E. B. (1969), "User's manual for a system for interactive probability plotting on graphic-2," Technical Memorandum. Murray Hill, NJ: AT&T Bell Laboratories.
- [31] FRANZ INC. (1988). *Common Lisp: The Reference*, Reading, MA: Addison-Wesley.
- [32] FRIEDMAN, R. M., BIERBAUM, R. M., CATHERWOOD, P. A., DIAMOND, S. C., HOBERG, G. G., JR., AND LEE, V. A. (1983), "The acid rain controversy: The limits of confidence," *Am. Statistician* 37, 385-394.
- [33] GOLDBERG, A. (1984), *Smalltalk-80: The Interactive Programming Environment*, Reading, MA: Addison-Wesley.
- [34] HINKLEY, D. V. (1977), "On quick choice of power transformation," *Appl. Statist.* 26, 67-69.
- [35] INSELBERG, A., AND DIMSDALE, B. (1988), "Visualizing multi-dimensional geometry with parallel coordinates," *Computing Science and Statistics: Proceedings of the 20th Symposium on the Interface*, E. J. Wegman, D. T. Ganz, and J. J. Miller, editors, Alexandria, VA: ASA, 115-120.
- [36] KEENE, S. E. (1988), *Object-Oriented Programming in Common Lisp: A Programmer's Guide*, Reading, MA: Addison-Wesley.
- [37] KROLL, B., AND RAMEY, M. R. (1977), "Effects of bike lanes on driver and bicyclist behavior," *ASCE Trans. Eng. J.*, 243-256.
- [38] LALONDE, W. R. (1989), "Designing families of data types using exemplars," *ACM Trans. on Prog. Lang. and Sys.* 11, 212-248.
- [39] LANKTON, M. (1990), "External Functions," *Mac Tutor* 6, 1, 58-66.
- [40] LIEBERMAN, H. (1986), "Using prototypical objects to implement shared behavior in object-oriented systems," *Proceedings of the ACM Conference on Object-Oriented Systems and Languages (OOPSLA)*, 214-233.
- [41] LORENTZEN, T. J. (1980), "Determining statistical characteristics of a vehicle emissions audit procedure," *Technometrics* 22, 483-493.

- [42] McCULLAGH, P., AND NELDER, J. A. (1983), *Generalized Linear Models*, London: Chapman and Hall.
- [43] McDONALD, J. A. (1982), "Interactive Graphics for Data Analysis," unpublished Ph. D. thesis, Stanford University, Department of Statistics.
- [44] McDONALD, J. A. (1986), "Antelope: data analysis with object-oriented programming and constraints, *Proceedings of the 1986 Joint Statistical Meetings, Statistical Computing Section*, 1-10.
- [45] McDONALD, J. A., AND PEDERSEN, J. (1988), "Computing environments for data analysis III: Programming environments," *SIAM J. Sci. Stat. Comput.* 9, 191-211.
- [46] MOON, D. A. (1986), "Object-oriented programming with flavors," *Proceedings of the ACM Conference on Object-Oriented Systems and Languages (OOPSLA)*, 1-8.
- [47] OEHLMERT, G. W. (1987), "MacAnova User's Guide," Technical Report 493, University of Minnesota.
- [48] OLDFORD, R. W., AND PETERS, S. C. (1988), "DINDE: Towards more sophisticated software environments for statistics," *SIAM J. Sci. Stat. Comput.* 9, 191-211.
- [49] PICKUP, G., AND NELSON, D. J. (1984), "Use of Landsat radiance parameters to distinguish soil erosion, stability, and deposition in arid central Australia," *Remote Sensing of Environment* 16, 195-209
- [50] PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., AND VETTERLING, W. T. (1988), *Numerical Recipes in C*, Cambridge: Cambridge University Press.
- [51] PROSCHAN, F. (1963), Theoretical explanation of observed decreasing failure rate, *Technometrics* 5, 376-383.
- [52] REAVEN, G. M., AND MILLER, R. G. (1979), "An attempt to define the nature of chemical diabetes using a multidimensional analysis," *Diabetologia* 16, 17-24.
- [53] SCHAFFERT, S., COOPER, T., BULLIS, B., KILIAN, M., AND WILPOLT, C. (1986), "An introduction to Trellis/Owl," *Proceedings of the ACM Conference on Object-Oriented Systems and Languages (OOPSLA)*, 9-16.
- [54] SCHMUCKER, K. J. (1986), *Object-Oriented Programming for the Macintosh*, Rochelle Park, NJ: Hayden.

- [55] SNYDER, A. (1986), "Encapsulation and inheritance in object-oriented systems," *Proceedings of the ACM Conference on Object-Oriented Systems and Languages (OOPSLA)*, 38-45.
- [56] STEELE, GUY L. (1990), *Common Lisp: The Language*, second edition, Bedford, MA: Digital Press.
- [57] STROUSTRUP, B. (1986), *The C++ Programming Language*, Reading, MA: Addison-Wesley.
- [58] STUETZLE, W. (1987), "Plot windows," *J. Amer. Statist. Assoc.* 82, 466-475.
- [59] TATER, D. G. (1987), *A Programmer's Guide to Common Lisp*, Bedford, MA: Digital Press.
- [60] TEJADA, C., CHARM, S., GUZMAN, M., MENDEZ, J., AND KURLAND, G. (1964), "The blood viscosity of various socioeconomic groups in Guatemala," *Amer. J. of Clinical Nutrition* 15, 303-307.
- [61] THISTED, R. A. (1988), *Elements of Statistical Computing*, New York, NY: Chapman and Hall.
- [62] TIERNEY, L., AND KADANE, J. B. (1986), "Accurate approximations for posterior moments and marginal densities," *J. Amer. Statist. Assoc.* 81, 82-86.
- [63] TIERNEY, L., KASS, R. E., AND KADANE, J. B. (1989), "Fully exponential Laplace approximations to expectations and variances of non-positive functions," *J. Amer. Statist. Assoc.* 84, 710-716.
- [64] TIERNEY, L., KASS, R. E., AND KADANE, J. B. (1989), "Approximate marginal densities for nonlinear functions," *Biometrika* 76, 425-434..
- [65] WEISBERG, S. (1985), *Applied Linear Regression*, second edition, New York, NY: Wiley.
- [66] WEISBERG, S. (1982), "MULTREG Users Manual," Technical Report 298, University of Minnesota.
- [67] WINSTON, P. H., AND HORN, B. K. P. (1988), *LISP*, third edition, Reading, MA: Addison-Wesley.
- [68] YAMADA, T., TAKAHASHI-ABBE, S., AND ABBE, K. (1985), "Effects of oxygen on pyruvate formate-lyase in situ and sugar metabolism of streptococcus mutans and streptococcus sanguis," *Infection and Immunity* 47, 129-134.

- [69] ZULIANI, U., MANDRAS, A., BELTRAMI, G. F., BONETTI, A., MONTANI, G., AND NOVARINI, A. (1983), "Metabolic modifications caused by sport activity: Effect in leisure-time cross-country skiers," *J. Sports Med.* 23, 385-392.

Appendix A

Answers to Selected Exercises

2.1

- a) 14
- b) 2
- c) (+ 3 5 6)
- d) 10
- e) X

The interpreter prints the symbol using upper case letters.

- f) (QUOTE X)

The interpreter expands the single quote to the full quote expression before returning it.

2.2

- a) 2
- b) (5 6 7)
- c) (4 10 18)
- d) This generates an error, since the vectorized arithmetic system requires that compound data arguments be the same length.

2.5

```
(def x (rseq -2 3 30))
(plot-lines x (+ (* 2 x) (* x x)))
```

2.7

- a) (1 1 1 2 2 2 3 3 3 4 4 4)
- b) (3 3 3 3 3 3 3 3 3 3)
- c) (A B B C A A B C C)

2.8

```
(def board (repeat (iseq 1 5) (repeat 8 5)))
(def chip-type (repeat (repeat (iseq 1 4) (repeat 2 4)) 5))
```

2.9

```
(+ 3.2 (* 1.7 (normal-rand 30)))
```

2.10

```
(select x (which (<= 3 x 5)))
```

2.11

x is the list (a 3). Since this list is also part of y, the value of y is the list (1 (a 3) 4).

2.13

The random numbers fall on a set of 15 parallel planes. As a result, you should be able to find a rotation of the data in which you are looking along the planes and the data appear to fall in a set of parallel lines.

3.1

```
(defun non-neg-real-p (x)
  (and (numberp x)
       (not (complexp x))
       (<= 0 x)))
```

3.2

Define a function as

```
(defun f (x) (if (/= x 0) (/ (sin x) x) 1))
```

If if were a function, not a special form, then evaluating (f 0) would produce a division by zero error, since the second argument to the if expression would be evaluated even though it is not needed.

3.5

- a) 17
- b) 15
- c) 32
- d) 52

3.7

```
(defun make-norm-log-lik (x)
  (let ((n (length x))
        (x-bar (mean x))
        (s-2 (^ (standard-deviation x) 2)))
    #'(lambda (mu sigma-2)
        (- (* 0.5
              n
              (+ (log sigma-2)
                  (/ (^ (- x-bar mu) 2) sigma-2)
                  (/ s-2 sigma-2)))))))
```

3.8

Define the function `make-composition` by

```
(defun make-composition (f g)
  #'(lambda (x) (funcall f (funcall g x))))
```

Then

```
> (def ident (make-composition #'log #'exp))
IDENT
> (funcall ident 3.7)
3.7
```

6.4

```
(defun check-send (object selector &rest args) (if (send
object :has-method selector) (apply #'send object
selector args)))
```

6.9

```
(defun data-from-file (&optional name)
  (let ((cols (if name
                  (read-data-columns name)
                  (read-data-columns))))
```

```
(if cols
    (let ((data (mapcar #'rest cols))
          (labels (mapcar #'first cols)))
      (send rect-data-proto :new
            data
            :variable-labels labels)))))
```

6.12

```
(defmeth data-set-proto :regression
  (x y &key (intercept t) (print t))
  (let* ((x (if (consp x) x (list x)))
         (xvals (mapcar #'(lambda (x)
                             (send self :values x))
                         x))
         (yvals (send self :values y))
         (xnames (mapcar #'string x))
         (yname (string y)))
    (regression-model xvals yvals
                      :intercept intercept
                      :print print
                      :predictor-names xnames
                      :response-name yname
                      :case-labels (send self
                                         :values
                                         :label))))
```

7.1

```
(defmeth window-proto :center-window ()
  (let* ((size (send self :frame-size))
         (loc (round (/ (- (screen-size) size) 2))))
    (apply #'send self :frame-location loc)))
```

8.1

```
(defmeth w :draw-wheel (x y r)
  (let ((cleft (- x r))
        (ctop (- y r))
        (cheight (* 2 r))
        (sleft (round (- x (/ r (sqrt 2))))))
        (stop (round (- y (/ r (sqrt 2))))))
        (sheight (round (* (sqrt 2) r)))))
    (send self :frame-oval cleft ctop cheight cheight)
    (send self :frame-rect sleft stop sheight sheight)))
```

8.2

Redefine the :draw-wheel method to take an angle argument as

```
(defmeth w :draw-wheel (x y r &optional (angle 0))
  (let ((cleft (- x r))
        (ctop (- y r))
        (cheight (* 2 r)))
    (flet ((point (angle)
               (list (round (+ x (* r (cos angle))))))
           (round (- y (* r (sin angle)))))))
      (let ((p1 (point (+ angle (/ pi 4)))))
        (p2 (point (+ angle (* 3 (/ pi 4))))))
        (p3 (point (+ angle (* 5 (/ pi 4))))))
        (p4 (point (+ angle (* 7 (/ pi 4)))))))
        (send self :frame-oval cleft ctop cheight cheight)
        (send self :frame-poly (list p1 p2 p3 p4 p1))))
```

The expressions

```
(send w :draw-mode 'xor)
(dotimes (i 51)
  (let ((a (* i (/ pi 25))))
    (send w :draw-wheel 100 100 20 a)
    (pause 1)
    (send w :draw-wheel 100 100 20 a)))
(send w :draw-mode 'normal)
```

rotate the circle with XOR drawing, and the expression

```
(dotimes (i 51)
  (let ((a (* i (/ pi 25))))
    (send w :start-buffering)
    (send w :erase-window)
    (send w :draw-wheel 100 100 20 a)
    (send w :buffer-to-screen)))
```

rotates the circle with double buffering.

8.3

Add `center`, `radius` and `angle` slots as

```
(send w :add-slot 'center '(100 100))
(send w :add-slot 'radius 20)
(send w :add-slot 'angle 0)
```

and accessors

```
(defmeth w :center (&optional new)
  (if new (setf (slot-value 'center) new))
  (slot-value 'center))

(defmeth w :radius (&optional new)
  (if new (setf (slot-value 'radius) new))
  (slot-value 'radius))
```

and

```
(defmeth w :angle (&optional new)
  (if new (setf (slot-value 'angle) new))
  (slot-value 'angle))
```

Then the :resize method is defined as

```
(defmeth w :resize ()
  (let ((x (round (/ (send self :canvas-width) 2))))
    (y (round (/ (send self :canvas-height) 2))))
  (send self :center (list x y))))
```

and the :redraw method by

```
(defmeth w :redraw ()
  (let ((center (send self :center))
        (r (send self :radius))
        (a (send self :angle)))
    (send self :start-buffering)
    (send self :erase-window)
    (send self :draw-wheel
          (first center) (second center) r a)
    (send self :buffer-to-screen)))
```

8.7

```
(defmeth w :do-idle ()
  (send self :angle (+ (send self :angle) .1))
  (send self :redraw))
```

9.1

```
(defmeth w :set-point-symbols (sym &key (draw t))
  (let ((i (iseq (send self :num-points))))
    (send self :point-symbol i sym)
    (if draw (send self :redraw))))
```

9.2

```
(defmeth w :add-segments (x1 x2)
  (let* ((n (length (first x1)))
         (i1 (iseq n))
         (i2 (+ n i1))
         (starts (mapcar #'append x1 x2)))
    (send self :add-lines starts :draw nil)
    (send self :linestart-next i1 i2)
    (send self :linestart-next i2 nil)
    (send self :redraw)))
```

9.6

```
(defmeth w :do-identify (x y m1 m2)
  (let* ((cr (send self :click-range))
         (p (first (send self :points-in-rect
                           (- x (/ (first cr) 2))
                           (- y (/ (second cr) 2))
                           (first cr)
                           (second cr)))))

    (if p
        (let ((mode (send self :draw-mode))
              (label (send self :point-label p)))
          (flet ((move-label (new-x new-y)
                  (send self :draw-string label x y)
                  (setf x new-x)
                  (setf y new-y)
                  (send self :draw-string label x y)))
              (send self :draw-mode 'xor)
              (send self :draw-string label x y)
              (send self :while-button-down #'move-label)
              (send self :draw-string label x y)
              (send self :draw-mode mode)))))))
```

9.7

```
(defmeth w :adjust-screen-point (i)
  (let* ((state (send self :point-state i))
         (color (if (eq state 'normal)
                    (send self :back-color))))
    (send self :point-color i color))
  (call-next-method i))
```

Appendix B

The XLISP-STAT Implementation

XLISP-STAT is an implementation of Lisp-Stat based on the XLISP language. XLISP-STAT Version 2.1, the current version at the time of writing, is based on XLISP Version 2.0. Older versions of XLISP-STAT may deviate in a number of ways from the description in this book.

XLISP is a dialect of Lisp developed and implemented by David Betz [10,11]. The system is written in C and can be ported easily to a wide range of computers. XLISP is intended to be a subset of Common Lisp, and provides a large subset of the functions required by the Common Lisp specification [56]. Two Common Lisp features not supported in XLISP 2.0 are packages and structures. XLISP also does not have a compiler.

The XLISP-STAT implementation adds a number of additional Common Lisp functions. It also adds the vectorized arithmetic system, the statistical and linear algebra functions, the Lisp-Stat object system, and the graphics system described in this book. Many of the additions are written in C, but some are written in Lisp.

The XLISP-STAT system loads a number of files containing Lisp definitions when it starts up. After loading its own files, it looks for a file `statinit.lsp`. If such a file is found, it is loaded. You can use this file to add definitions of your own that you would like to load on start-up.

The XLISP-STAT graphics system was originally designed for use on the Apple Macintosh, but has now been adapted to the *X11* and *SunView* systems. Several other versions are currently under development.

XLISP 2.0 includes a class-based object system. Some aspects of the Lisp-Stat object system described in Chapter 6 are motivated by the XLISP object system, but the basic design of the Lisp-Stat object system is quite different. XLISP-STAT uses the prototype-based Lisp-Stat object system, not the system provided by XLISP.

The XLISP-STAT distribution includes several sample data sets and ex-

ample files. The data sets used in the examples in this book can be read into XLISP-STAT by loading the file `book.lsp`.

B.1 Obtaining the XLISP-STAT Software

The source code for the Macintosh and UNIX versions of XLISP-STAT and compiled versions for the Macintosh are available free of charge. At the time of writing, both the source code and the Macintosh executables can be obtained by anonymous *ftp* over the internet from *umnstat.stat.umn.edu* (*128.101.51.1*).

If you do not have access to the internet but can use electronic mail, you can obtain XLISP-STAT from the `statlib` archive. To find out how to do this, send a mail message containing the line

```
send index from xlispstat
```

to *statlib@temper.stat.cmu.edu*.

Since there are no restrictions on copying the software, you can also obtain a copy from anyone else who has a copy.

Other versions and other methods of distribution may become available in the future. For more information on current versions, write to

Lisp-Stat Information
School of Statistics
270 Vincent Hall
University of Minnesota
Minneapolis, MN 55455

or send electronic mail to *lispstat-info@umnstat.stat.umn.edu*.

B.2 XLISP-STAT on the Macintosh

B.2.1 Basics

The Macintosh version of XLISP-STAT consists of an application program and a number of Lisp text files with names ending in `.lsp`. Examples and sample data sets are contained in folders named `Examples` and `Data`. To run properly, the application must be placed in the same folder as the `.lsp` files and the `Data` and `Examples` folders.

Two versions of the application are available for the Macintosh. One version can run on any Macintosh architecture, the other requires a 68020 CPU or higher and a math coprocessor. The coprocessor version is considerably faster.

It may be possible to use XLISP-STAT on small data sets on a computer with 1 megabyte of memory, but to work comfortably it requires 2 megabytes.

It may require more memory on a system with a large screen. The default multifinder partition of the application is set to 2 megabytes.

To start XLISP-STAT, double click on the application. It takes a little time while the Lisp files are loaded. When the system is ready, it provides a *listener* window for typing commands, and a number of menus. The listener window provides parenthesis matching, and hitting a *tab* in a multiline expression indents the current line of the expression to a reasonable position.

The system also provides a simple text-editing facility. The **New** and **Open** items on the **File** menu can be used to create new text files or to open existing text files. The editor windows can only handle files with up to 32K characters, but they do provide the same parenthesis-matching and indentation assistance as the listener window.

Both the listener window and the editor windows allow a selection of text to be evaluated by the lisp system by choosing the **Eval Selection** item from the **Edit** menu. This provides a convenient way of evaluating function definitions in a file opened for editing.

Files of Lisp expressions can be loaded into XLISP-STAT by using the **load** command or by selecting the **Load** item from the **File** menu.

You can interrupt a calculation that is taking too long or was started in error by typing and holding down the COMMAND key and the PERIOD key until the system is reset to the top level.

On the Macintosh, XLISP-STAT includes the symbol **MACINTOSH** in the ***features*** list.

B.2.2 Menus

Menus and menu items on the Macintosh include a few additional features that are part of the Macintosh user interface.

Apple Menus

A second basic menu prototype is available on the Macintosh, the apple menu prototype **apple-menu-proto**. An apple menu behaves like an ordinary menu, except that it has desk accessories below the last regular menu item.

Style Options

A Macintosh menu item can have its title displayed in several different styles. Style options are represented by the symbols **bold**, **italic**, **underline**, **shadow**, **condense**, and **extend**. The **:style** message can be used to modify the style of an item. The argument to the message should be **nil**, a single style symbol, or a list of style symbols.

The **:isnew** method for the menu item prototype allows the **:style** keyword to be used to set an item's style.

Keyboard Equivalents

On the Macintosh you can give a menu a command key equivalent. Typing the specified key while holding down the COMMAND key is then equivalent to selecting the item. The message :key sets and retrieves the current key character. To change the key character of an item, send the :key message with a Lisp character or nil as the argument. If hello is a menu item, then

```
(send hello :key #\H)
```

sets the keyboard equivalent of the item to *Command-H*, and

```
(send hello :key nil)
```

removes the keyboard equivalent.

The :key keyword can also be used to specify a keyboard equivalent to the menu item :isnew method.

Standard Menus and the Default Menu Bar

When XLISP-STAT starts up, it places the **Apple**, **File**, **Edit**, and **Command** menus in the menu bar. The menu objects for these menus are the values of the variables *apple-menu*, *file-menu*, *edit-menu*, and *command-menu*. The variable *standard-menu-bar* contains the list of these menus.

The function **set-menu-bar** takes a list of menus, removes all currently installed menus, and installs the menus in the argument list as the new menu bar.

The function **find-menu** returns the Lisp-Stat object representing a menu with a specified string as its title, if one is installed in the menu bar. Otherwise, it returns nil. The string match ignores the case of characters in the string.

B.2.3 Windows

Operating System Interface

The Macintosh user interface directs all keyboard input and mouse actions to the current front window. Usually this window belongs to the current applications, but at times it belongs to a desk accessory.

All windows created by XLISP-STAT are associated with window objects. The function **front-window** returns the window object for the front window if it is an XLISP-STAT window, or nil if the front window belongs to a desk accessory.

When an XLISP-STAT window becomes the front window, it is sent the :activate message with the argument t. When another window becomes the front window, the previous front window object receives the message :activate with argument nil. The **graph-proto** method for this message is

responsible for installing or removing a plot's menu in the menu bar. When an obscured portion of a window is exposed or the window is resized, the system sends the window the :update message. The message is sent with one argument, t if the window was resized and nil if the window was not resized. The :update methods take care of adjusting scroll bars and sending the :redraw or :resize messages.

Clipboard Support

The Macintosh uses the *clipboard* for transferring text and other items between windows and applications. To support using the clipboard, all windows include methods for the messages :cut-to-clip, :copy-to-clip, :paste-from-clip, and :clear. These messages are sent to the front window by the corresponding items in the **Edit** menu. The **window-proto** prototype provides stub methods for these messages that do nothing. The edit window prototype described below includes methods that use the clipboard.

Several other messages used by the **Edit** menu items are :paste-stream, :paste-string, :selection-stream, :find, and :find-again. The function **system-edit** implements the *SystemEdit Toolbox* routine. This function is used by the standard **Edit** menu items to determine if a user action is meant for a desk accessory.

Edit Windows

The **edit-window-proto** prototype provides an interface to the editing windows available in XLISP-STAT on the Macintosh. Editor windows are implemented using Paul DuBois' *TransEdit* package. The windows can only handle files of at most 32K characters. A new untitled editor window can be set up by using the expression

```
(send edit-window-proto :new)
```

The expression

```
(send edit-window-proto :new :bind-to-file t)
```

presents an open file dialog, and opens an editor window with the selected file if the **OK** button is pressed. These expressions are the bodies of the action functions for the **New Edit** and **Open Edit** items in the standard **File** menu.

The :remove method of **edit-window-proto** checks whether the text in the window has been changed since the file was last saved. If the window's contents have been changed, a dialog is put up to ask if the changes should be saved, or discarded, or if the :remove should be canceled.

The **edit-window-proto** prototype provides methods for the clipboard messages :cut-to-clip, :copy-to-clip, :paste-from-clip, and :clear. These methods take no arguments.

The messages :revert, :save, :save-as, and :save-copy are used to manipulate the file currently being edited in the window. They take no arguments, and are sent to the front editor window by the corresponding items in the **File** menu.

If you want to insert text directly into an editor window, you can use :paste-stream to insert text from a stream or :paste-string to insert a string. These insertions are treated like text typed in from the keyboard: the text is placed at the current insertion point, replacing any selection. You can also extract the current selection by sending the window the :selection-stream message. This message takes no arguments, and returns a stream containing the selected text.

Methods for two messages used by items in the **Edit** menu are implemented in terms of these messages. The :edit-selection method opens a new edit window containing the current selection, and the :eval-selection method evaluates the current selection.

The :find-string message takes a string argument and selects the string if it is found in the window. The string comparison is case insensitive and starts at the *end* of the current selection. This facilitates repeated searches for the same string. The result returned is t if the string is found, nil otherwise. The method for :find opens a dialog to prompt for the string to find, and then sends :find-string to the window. If the string is not found, a beep is sounded. The :find-again message uses :find-string to search for the next occurrence of the string requested in the last :find message.

Display Windows

A variant of the **edit-window-proto** prototype is **display-window-proto**. Input for a display window is disabled, and it is not bound to a file. But it can have text added to it by using the :paste-stream and :paste-string messages. In addition, it is possible to delete text from the beginning of a display window by sending it the :flush-window message with the number of characters to be deleted as its argument. With no argument, the method for this message deletes all text in the window.

Display windows are useful for displaying help text or results of computations.

The Listener Window

The internal listener window is accessed through an object inheriting from the **listener-proto** prototype. This prototype inherits from **edit-window-proto**. A new listener object is created and assigned to the variable *listener* when the system starts up. Creating a new listener object does not create a new internal listener; it just sets up a new Lisp-Stat object for communicating with the internal listener window.

Some Macintosh Dialogs

XLISP-STAT includes three dialogs that are specific to the Macintosh.

The `about-xlisp-stat` function is used by the **About XLISP-STAT** item in the standard apple menu to presents a simple alert dialog giving information about the system.

The function `open-file-dialog` requires no arguments, and presents the standard Macintosh dialog for opening a file. If the **OK** button in the dialog is pressed, the function returns the name of the selected file as a string. If the **Cancel** button is pressed, the function returns `nil`. The `open-file-dialog` function accepts an optional argument that should be `t` or `nil`. This argument specifies whether the default folder for reading and writing files is to be set to the folder from which the file name was chosen. The default folder is not changed if the dialog is dismissed with the **Cancel** button. The default value of this optional argument is `t`.

The standard Macintosh dialog for naming a new file is presented by the function `set-file-dialog`. This function requires one argument, a prompt string, and takes two optional arguments. The first optional argument is a string to use as the initial file name; the default is an empty string. The second optional argument specifies whether the default folder is to be changed. The `set-file-dialog` function returns the specified name as a string if the **OK** button is pressed, and `nil` for the **Cancel** button.

B.2.4 More Advanced Features

Cursors

The Macintosh operating system allows cursors to be constructed as *resources* using, for example, the `ResEdit` application. Resources are classified by type, and the cursor resource type is '`'CURS'`'. Within a type, resources are identified by an ID number or a name. On the Macintosh, the `make-cursor` function can be called with an integer, specifying a cursor resource ID, or a string, specifying a cursor name, as its second argument. An error is signaled if no cursor resource with the specified ID or name is found.

By default, the search for resources starts by searching the resource fork of the application and then searches the system resource file. You can place the resource fork of another file at the head of this search path by opening the file with the `open-resource-file` function. This function takes one argument, a string specifying the file to open. It returns an integer, the resource file ID. The system maintains a list of resource files to search, with the most recently opened file at the head of the list. You can close a resource file by calling the function `close-resource-file` with the resource file ID as its argument.

External Functions

The Macintosh version of XLISP-STAT provides a simple mechanism for adding compiled C functions to the system.¹ This mechanism assumes that the C function is compiled to a code resource of type 'XLSX' with its entry point at the function. The resource must be named. It can be placed in the XLISP-STAT application, or it can be made available by placing it in another file and opening the file with the `open-resource-file` function.

The C function should be declared to match the prototype

```
void foo(XLSXblock *params);
```

The declaration of the parameter block `XLSXblock` is contained in the include file `xlsx.h`, which is included in the Macintosh distribution of XLISP-STAT. The parameter block referenced by the pointer `params` is used to pass the arguments from the system to `foo`, and to return any results.

The function `call-cfun` takes a string identifying the 'XLSX' resource containing your function, followed by additional arguments for your C function. These additional arguments must be integers, sequences of integers, real numbers, or sequences of real numbers. Pointers to `int` or `double` data containing these values are passed to your routine through the parameter block. After your routine returns, the contents of the data referred to by these pointers are copied into lists, and `call-cfun` returns a list of these lists.

Several macros are defined in `xlsx.h`. The macro `XLSXarg` takes the parameter block pointer and an index `i`, and returns the pointer to the `i`-th argument, cast to `(char *)`. The macro `XLSXargc` takes the parameter block pointer and returns the number of arguments in the call.

As an example, suppose that the file `foo.c` contains the definitions

```
#include "xlsx.h"

void foo(params)
    XLSXblock *params;
{
    int *n, i;
    double *x, *sum;

    n = (int *) XLSXargv(params, 0);
    x = (double *) XLSXargv(params, 1);
    sum = (double *) XLSXargv(params, 2);
    for (i = 0, *sum = 0.0; i < *n; i++) {
        *sum += x[i];
    }
}
```

¹The mechanism is based on an adaptation of the mechanism outlined by Lankton [39].

The function `foo` simply adds up a set of floating point numbers, and returns their sum. If you are using the MPW C compiler, you can compile and link the file with the commands

```
c foo.c
```

and

```
link -sg "foo" foo.c.o -rt XLSX -m foo {LIBS} -o foo
```

This link expression assumes that the MPW shell variable `LIBS` contains the required libraries for the code resource.

After linking the resource into a file `foo`, we can make it available with the expression

```
(open-resource-file "foo")
```

We can then call the function `foo` by using a list of real numbers as the second argument. The function `float` can be used to coerce numbers to reals:

```
> (call-cfun "foo" 5 (float (iseq 1 5)) 0.0)
((5) (1 2 3 4 5) (15))
```

The third argument to `foo` is used to return the result.

External functions contain only the error checking you build into them. If a function is not called with the proper arguments, it will most likely cause XLISP-STAT to crash, losing any variables you have not saved.

B.3 XLISP-STAT on UNIX Systems

XLISP-STAT is also available on UNIX systems. If it has been installed in a directory in your search path, you should be able to start it up by typing

```
xlispstat
```

at the UNIX shell level. There are a few differences between the Macintosh and UNIX versions:

- UNIX versions of XLISP-STAT are designed to run on a standard terminal and therefore do not provide parenthesis matching or indentation support. If you use the *GNU emacs* editor, you can obtain both of these features by running XLISP-STAT from within *emacs*. Otherwise, for editing files with *vi*, you can use the `-l` flag to get some Lisp editing support.
- To quit from the program, type

```
(exit)
```

On most systems you can also quit by typing a *Control-D*.

- You can interrupt a calculation that is taking too long or was started in error by typing a *Control-C*.
- Data and example files are stored in the **Data** and **Examples** subdirectories of the library tree. Within XLISP-STAT the variable ***default-path*** shows the root directory of the library; you can look there if you want to examine the example files.

On basic UNIX systems the only graphics available are the functions **plot-points** and **plot-lines**. These functions assume that you are using a *Tektronix* terminal or emulator.

On the UNIX systems, XLISP-STAT includes the symbol **UNIX** in the ***features*** list.

B.3.1 XLISP-STAT under X11

Window-based graphics are available in XLISP-STAT on a workstation running the *X11* window system. Graphics under *X11* are similar to Macintosh graphics. A few minor differences are:

- Plot menus are popped up using a menu button in the top right corner of a plot.
- The extend modifier for mouse and key events is the *shift* key. The option modifier is the *control* key.
- How plot windows are opened in response to a graphics command depends on the window manager you are using. Under **twm**, the default window manager on many systems, plots appear immediately on the screen. Other window managers may provide you with a little corner that you can use to choose the position of your plot window.
- Slider dialog items are the only items that assume a three button mouse. In the central part of the slider the right button increases and the left button decreases the slider value. The middle button drags the thumb indicator.
- *PostScript* images of plots can be saved by selecting the **Save to File** item in a plot menu. The *PostScript* file can then be printed by using a standard printing command.

When run under *X11*, XLISP-STAT includes the symbol **X11** in the ***features*** list.

More Advanced *X11* Features

You can have XLISP-STAT use an alternate display for its graphics by setting the `DISPLAY` environment variable before starting XLISP-STAT. You can specify alternate fonts and a few other options by using the *X11* resource management facilities. Resources controlling appearance are

```
xlisp*menu*titles:    on for a title on a menu, off otherwise  
xlisp*menu*font:  
xlisp*dialog*font:  
xlisp*graph*font:
```

There are also a few experimental options for controlling performance. These are

```
xlisp*graph*fastlines:    on to use 0 width lines  
xlisp*graph*fastsymbols:  on to use DrawPoints instead of bitmaps  
xlisp*graph*motionsync:   on to use XSync during mouse motion
```

By default all three options are *on*. These settings seems to give the best performance on a *Sun 3/50*. It may not be the best choice on other workstations. You can use the function `x11-options` to change these three options from within XLISP-STAT. The `fastlines` option does not take effect immediately when changed in this way, but will affect the next plot created. The other two options do take effect immediately.

B.3.2 XLISP-STAT under *SunView*

Window-based graphics are also available when XLISP-STAT is run on a *Sun* console running `suntools`. Graphics under `suntools` work like graphics on the Macintosh, with the following modifications:

- To close or resize plots or dialogs, use the frame menu or the standard *SunView* shortcuts (e.g., to resize a plot window, drag the frame with the middle mouse button while holding down the *control* key).
- Plot menus are popped up by pressing the right mouse button in the interior of the plot. Check marks do not appear on menu items, so it may not always be possible to determine the state of an item.
- Clicking with no modifiers corresponds to clicking with the left mouse button. Clicking with the extend modifier corresponds to clicking the middle mouse button.
- *PostScript* images of plots can be saved by selecting the **Save to File** item on a plot menu.

When run under *SunView*, XLISP-STAT includes the symbol `sunview` in the `*features*` list.

B.3.3 Running UNIX Commands from XLISP-STAT

The `system` function can be used to run UNIX commands from within XLISP-STAT. This function takes a shell command string as its argument and returns the shell exit code for the command. For example, you can print the date by using the UNIX `date` command:

```
> (system "date")
Wed Jul 19 11:06:53 CDT 1989
0
>
```

The return value is 0, indicating successful completion of the UNIX command.

B.3.4 Dynamic Loading and Foreign Function Calling

Some UNIX implementations of XLISP-STAT provide a facility to allow you to use your own C functions or FORTRAN subroutines from within XLISP-STAT. The facility, patterned after the approach used in the New *S* language [7], consists of the function `dyn-load` for loading your code into a running XLISP-STAT process and the functions `call-cfun`, `call-fsub`, and `call-lfun` for calling subroutines in the loaded code. The `dyn-load` function requires one argument, a string containing the name of a file to be linked and loaded. The file is linked with standard C libraries before loading. If you need it to be linked with the standard FORTRAN libraries as well, you can give the keyword argument `:fortran` the value `t`. If you need to link other libraries, you can supply a string containing the library flags that you would specify in a linking command as the value of the keyword argument `:libflags`. For example, to include the library `cmlib`, use the string `"-lcmlib"`.²

The function `call-cfun` takes a string identifying the C function that you want to call, followed by additional arguments for your C function. The additional arguments must be integers, sequences of integers, real numbers, or sequences of real numbers. Pointers to `int` or `double` data containing these values are passed to your routine. After your routine returns, the contents of the data referred to by these pointers are copied into lists, and `call-cfun` returns a list of these lists.

As an example, suppose that the file `foo.c` contains the following C function definition:

```
foo(n, x, sum)
  int *n;
  double *x, *sum;
```

²There may be slight variations in the implementation of `dyn-load` on different systems. The help information for this function should give information that is appropriate for your system.

```
{
    int i;

    for (i = 0, *sum = 0.0; i < *n; i++) {
        *sum += x[i];
    }
}
```

After compiling the file to `foo.o`, we can load it into XLISP-STAT by using the expression

```
(dyn-load "foo.o")
```

We can then call the function `foo` by using a list of real numbers as the second argument:

```
> (call-cfun "foo" 5 (float (iseq 1 5)) 0.0)
((5) (1 2 3 4 5) (15))
```

The third argument to `foo` is used to return the result.

The function `call-fsub` is used for calling FORTRAN subroutines that have been loaded dynamically. A FORTRAN subroutine analogous to the C function `foo` might be written as

```
subroutine foo(n, x, sum)
integer n
double precision x(n), sum

integer i

sum = 0.0
do 10 i = 1, n
    sum = sum + x(i)
10 continue
return
end
```

After compiling and loading this routine, it can be called by using `call-fsub`:

```
> (call-fsub "foo" 5 (float (iseq 1 5)) 0.0)
((5) (1 2 3 4 5) (15))
```

Two C functions you may want to call from your C code are `xscall_alloc` and `xscall_fail`. The function `xscall_alloc` is like `malloc`, except it ensures that the allocated memory is garbage collected after the call to `call-cfun` returns. The function `xscall_fail` takes a character string as its argument. It prints the string and signals an error.

The function `call-lfun` can be used to call C functions written using the internal XLISP conventions for obtaining arguments and returning results.

This allows you to accept any kinds of arguments. Unfortunately, the source code is the only documentation for the internal calling conventions.

Dynamically loaded code contains only the error checking you build into it. If a function is not called with the proper arguments, it will most likely cause XLISP-STAT to crash, losing any variables you have not saved.

At present, in XLISP-STAT the number of arguments you can pass to C functions or FORTRAN subroutines using `call-cfun` or `call-fsub` is limited to 15.

B.4 Some Additional Features

Loading Data and Code Examples

Each XLISP-STAT implementation provides standard locations for storing data and code examples. Several examples are included with the distribution. The functions `load-data` and `load-example` are defined to take a file name string, with or without a `.lsp` extension, and load the file from the appropriate location. Thus the expression

```
(load-data "book")
```

loads the example data sets used in this book.

Tracing Messages

The XLISP-STAT tracing system allows both messages and functions to be traced. The expression

```
(trace :close)
```

traces `:close` messages. Closing a window after issuing this trace command prints

```
Entering: :CLOSE, Argument list: ()  
Exiting: :CLOSE, Value: NIL
```

to the interpreter.

Memory Management

The XLISP memory management system is responsible for providing memory space when it is needed by allocating new memory or by reclaiming memory that is no longer being used. For the most part you can ignore this system. But the system does include a few tuning parameters that can be used to improve performance.

The basic memory unit is a *node*. A node is a chunk of memory large enough to hold any lisp atom, such as a number or a cons cell. The system starts with an initial allocation of nodes. When all nodes are used up, the

garbage collector is run to reclaim any unused storage. If that does not free up enough space, more nodes are allocated.

The function `room` prints information about the current state of the memory management system. The result might look like

```
> (room)
Nodes:      36640
Free nodes: 9621
Segments:   20
Allocate:   1
Total:      522267
Collections: 4
```

New nodes are allocated in segments. The current segment size is given by the `Allocate` item in this list. An alternative value can be specified by using the `alloc` function. You can instruct the system to allocate a number of additional segments by calling the `expand` function with an integer argument, the number of new segments. Thus after evaluating

```
(expand 5)
```

the `room` function prints

```
> (room)
Nodes:      46640
Free nodes: 19419
Segments:   25
Allocate:   2000
Total:      642339
Collections: 4
```

You can force a garbage collection by calling the `gc` function with no arguments.

The garbage collection strategy used by XLISP is a *mark and sweep* strategy. The garbage collector marks all nodes that are accessible to the system for safe keeping, and then sweeps through all nodes in the system and places any unmarked nodes on a list of free nodes. The most time-consuming step in this strategy is usually the mark phase. The time required depends on the complexity of the graph of accessible nodes, which in turn depends on the expressions that have been evaluated. It is essentially independent of the size of the node space. The time required by the sweep phase does depend on the size of the node space, but this time is usually relatively small compared to the time required by the mark phase. As a result, it is usually a good idea to use `expand` to make the node space quite large to reduce the frequency of garbage collections.

The optimal size of the node space depends on the amount of system memory available, the operating system, the nature of the load on the system, and

other factors specific to the particular platform. When XLISP-STAT starts up, it attempts to set the size of the node space to a reasonable level with an `expand` command in the file `init.lsp`. But you may be able to improve the performance of the system on a particular platform by experimenting with the allocation.

Index

- ' (quote) read macro, 12, 120
- * function, 10, 11, 158, 200
- * variable, 24
- ** variable, 24
- *** variable, 24
- + function, 10, 11, 15, 99, 128, 158, 200
- + variable, 24
- ++ variable, 24
- +++ variable, 24
- , (comma) read macro, 98, 120
- function, 4, 158
- variable, 24
- / function, 4, 56, 158
- /= function, 33
- ;(comment character), 10
- < function, 33, 79
- <= function, 34
- = function, 79
- > function, 79
- #' read macro, 23, 120
- #(read macro, 140
- #+ read macro, 150
- #- read macro, 150
- #< read macro, 187
- #C read macro, 133, 134
- ^ (exponentiation) function, 56
- ` (backquote) read macro, 98, 120
- :**1stmoments** message
 - for bayes models, 72, 74
- :**abcplane** message
 - for spinning plots, 294
- :**abline** message
 - for graphs, 292
 - for scatterplots, 48
- about-xlisp-stat** function, 361
- abrasion loss data, 37, 41, 51, 317, 321
- abs** function, 60, 134
- abstract data, *see* data abstraction
- accessor functions, 110
- accessor methods, 185, 193, 221
- accumulate** function, 159
- :**action** keyword
 - for sequence-slider-dialog, 225, 226
- :**action** message
 - for dialog items, 228
 - for interval sliders, 226
 - for sequence sliders, 226
- action slot**
 - for dialog items, 228
 - for menu items, 219-221
- :**activate** message
 - for windows, 358
- active-windows** function, 233
- :**add-function** message
 - for graphs, 292
 - for spinning plots, 294
- :**add-lines** message
 - for graphs, 265, 266, 305, 332
 - for histograms, 295
 - for name lists, 296
 - for scatterplot matrices, 293
 - for scatterplots, 49, 50, 292
- :**add-mouse-mode** message
 - for graphs, 274
- :**add-overlay** message
 - for graphs, 287
- :**add-points** message
 - for graphs, 263, 298, 332

for histograms, 52, 181, 294
 for name lists, 295
 for scatterplot matrices, 293
 for scatterplots, 49, 292
:add-slot message
 for objects, 182, 193
:add-subordinate message
 for windows, 217, 299
adjoin function, 139
:adjust-depth-cuing message
 for graphs, 292
:adjust-points-in-rect message
 for graphs, 278, 279, 282, 339
 for histograms, 295
 for name lists, 296
:adjust-screen message
 for graphs, 282, 283, 285, 308
 for histograms, 295
 for name lists, 296
:adjust-screen-point message
 for graphs, 281, 282
 for histograms, 295
 for name lists, 296
 for scatterplot matrices, 293
:adjust-to-data message
 for graphs, 263, 267, 270, 271,
 291, 327, 331
 for histograms, 295
 for scatterplots, 293
 for spinning plots, 293
:all-points-showing-p message
 for graphs, 283, 291
alloc function, 369
ancestors, 180, 182
and macro, 79, 80, 174
:angle message
 for spinning plots, 293, 324
animation, 5, 60, 225, 297
 precomputed, 299
animation techniques, 244
 double buffering, 245
 XOR drawing, 244
anonymous functions, 92
:any-points-selected-p message
 for graphs, 283, 291

APL, 4
append function, 34, 138, 142
:append-items message
 for menus, 220
Apple Macintosh, *see* Macintosh
apple-menu variable, 358
apple-menu-proto prototype, 357
applicative programming, *see* functional programming
apply function, 94–96, 129, 315
:apply-transformation message
 for graphs, 272, 273, 294, 310
apropos function, 26
arc angles, 238
arcs, 238
aref function, 145, 146
arguments, 27
 arbitrary number, 27
 keyword, 27, 126
 default values, 126, 128
 optional, 27, 127
 order, 128, 129
 variable number, 128
array access, 145
array construction, 144
array-dimension function, 147
array-dimensions function, 147
array-in-bounds-p function, 147
array-rank function, 146
array-row-major-index function,
 147
array-total-size function, 147
arrays, 144
:ask-save-image message
 for graphs, 291
aspect ratio, 284
assignment, 100
assoc function, 115
association lists, 115
axes, 263, 267
 in spinning plots, 40

:back-color message
 for graph windows, 239
back-solving, 167, 169

background color, 236
backquote (`) read macro, 98, 120
backquote mechanism, 98
backslash, 120, 121
backsolve function, 169
baktrace function, 151
:basis keyword
 for **:apply-transformation**,
 273
:basis message
 for regression models, 55, 201
bayes model messages
 :1stmoments, 72, 74
 :margin1, 73, 74
 :moments, 73, 74
bayes-model function, 71, 72
Bayesian computing, 70
Bayesian residual plot, 57
best-cursor-size function, 256
beta-cdf function, 163
beta-dens function, 163
beta-quant function, 163
beta-rand function, 31
:bin-counts message
 for histograms, 295
bind-columns function, 164, 172
bind-rows function, 164
bindings, 86
 parallel, 89, 91, 131
 sequential, 90, 91, 131
binomial-cdf function, 163
binomial-dens function, 163
binomial-pmf function, 163
binomial-quant function, 163
binomial-rand function, 31
bitmaps, 238
bivnorm-cdf function, 163
biweight weight function, 173
block structure, 107
bound variable, 87
boundp function, 136
Box-Cox power transformation, *see*
 examples
boxplot function, 15, 17
boxplots, 15
 parallel, 17
break function, 151
breakenable variable, 151
:brush message
 for graphs, 279
brushing, 42, 278
:buffer-to-screen message
 for graph windows, 245, 246
button item messages
 :do-action, 229, 230
button-item-proto prototype, 229

C, xi, 3, 100, 103, 121, 131, 180,
 206, 362, 366
C++, 205
call-cfun function, 362, 366–368
call-fsub function, 366–368
call-lfun function, 366, 367
call-method function, 192, 193
call-next-method function, 191–
 193
canvas, 235
canvas dimensions, 254
 elastic, 254
 fixed, 254
canvas range, 286
:canvas-height message
 for graph windows, 242
:canvas-range message
 for graphs, 286
:canvas-to-real message
 for graphs, 275
:canvas-to-scaled message
 for graphs, 275
:canvas-width message
 for graph windows, 242
car function, 137
case clauses, 114
case conversion, 11
case macro, 114, 129
Cauchy weight function, 173
cauchy-cdf function, 163
cauchy-dens function, 163
cauchy-quant function, 163
cauchy-rand function, 31

cdr function, 137
ceiling function, 132
:center message
 for graphs, 271
centering, 270
char function, 135
characterp function, 135
characters, 134, 238
chisq-cdf function, 163
chisq-dens function, 163
chisq-quant function, 163
chisq-rand function, 31
choice item messages
 :**isnew**, 229
 :**value**, 229
choice-item-proto prototype, 229
chol-decomp function, 168
 Cholesky decomposition, 168
choose-item-dialog function, 223
:choose-mouse-mode message
 for graphs, 291
choose-subset-dialog function,
 223
Classcal, 205
classes, 206
clean-up function, 151
:clear message
 for edit windows, 359
 for graphs, 269, 298
 for histograms, 51
 for scatterplots, 49
:clear-lines message
 for graphs, 269, 305
:clear-points message
 for graphs, 269
 for histograms, 295
 clearing plot data, 269
click events, 248
:click-range message
 for graphs, 276
clip rectangle, 255
:clip-rect message
 for graph windows, 255
clipboard, 24
 clipboard support, 359
clipping, 255
CLOS, 204, 206
 metaclass protocol, 206
close function, 124
:close message
 for windows, 217
close-resource-file function, 361
closure, *see* function closure
:coef-estimates message
 for nonlinear models, 203
 for regression models, 54, 55,
 62, 201
:coef-standard-errors message
 for nonlinear models, 66
 for regression models, 54, 201
coerce function, 141, 142, 171
color-symbols function, 240, 256,
 264
colors, 235, 236
 adding, 256
 RGB system, 256
column-list function, 165
combine function, 34
comma (,) read macro, 98, 120
command-menu variable, 358
comment character (;), 10
comments, 149
compilers, 7
complex function, 28, 133, 134
complex numbers, *see* numbers
complexp function, 80, 132
compound data, 11, 99, 157
compound data object messages
 :**data-length**, 199
 :**data-seq**, 199, 200
 :**make-data**, 199, 200
 :**print**, 200
compound data objects, 199
compound expressions, 11
compound-data-p function, 99, 157
compound-data-proto prototype,
 199
compound-data-seq function, 157,
 158, 171
:compute message

- for nonlinear models, 203
- for regression models, 202
- concatenate** function, 142
- cond** clauses, 80
- cond** macro, 80–82, 129
- condition number, 169
- conditional distribution, 44
- conditional evaluation, 5, 80, 129
- congruent generator, 102
- cons** cells, 137
- cons** function, 116, 138
- constructor functions, 110, 189
- content origin, 286
- content rectangle, 283
 - :content-origin** message
 - for graphs, 286
 - :content-rect** message
 - for graphs, 284
 - :content-variables** message
 - for graphs, 286
 - for spinning plots, 293
- continue** function, 151
- contour-function** function, 59
- control structure, 129
- Cook's distances, 56
 - :cooks-distances** message
 - for regression models, 56, 201
- coordinate systems, 235, 270
 - canvas, 254
 - real, 270
 - scaled, 270
- copy-list** function, 35, 104
- copy-seq** function, 141
 - :copy-to-clip** message
 - for edit windows, 359
 - for graph windows, 255
- copying data, 35, 104
- count-elements** function, 158, 160
 - :count-limit** keyword
 - for nreg-model, 66
- covariance-matrix** function, 160
- cross-product** function, 167
- cumsum** function, 159
- current variables, 267
 - :current-variables** message
- for graphs, 269, 286
- cursor-symbols** function, 250, 256
- cursors, 250, 361
 - adding, 256
- customizing Lisp-Stat, 5
- :cut-to-clip** message
 - for edit windows, 359
- dash-item-proto** prototype, 221
- data abstraction, 109, 185
- data-directed programming, 117
- :data-length** message
 - for compound data objects, 199
- :data-seq** message
 - for compound data objects, 199, 200
- debug** function, 151
- debugging tools, 151
 - break loop, 151
 - stepping, 153
 - tracing, 152
- def** macro, 14, 23, 28, 35, 36, 86, 102
- default button, 223, 227
 - :default-button** message
 - for dialogs, 227
- *default-path*** variable, 150, 364
- defconstant** macro, 155
- defmeth** macro, 63, 184, 196
- defparameter** macro, 155
- defproto** macro, 187, 189, 190, 193–196
- defsetf** macro, 154
- defun** macro, 5, 58, 59, 78, 86, 91, 149
- defvar** macro, 155
- delete** function, 143
 - :delete-documentation** message
 - for objects, 197
 - :delete-items** message
 - for menus, 221
 - :delete-method** message
 - for objects, 186
 - :delete-mouse-mode** message
 - for graphs, 276

:**delete-overlay** message
 for graphs, 287

:**delete-slot** message
 for objects, 183

:**delete-subordinate** message
 for windows, 218

deleting cases, 32, 58

:**delta** keyword
 for lowess, 161

depth cuing, 40

:**depth-cuing** message
 for spinning plots, 293

derivatives
 exact, 69, 75
 numerical, 68, 75, 107
 symbolic, 109

desktop metaphor, 214

destructive modification, 35, 100,
 138

determinant function, 168

:**df** message
 for regression models, 201

diagonal function, 165

dialog item messages
 :**action**, 228
 :**dialog**, 230
 :**do-action**, 227
 :**isnew**, 228

dialog item slots
 action, 228

dialog items, 222, 228
 buttons, 229
 choice items, 229
 scroll bars, 231
 text items, 229
 toggle items, 228

:**dialog** message
 for dialog items, 230

dialog messages
 :**default-button**, 227
 :**isnew**, 227, 228
 :**items**, 227
 :**modal-dialog**, 228, 231
 :**modal-dialog-return**, 228,
 230

 :**remove**, 228

dialog-item-proto prototype, 228

dialog-proto prototype, 226

dialogs, 222
 constructing, 226
 layout, 227, 228, 231
 modal, 222
 modeless, 222
 prototypes, 226
 sliders, 225
 standard, 222

difference function, 159

:**direction** keyword
 for open, 123

dispatching, 117, 181

displaced array, 145

:**displaced-to** keyword
 for make-array, 145

:**display** keyword
 for sequence-slider-dialog,
 225

:**display** message
 for regression models, 54, 201

display windows, 360

display-window-proto prototype,
 360

distributions, *see* probability distributions

do macro, 85, 93, 106, 130–132,
 173

do* macro, 131, 132

:**do-action** message
 for button items, 229, 230
 for dialog items, 227, 228
 for interval scroll bars, 232
 for list items, 233
 for menu items, 219, 220, 325
 for scroll bars, 231
 for sequence scroll bars, 232

:**do-brush-click** message
 for graphs, 278, 279

:**do-brush-motion** message
 for graphs, 278, 279

:**do-click** message

for graph windows, 248–251,
 254
for graphs, 274, 287
for scatterplot matrices, 293
:do-idle message
 for graph windows, 252
 for graphs, 310, 323
 for spinning plots, 293
:do-key message
 for graph windows, 251, 252
:do-motion message
 for graph windows, 248
 for graphs, 274
 for scatterplot matrices, 293
:do-select-click message
 for graphs, 278
:doc-topics message
 for objects, 196
documentation, 149
 for functions, 28, 59, 149
 for methods, 184
 for objects, 195
 for prototypes, 189
 for types, 28, 149
 for variables, 28, 149, 155
documentation function, 149, 155
:documentation message
 for objects, 196
documentation slot
 for objects, 181, 196
dolist macro, 52, 131, 132
dotimes macro, 51, 63, 103, 131,
 132, 245
double buffering, 244, 245, 272
double quote read macro, 120
:drag-grey-rect message
 for graph windows, 250
:drag-point message
 for graphs, 277, 278, 307
dragging, 249
:draw keyword
 for **:add-lines**, 266
 for **:add-points**, 263
 for **:clear**, 51, 269
 for **:current-variables**, 269
 for **:drag-point**, 277
 for **:range**, 267
 for **:transformation**, 272
 for **:x-axis**, 267
 for **:y-axis**, 267
:draw-axes message
 for spinning plots, 293
:draw-bitmap message
 for graph windows, 244
:draw-color message
 for graph windows, 239
:draw-line message
 for graph windows, 241
:draw-mode message
 for graph windows, 239
:draw-point message
 for graph windows, 240
:draw-string message
 for graph windows, 243
:draw-string-up message
 for graph windows, 243
:draw-symbol message
 for graph windows, 243, 248
:draw-text message
 for graph windows, 243
:draw-text-up message
 for graph windows, 243
drawable objects, 237
 arcs, 237, 238
 bitmaps, 238
 ovals, 237
 polygons, 238
 rectangles, 237
 strings, 238
 symbols, 238
drawing color, 236
drawing modes, 235, 237
drawing system states, 237
 colors, 237
 drawing mode, 237
 line type, 237
 line width, 237
dribble function, 23, 24
dyn-load function, 366
dynamic graphics, 1, 3, 37, 213

dynamic linking, *see* dynamic loading
 dynamic loading, 6, 362, 366
 dynamic scoping, *see* scoping rules
 dynamic simulations, 50
 compared to brushing, 52

edit windows messages
 :clear, 359
 :copy-to-clip, 359
 :cut-to-clip, 359
 :edit-selection, 360
 :eval-selection, 360
 :find, 359, 360
 :find-again, 359, 360
 :find-string, 360
 :flush-window, 360
 :paste-from-clip, 359
 :paste-stream, 359, 360
 :paste-string, 359, 360
 :remove, 359
 :revert, 360
 :save, 360
 :save-as, 360
 :save-copy, 360
 :selection-stream, 359, 360
edit-menu variable, 358
:edit-selection message
 for edit windows, 360
edit-text-item-proto prototype,
 229
edit-window-proto prototype, 359,
 360
 editor windows, 357
eigen function, 169
eigenvalues function, 169
eigenvectors function, 169
element-seq function, 157, 158,
 160
 elementwise arithmetic, *see* vector-
 ized arithmetic
elt function, 140, 141
:enabled message
 for menus, 219
enabled slot
 for menu items, 219–221
 for menus, 219, 221
environments, 86
:epsilon keyword
 for **nreg-model**, 66
eq function, 104, 105, 112, 136,
 138, 141
eq1 function, 104, 105, 139, 143
equal function, 105
equality, 104
equalp function, 105, 140
:erase-arc message
 for graph windows, 241
:erase-oval message
 for graph windows, 241
:erase-poly message
 for graph windows, 241
:erase-rect message
 for graph windows, 241, 258
:erase-selection message
 for graphs, 283, 291, 338
:erase-window message
 for graph windows, 242
 erasing, 236
error function, 111, 148
 errors, 148
 escape characters, 117, 120
:eval-selection message
 for edit windows, 360
 evaluation rules, 10
 event loop, 214
 events, 214, 246
 click, 248
 exposure, 247
 key, 251
 motion, 248
 mouse, 248, 274
 resize, 247
every function, 142
 examples
 alternate data representation,
 209, 334
 alternative linking strategy, 334
 binomial parameters dialog, 232
 bitmap editor, 257

factorial function, 84, 100, 103, 126, 128, 131, 152
grand tours, 321
graphical function input, 310
hand rotation, 309
interpolation button, 287
Newton's method, 106
parallel coordinates, 329
plot control overlays, 313
plot interpolation, 300
power transformations, 5, 60, 297
projection, 89, 91, 95, 171
rectangular data sets, 207
regression dialog, 226
regression leverage, 306
smoothing, 304
symbolic differentiation, 109
exemplars, 206
exit function, 13
exp function, 73, 113
expand function, 369, 370
experimental programming, 206
exponentiation (^) function, 56
exposure events, 247
expressions as data, 2, 96
extend dragging, 43
extend modifier, *see* modifiers, 365
extending selections, 214
external functions, *see* foreign functions
:externally-studentized-residuals message
for regression models, 58, 201

:f keyword
for **lowess**, 161
F statistic, 58
f-cdf function, 163
f-dens function, 163
f-quant function, 163
f-rand function, 31
fboundp function, 136
features variable, 150, 233, 357, 364, 365

file-menu variable, 358
file-position function, 125
files, 6, 36, 119, 123
find function, 143
:find message
for edit windows, 359, 360
:find-again message
for edit windows, 359, 360
find-menu function, 358
:find-string message
for edit windows, 360
first function, 96, 137, 277
:fit-values message
for nonlinear models, 203
for regression models, 201
fivnum function, 161
:fixed-aspect message
for graphs, 285
Flavors, 204
flet special form, 91, 96, 107
float function, 132, 133
floating point numbers, *see* numbers
floatp function, 132
floor function, 132
:flush-window message
for edit windows, 360
focus window, 247
:focus-on-selection message
for graphs, 283, 291, 338
font
for graph windows, 243
italic, xi
typewriter, xii
force-output function, 125
foreign functions, 6, 362, 366
format function, 121–123, 125, 225
FORTRAN, xi, 3, 32, 100, 366–368

:frame-arc message
for graph windows, 241
:frame-location message
for windows, 216
:frame-oval message
for graph windows, 241

:frame-poly message
 for graph windows, 241
:frame-rect message
 for graph windows, 241
:frame-size message
 for windows, 216
free variable, 87
free-color function, 256
free-cursor function, 257
front-window function, 358
funcall function, 94–96, 103, 220
function application, 11, 78
function arguments, *see* **arguments**
function closure, 93
function definitions, 23, 25, 27, 58
function special form, 23, 59, 93
functional data, 2, 93, 95
functional programming, 77, 101
functions as arguments, 59, 93
functions as results, 95

gamma distribution, 67
gamma-cdf function, 163
gamma-dens function, 163
gamma-quant function, 163
gamma-rand function, 31
garbage collection, 52, 195, 369
Gauss-Newton algorithm, 66, 203
gc function, 369
generalized variables, 103, 136, 141,
 154
generating data
 random, 31
 systematic, 29
generic functions, 189
gensym function, 209
get-internal-real-time function,
 154
get-internal-run-time function,
 154
get-nice-range function, 268
get-string-dialog function, 224,
 259
get-value-dialog function, 224
global environment, 87

global functions, 86
global variables, 86
:go-away keyword
 for **:isnew**, 217
gradient, 68
grand tours, 321, 327
 using histograms, 327
 using scatterplot matrices, 328
graph messages
:abline, 292
:add-function, 292
:add-lines, 265, 305, 332
:add-mouse-mode, 274
:add-overlay, 287
:add-points, 263, 298, 332
:adjust-depth-cuing, 292
:adjust-points-in-rect, 278,
 282, 339
:adjust-screen, 282, 283, 285,
 308
:adjust-screen-point, 281,
 282
:adjust-to-data, 263, 267,
 270, 271, 291, 327, 331
:all-points-showing-p, 283,
 291
:any-points-selected-p, 283,
 291
:apply-transformation, 272,
 273, 294, 310
:ask-save-image, 291
:brush, 279
:canvas-range, 286
:canvas-to-real, 275
:canvas-to-scaled, 275
:center, 271
:choose-mouse-mode, 291
:clear, 269, 298
:clear-lines, 269, 305
:clear-points, 269
:click-range, 276
:content-origin, 286
:content-rect, 284
:content-variables, 286
:current-variables, 269, 286

:delete-mouse-mode, 276
:delete-overlay, 287
:do-brush-click, 278, 279
:do-brush-motion, 278, 279
:do-click, 274, 287
:do-idle, 310, 323
:do-motion, 274
:do-select-click, 278
:drag-point, 277, 278, 307
:erase-selection, 283, 291,
 338
:fixed-aspect, 285
:focus-on-selection, 283, 291,
 338
:idle-on, 324
:isnew, 262, 271, 290
:last-point-state, 281
:linestart-canvas-coordinate,
 286
:linestart-color, 266
:linestart-coordinate, 266,
 305
:linestart-next, 266
:linestart-transformed-co-
 ordinate, 274
:linestart-type, 266
:linestart-width, 266
:linked, 280, 291
:links, 280
:margin, 284, 285, 321
:menu-template, 290, 291, 325,
 331, 338
:menu-title, 290, 291
:mouse-mode, 275
:mouse-modes, 274
:needs-adjusting, 282, 308
:new-menu, 290, 291
:num-variables, 262
:point-canvas-coordinate,
 286
:point-color, 263, 264, 279
:point-coordinate, 265, 298-
 300
:point-hilited, 283
:point-label, 263, 264
:point-selected, 283
:point-showing, 283
:point-state, 264, 281, 338
:point-symbol, 263, 264
:point-transformed-coordi-
 nate, 273
:points-hilited, 283
:points-in-rect, 276, 277,
 339
:points-selected, 283
:points-showing, 283
:range, 267, 301
:real-to-canvas, 276
:redraw, 261, 266, 267, 269,
 285, 287, 291, 321
:redraw-background, 285
:redraw-content, 285, 286,
 292, 299, 332, 333
:redraw-overlays, 285, 287
:resize, 261, 267, 283, 285-
 287, 321, 332
:resize-brush, 279
:resize-overlays, 287
:rotate-2, 273, 285, 294
:scale, 271
:scale-type, 270, 271
:scaled-range, 270, 271
:scaled-to-canvas, 276
:selection-dialog, 291
:set-options, 291
:set-selection-color, 290
:set-selection-symbol, 291
:show-all-points, 283, 291,
 338
:showing-labels, 291, 292
:transformation, 272, 273,
 295
:unselect-all-points, 278,
 279, 338
:variable-label, 262, 320
:variable-labels, 262
:visible-range, 271
:while-button-down, 275, 310,
 316
:x-axis, 267, 285

:y-axis, 267, 285
graph window messages
:back-color, 239
:buffer-to-screen, 245, 246
:canvas-height, 242
:canvas-width, 242
:clip-rect, 255
:copy-to-clip, 255
:do-click, 248–251, 254
:do-idle, 252
:do-key, 251, 252
:do-motion, 248
:drag-grey-rect, 250
:draw-bitmap, 244
:draw-color, 239
:draw-line, 241
:draw-mode, 239
:draw-point, 240
:draw-string, 243
:draw-string-up, 243
:draw-symbol, 243, 248
:draw-text, 243
:draw-text-up, 243
:erase-arc, 241
:erase-oval, 241
:erase-poly, 241
:erase-rect, 241, 258
:erase-window, 242
:frame-arc, 241
:frame-oval, 241
:frame-poly, 241
:frame-rect, 241
:h-scroll-incs, 255
:has-h-scroll, 254
:has-v-scroll, 254
:idle-on, 252
:isnew, 239, 254, 262
:line-type, 239
:line-width, 239
:menu, 253
:paint-arc, 241
:paint-oval, 241
:paint-poly, 241
:paint-rect, 241, 258
:redraw, 240, 247, 248, 255, 359
:replace-symbol, 244, 282
:resize, 247, 248, 255, 359
:reverse-colors, 240
:save-image, 255
:scroll, 255
:start-buffering, 245, 246
:text-ascent, 243
:text-descent, 243
:text-width, 243
:use-color, 240
:v-scroll-incs, 255
:view-rect, 255
:while-button-down, 249, 251
graph-overlay-proto prototype, 287
graph-proto prototype, 261, 262, 267, 269, 271, 273, 274, 278–280, 283, 285–287, 290, 292, 302, 329, 330, 332, 334, 358
graph-window-proto prototype, 239, 262
graphical model, 235
graphical user interfaces, 213
graphics windows, 235
:h-scroll-incs message
for graph windows, 255
hardware model, 7
:has-h-scroll message
for graph windows, 254
:has-method message
for objects, 186
:has-slot message
for objects, 183, 186
:has-v-scroll message
for graph windows, 254
help function, 25–27, 59, 149, 155
:help message
for nonlinear models, 66
for objects, 181, 184, 189, 196
for regression models, 54
for scatterplots, 48, 49

help system, 25
help* function, 25, 26, 149, 155
Hessian matrix, 69
:hide-window message
 for windows, 217
hierarchical menus, 218
highlighting, 43
histogram function, 15, 181, 215
histogram messages
 :add-lines, 295
 :add-points, 52, 181, 294
 :adjust-points-in-rect, 295
 :adjust-screen, 295
 :adjust-screen-point, 295
 :adjust-to-data, 295
 :bin-counts, 295
 :clear, 51
 :clear-points, 295
 :num-bins, 295
 :point-hilited, 51
 :point-selected, 51
 :point-showing, 51
 :redraw, 295
 :redraw-content, 295
 :resize, 295
histogram-proto prototype, 62, 294
histograms, 294
 dimensions, 294
history mechanism, 24
Huber weight function, 173

identity-matrix function, 165
idle actions, 252
 and errors, 252
:idle-on message
 for graph windows, 252
 for graphs, 324
if special form, 81, 82, 85, 129,
 220, 258
if-else function, 100, 312
imagpart function, 134
indentation, 149
index base, 32
inheritance, 62, 180, 181, 187, 189
 as analogy, 202, 203
 as specialization, 202
:initial keyword
 for choose-item-dialog, 223
 for choose-subset-dialog, 224
 for get-value-dialog, 225
:initial-contents keyword
 for make-array, 145
:initial-element keyword
 for make-array, 144
:initial-value keyword
 for reduce, 142
inner-product function, 78, 166
input/output, 119
:install message
 for menus, 221
instance slots, 189
instance-slots slot
 for objects, 181, 187, 188, 193,
 194
instances, 181, 188
 constructing, 187
integerp function, 132
integers, *see* numbers
intercept, 53
:intercept keyword
 for regression-model, 53
:intercept message
 for regression models, 201
internal-time-units-per-second
 variable, 154
interpolation, 161
interquartile-range function, 15,
 161
interrupt, 357, 364
interrupting calculations, 29, 246,
 357, 364
intersection function, 139
interval scroll bars
 :do-action, 232
 :isnew, 232
interval slider messages
 :action, 226
 :value, 226
interval-scroll-item-proto proto-
 type, 232

interval-slider-dialog function, 226, 299, 301
inverse function, 168
iseq function, 19, 29, 159
:isnew message
 for choice items, 229
 for dialog items, 228
 for dialogs, 227, 228
 for graph windows, 239, 254, 262
 for graphs, 262, 271, 290
 for interval scroll bars, 232
 for menu items, 357, 358
 for menus, 219
 for name lists, 296
 for objects, 188, 189, 192
 for regression models, 202
 for scroll bars, 231
 for sequence scroll bars, 232
 for spinning plots, 293, 294
 for toggle items, 228
 for windows, 215–217
:items message
 for dialogs, 227
 for menus, 219
items slot
 for menus, 221
iteration, 82
iterative process, 84
iteratively reweighted least squares, 173
Jacobian, 203
kernel-dens function, 162, 304
kernel-smooth function, 161, 162
key events, 251
&key lambda-list keyword, 126, 127, 129, 184
:key message
 for menu items, 358
keyboard equivalents, 358
keyword arguments, 27, 126
 default values, 126, 128
keyword symbols, 27, 156
kind-of-p function, 194
:labels keyword
 for plot-function, 123
labels special form, 91, 96, 174
lambda calculus, 92
lambda expression, 92
:last-point-state message
 for graphs, 281
length function, 33, 140, 141
let special form, 59–61, 88–91, 97, 101, 108, 131, 194, 244
let* special form, 60, 61, 90, 91, 127, 131
:levels keyword
 for contour-function, 59
:leverages message
 for nonlinear models, 66
 for regression models, 56, 201
lexical analyzer, 120
lexical scoping, *see* scoping rules
line type, 237
line width, 237
:line-type message
 for graph windows, 239
:line-width message
 for graph windows, 239
linear algebra floating point type, 170
linear algebra functions, 167
linear recursion, 84
:linestart-canvas-coordinate message
 sage
 for graphs, 286
:linestart-color message
 for graphs, 266
:linestart-coordinate message
 for graphs, 266, 305
:linestart-next message
 for graphs, 266
:linestart-transformed-coordinate message
 for graphs, 274
:linestart-type message
 for graphs, 266

:linestart-width message
 for graphs, 266
linestarts, 265
:linked message
 for graphs, 280, 291
linked plots, 46, 55, 264, 280
 alternative strategies, 334
 protocol, 281
linked-plots function, 280
:links message
 for graphs, 280
Lisp reader, 119
list function, 11, 13, 14, 23, 105
list item messages
 :do-action, 233
 :new, 233
 :selection, 233
 :set-text, 233
list structure, 137
list-item-proto prototype, 233
listener variable, 360
listener window, 357
listener-proto prototype, 360
lists, 11, 137
 as sets, 139
load function, 24, 357
load-data function, 368
load-example function, 368
local functions, 91
local variables, 59, 60, 88, 101
:location message
 for windows, 216
log function, 16, 100, 133, 158, 200
log-gamma function, 68
logical expressions, 79
logical values, 10
loop macro, 131
looping, 51, 85, 130
LOWESS algorithm, 161
lowess function, 161
LU decomposition, 167
lu-decomp function, 167, 168
lu-solve function, 167, 168
machine-epsilon variable, 170
Macintosh, 3, 6, 13, 24, 29, 40, 43, 135, 149–151, 153, 205, 206, 213–215, 217, 219, 222, 233, 235, 249, 253, 255, 317, 355–359, 362–365
Macintosh dialogs, 361
make-array function, 144, 145
make-color function, 256
make-cursor function, 256, 257, 361
:make-data message
 for compound data objects, 199, 200
make-object function, 193
make-projection function, 171
:make-prototype message
 for objects, 193, 194
make-random-state function, 163, 164
make-rotation function, 170, 310, 323, 324
make-sweep-matrix function, 170
map function, 142, 145, 158
map-elements function, 99, 100, 159, 199
mapcar function, 98, 99, 158, 165
mapping, 98, 142, 158
:margin message
 for graphs, 284, 285, 321
:margin1 message
 for bayes models, 73, 74
margins, *see* plot margins
:mark message
 for menu items, 220
mark slot
 for menu items, 219, 221
mask bitmaps, 239
matmult function, 166
matrix functions, 164
matrix multiplication, 166
matrixp function, 164
max function, 26, 60, 160
:max-value message

for scroll bars, 231
maximization, 67
 Nelder-Mead method, 67
 Newton's method, 67
maximum likelihood, 67, 108
 standard errors, 69
mean function, 4, 14, 149, 160
mean-function slot
 for nonlinear models, 202
median function, 14, 25, 161
member function, 139
memory management, 368
menu item messages
 :**do-action**, 219, 220, 325
 :**isnew**, 357, 358
 :**key**, 358
 :**mark**, 220
 :**menu**, 221
 :**style**, 357
 :**update**, 219, 220, 324
menu item slots
action, 219-221
enabled, 219-221
mark, 219, 221
title, 219, 221
:menu message
 for graph windows, 253
 for menu items, 221
menu slots
enabled, 219, 221
items, 221
title, 219, 221
menu-item-proto prototype, 219,
 220
menu-proto prototype, 218, 219
:menu-template message
 for graphs, 290, 291, 325, 331,
 338
:menu-title message
 for graphs, 290, 291
menus, 218
 hierarchical, 218, 233
menus messages
 :**append-items**, 220
 :**delete-items**, 221
 :**enabled**, 219
 :**install**, 221
 :**isnew**, 219
 :**items**, 219
 :**popup**, 254
 :**remove**, 221
 :**title**, 219
message arguments, 181
message passing, 180
message selectors, 181
message-dialog function, 222
messages, 47, 180
:method-selectors message
 for objects, 186
methods, 62, 180, 181
 defining, 62, 184
 overriding, 191
Michaelis-Menten model, 64
min function, 26, 60, 160
:min-value message
 for scroll bars, 231
mixins, 205, 323
modal dialogs, *see dialogs*
modal-button-proto prototype, 230
:modal-dialog message
 for dialogs, 228, 231
modal-dialog-proto prototype, 226
:modal-dialog-return message
 for dialogs, 228, 230
modeless dialogs, *see dialogs*
modifiers, 215, 251
 extend, 40, 249, 364
 option, 249
modules, 149
***modules* variable**, 149
:moments message
 for bayes models, 73, 74
motion events, 248
mouse events, 248
mouse modes, 43, 274, 306
 standard, 278
:mouse-mode message
 for graphs, 275
:mouse-modes message
 for graphs, 274

multiple inheritance, 203
multiple regression, *see* regression
multiple screens, 216, 233

name lists, 295
name lists messages
 :*add-lines*, 296
 :*add-points*, 295
 :*adjust-points-in-rect*, 296
 :*adjust-screen*, 296
 :*adjust-screen-point*, 296
 :*isnew*, 296
 :*redraw-content*, 296
name-list function, 46, 47, 255
name-list-proto prototype, 295
:*needs-adjusting* message
 for graphs, 282, 308
:*needs-computing* message
 for regression models, 202
Nelder-Mead method, 69
nelmeadmax function, 69
:*new* message
 for list items, 233
 for objects, 182, 188, 194, 195
 for regression models, 202
:*new-initial-guess* message
 for nonlinear models, 203
:*new-menu* message
 for graphs, 290, 291
Newton's method, 67, 83, 106
newtonmax function, 67–69
:*nice* keyword
 for **sequence-slider-dialog**,
 226
nodebug function, 151
nonlinear model messages
 :*coef-estimates*, 203
 :*coef-standard-errors*, 66
 :*compute*, 203
 :*fit-values*, 203
 :*help*, 66
 :*leverages*, 66
 :*new-initial-guess*, 203
 :*parameter-names*, 203
 :*save*, 203

nonlinear model slots
 :*mean-function*, 202
 :*predictor-names*, 203
 :*theta-hat*, 202
 :*x*, 203

nonlinear regression, 64, 202
normal-cdf function, 163
normal-dens function, 163
normal-quant function, 163
normal-rand function, 31
not function, 79, 80
notation, xi
nreg-model function, 64–66
nreg-model-proto prototype, 66,
 202

null environment, *see* global environment

:*num-bins* message
 for histograms, 295

:*num-cases* message
 for regression models, 201

:*num-coefs* message
 for regression models, 201

:*num-included* message
 for regression models, 201

:*num-points* keyword
 for **contour-function**, 59
 for **spin-function**, 59

:*num-variables* message
 for graphs, 262

numberp function, 80, 111, 132

numbers, 10, 132
 complex, 133
 complex contagion, 134
 floating point, 119, 132
 integer, 132
 rational, 132
 reading, 119
 real, 132
 type coercion, 133

numerical integration, 93

numgrad function, 68

numhess function, 69

object messages

:add-slot, 182, 193
:delete-documentation, 197
:delete-method, 186
:delete-slot, 183
:doc-topics, 196
:documentation, 196
:has-method, 186
:has-slot, 183, 186
:help, 181, 184, 189, 196
:isnew, 188, 189, 192
:make-prototype, 193, 194
:method-selectors, 186
:new, 182, 188, 194, 195
:own-methods, 186
:own-slots, 181
:parents, 194
:precedence-list, 194
:print, 186, 187
:reparent, 194
:save, 197
:slot-names, 183
:slot-value, 181–183, 185
Object Pascal, 205
object prototype, 181, 182
object slots
 documentation, 181, 196
 instance-slots, 181, 187, 188,
 193, 194
 proto-name, 181, 187, 193
object system, 179
object systems
 class-based, 206
 prototype-based, 206
object-oriented programming, 2, 5,
 179
 and graphics, 2
 and models, 2
 and user interfaces, 214
Objective C, 205
objectp function, 180
objects, 47, 179, 180
 constructing, 182, 188
 initializing, 188
 printing, 181, 186, 188
 saving, 197
ok-or-cancel-dialog function, 223
open function, 123, 124
open-file-dialog function, 361
open-resource-file function, 361,
 362
operating system interface, 358, 366
operating systems, 6
option modifier, *see* modifiers
optional arguments, 27, 127
&optional lambda-list keyword, 127,
 129, 184
or macro, 79, 80
order function, 52, 160
outer-product function, 166, 167
ovals, 238
overlays, *see* plot overlays
overriding methods, 191
:own keyword
 for **:has-slot**, 183
:own-methods message
 for objects, 186
:own-slots message
 for objects, 181

:paint-arc message
 for graph windows, 241
:paint-oval message
 for graph windows, 241
:paint-poly message
 for graph windows, 241
:paint-rect message
 for graph windows, 241, 258
painting, 237
parallel boxplots, *see* boxplots
:parameter-names message
 for nonlinear models, 203
parents, 182, 189
:parents message
 for objects, 194
Pascal, 206
:paste-from-clip message
 for edit windows, 359
:paste-stream message
 for edit windows, 359, 360
:paste-string message

for edit windows, 359, 360
pause function, 52, 245
permute-array function, 166
phase function, 134
pi constant, 11, 19, 136, 156
:pivot keyword
 for **qr-decomp**, 169
pixels, 235, 236
place form, 103
plot interpolation, 300
 convex, 300
 trigonometric, 300
plot margins, 283, 314
plot menus, 6, 40, 63, 215, 253,
 290, 364
plot operations
 labeling, 45
 linking, *see* linked plots
 rescaling, 45
plot overlays, 285, 286
plot selection
 changing colors, 45
 changing symbols, 45
 removing, 45
 saving indices, 45
 setting indices, 46
:plot-bayes-residuals message
 for regression models, 57, 201
plot-function function, 22, 59,
 93, 123
plot-lines function, 19, 73, 292,
 364
plot-points function, 19, 21, 47,
 50, 292, 301, 302, 364
:plot-residuals message
 for regression models, 55, 201
plot-symbol-symbols function, 243,
 264
plotting functions, 22, 59
 contour plots, 59
 spinning plots, 59
pmax function, 160
pmin function, 160
point states, 264, 281, 283
:point-canvas-coordinate mes-
 sage
 for graphs, 286
:point-color message
 for graphs, 263, 264, 279
:point-coordinate message
 for graphs, 265, 298–300
:point-hilited message
 for graphs, 283
 for histograms, 51
:point-label message
 for graphs, 263, 264
:point-labels keyword
 for **:add-points**, 263
:point-selected message
 for graphs, 283
 for histograms, 51
:point-showing message
 for graphs, 283
 for histograms, 51
:point-state message
 for graphs, 264, 281, 338
:point-symbol message
 for graphs, 263, 264
:point-transformed-coordinate
 message
 for graphs, 273
:points keyword
 for **sequence-slider-dialog**,
 226
:points-hilited message
 for graphs, 283
:points-in-rect message
 for graphs, 276, 277, 339
:points-selected message
 for graphs, 283
:points-showing message
 for graphs, 283
poisson-cdf function, 163
poisson-dens function, 163
poisson-pmf function, 163
poisson-quant function, 163
poisson-rand function, 31
polygons, 238, 332
 absolute coordinates, 238

- relative coordinates, 238
- :popup message**
 - for menus, 254
- position function**, 143
- posterior distributions, 70
 - approximate marginal densities, 70
 - approximate moments, 70
- PostScript*, 24, 364, 365
- power transformations, 297
- precedence list, 62, 182, 191, 204
 - rules for constructing, 204
- :precedence-list message**
 - for objects, 194
- predicate naming, 83
- predicates, 79
- predictor-names slot**
 - for nonlinear models, 203
- prefix notation, 10
- prerequisites, xi
- press button controls, 313
- prin1 function**, 120
- princ function**, 117, 120, 121
- print function**, 81, 120, 121, 123, 125, 128, 166, 225
- :print keyword**
 - for bayes-model, 72
 - for nreg-model, 66
 - for regression-model, 53, 54, 201
- :print message**
 - for compound data objects, 200
 - for objects, 186, 187
- print-matrix function**, 165, 166, 169
- printing, 120
- printing objects, 181, 186
- probability distributions, 162
 - beta, 31, 162, 163
 - binomial, 31, 162, 163
 - bivariate normal, 162, 163
 - Cauchy, 31, 162, 163
 - χ^2 , 31, 162, 163
 - F*, 31, 162, 163
 - gamma, 31, 67, 108, 162, 163
- normal, 31, 162, 163
- Poisson, 31, 162, 163
- t*, 31, 162, 163
- uniform, 31, 162, 163
- probability plot, 297
- procedural programming, 100, 103
- prod function**, 26, 160
- programming styles, 5
- proto-name slot**
 - for objects, 181, 187, 193
- prototypes, 62, 181, 187
 - constructing, 187
- provide function**, 149, 150
- QR decomposition, 168
- qr-decomp function**, 168, 169
- quantile function**, 160, 161
- quitting from Lisp-Stat, 13
- quote ('') read macro, 12, 120
- quote special form, 12
- quoting, 12
- :r-squared message**
 - for regression models, 201
- random function**, 252, 324
- random number generators, 163
- random sample, *see* <sample>
- *random-state* variable**, 163, 164
- RANDU, 41
- :range message**
 - for graphs, 267, 301
- rank function**, 160, 298
- raster displays, 235
- rational numbers, *see* numbers
- :raw-residuals message**
 - for regression models, 201
- rcondest function**, 169
- read function**, 124, 125, 259
- read macros, 120
- read-char function**, 125
- read-data-columns function**, 36, 37
- read-data-file function**, 36, 37
- *read-default-float-format* variable**, 120

reader methods, 201, 209
reading, 119
reading data, 36
:real-to-canvas message
 for graphs, 276
realpart function, 134
rectangle coordinates, 237
rectangles, 237
recursion, 82
recursive process, 84
recursive vectorization, 159
:redraw message
 for graph windows, 240, 247,
 248, 255, 359
 for graphs, 261, 266, 267, 269,
 285, 287, 291, 321
 for histograms, 295
 for scatterplots, 246
:redraw-background message
 for graphs, 285
:redraw-content message
 for graphs, 285, 286, 292, 299,
 332, 333
 for histograms, 295
 for name lists, 296
 for scatterplot matrices, 293
 for scatterplots, 246
 for spinning plots, 293
:redraw-overlays message
 for graphs, 285, 287
reduce function, 142
referential transparency, 101
regression, 53, 201
 computations, 170
 degeneracy, 54
regression model messages
 :basis, 55, 201
 :coef-estimates, 54, 55, 62,
 201
 :coef-standard-errors, 54,
 201
 :compute, 202
 :cooks-distances, 56, 201
 :df, 201
 :display, 54, 201
 :externally-studentized-residuals, 58, 201
 :fit-values, 201
 :help, 54
 :intercept, 201
 :isnew, 202
 :leverages, 56, 201
 :needs-computing, 202
 :new, 202
 :num-cases, 201
 :num-coefs, 201
 :num-included, 201
 :plot-bayes-residuals, 57,
 201
 :plot-residuals, 55, 201
 :r-squared, 201
 :raw-residuals, 201
 :residuals, 56, 201
 :save, 202
 :sigma-hat, 56, 201
 :studentized-residuals, 201
 :sum-of-squares, 201
 :x, 201
 :x-matrix, 55, 201
 :xtxinv, 55, 201
 :y, 201
regression-model function, 53, 54,
 56, 201, 202
regression-model-proto prototype,
 62, 201, 202
rem function, 102
remove function, 33, 140, 143
:remove message
 for dialogs, 228
 for edit windows, 359
 for menus, 221
 for windows, 217, 218
remove-duplicates function, 143
remove-if function, 143
remove-if-not function, 143
:reparent message
 for objects, 194
repeat function, 29, 30, 159
:replace-symbol message
 for graph windows, 244, 282

require function, 149, 150
rescaling plots, 263
reset-graphics-buffer function,
 246
residual plot, 55
residuals, 55
:residuals message
 for regression models, 56, 201
resize events, 247
:resize message
 for graph windows, 247, 248,
 255, 359
 for graphs, 261, 267, 283, 285–
 287, 321, 332
 for histograms, 295
 for spinning plots, 293
:resize-brush message
 for graphs, 279
:resize-overlays message
 for graphs, 287
resources, 361, 365
rest function, 97, 137
&rest lambda-list keyword, 128,
 129, 184
return macro, 132
:return-derivs keyword
 for **newtonmax**, 69
reusing code, 180
reverse function, 143
:reverse-colors message
 for graph windows, 240
:revert message
 for edit windows, 360
robust regression, 173
room function, 369
root object, 181, 182, 184, 195
:rotate message
 for spinning plots, 293, 294,
 321
:rotate-2 message
 for graphs, 285, 294
rotating plots, *see* spinning plots
rotation angle, 273
rotation controls, 293
 axis rotation, 320
 rocking, 316
rotation matrix, 170
:rotation-type message
 for spinning plots, 293
round function, 132, 248
row-list function, 165
rseq function, 19, 29, 159
 S, xi, xii, 3, 157, 180, 206, 366
sample function, 31, 32
:save message
 for edit windows, 360
 for nonlinear models, 203
 for objects, 197
 for regression models, 202
:save-as message
 for edit windows, 360
:save-copy message
 for edit windows, 360
:save-image message
 for graph windows, 255
savevar function, 24, 197
saving
 objects, 197
 plot images, 23, 255
 session transcripts, 23
 variables, 24
:scale keyword
 for **scatterplot-matrix**, 45
 for **spin-plot**, 39, 40
:scale message
 for graphs, 271
:scale-type keyword
 for **plot-points**, 302
:scale-type message
 for graphs, 270, 271
:scaled-range message
 for graphs, 270, 271
:scaled-to-canvas message
 for graphs, 276
scaling, 269, 270
 fixed, 270
 variable, 270
scatmat-proto prototype, 293
scatterplot matrices, 41, 293

and categorical data, 44
scatterplot matrix messages
 :**add-lines**, 293
 :**add-points**, 293
 :**adjust-screen-point**, 293
 :**do-click**, 293
 :**do-motion**, 293
 :**redraw-content**, 293
scatterplot messages
 :**abline**, 48
 :**add-lines**, 49, 50, 292
 :**add-points**, 292
 :**adjust-to-data**, 293
 :**clear**, 49
 :**help**, 48, 49
 :**redraw-content**, 246
scatterplot-matrix function, 41, 45
scatterplot-proto prototype, 62, 292
scatterplots, 20, 292
scoping rules
 dynamic, 87, 155
 lexical, or static, 87
screen-has-color function, 233
screen-size function, 218, 254
scroll bars item messages
 :**do-action**, 231
 :**isnew**, 231
 :**max-value**, 231
 :**min-value**, 231
 :**scroll-action**, 231
 :**value**, 231
:**scroll** message
 for graph windows, 255
:**scroll-action** message
 for scroll bars, 231
scroll-item-proto prototype, 231
second function, 96
select function, 32, 33, 36, 96, 140, 141, 146, 171
selecting, 42, 278
:**selection** message
 for graphs, 283
 for list items, 233
 :**selection-dialog** message
 for graphs, 291
 :**selection-stream** message
 for edit windows, 359, 360
self variable, 63, 184, 185
send function, 47, 62, 180, 181, 186
sequence scroll bar messages
 :**do-action**, 232
 :**isnew**, 232
sequence slider messages
 :**action**, 226
 :**value**, 226
sequence-scroll-item-proto prototype, 232
sequence-slider-dialog function, 61, 225, 299
sequences, 49, 141
set-difference function, 139
set-file-dialog function, 361
set-menu-bar function, 358
:**set-options** message
 for graphs, 291
:**set-selection-color** message
 for graphs, 290
:**set-selection-symbol** message
 for graphs, 291
:**set-text** message
 for list items, 233
setf macro, 34–36, 101–104, 136, 141, 146, 154, 155, 185
setf methods, 154
shadowing, 87
shared arrays, 145
shared slots, 183, 189, 195
:**show** keyword
 for :**isnew**, 216
:**show-all-points** message
 for graphs, 283, 291, 338
:**show-window** message
 for windows, 217
:**showing-axes** message
 for spinning plots, 293
:**showing-labels** message
 for graphs, 291, 292

side effects, 81, 101, 102
:sigma-hat message
 for regression models, 56, 201
 simple data, 11, 99, 157
 simple regression, *see* regression
 Simula, 205
sin function, 19, 23, 133
 singular value decomposition, 169
:size keyword
 for **nelmeadmax**, 69
:size message
 for windows, 215, 216
 slider dialogs, 61, *see* dialogs
 slot values, 182
:slot-names message
 for objects, 183
slot-value function, 185
:slot-value message
 for objects, 181–183, 185
 slots, 180
 hiding, 193
 initializing, 188, 190
 owned, 183
 shared, 183, 195
 Smalltalk, 205
 smoothing, 161, 304
 software
 obtaining, 356
 software availability, xii
solve function, 168
some function, 129, 142
sort function, 143, 144
sort-data function, 144, 160
 sorting, 143, 160
 special form, 12
 special variables, 155
spin-function function, 59
spin-plot function, 37–39, 41, 45,
 293
spin-proto prototype, 293, 309,
 323–325, 327
 spinning plot messages
 :abcplane, 294
 :add-function, 294
 :adjust-to-data, 293
 :angle, 293, 324
 :content-variables, 293
 :depth-cuing, 293
 :do-idle, 293
 :draw-axes, 293
 :isnew, 293, 294
 :redraw-content, 293
 :resize, 293
 :rotate, 293, 294, 321
 :rotation-type, 293
 :showing-axes, 293
 spinning plots, 37, 293
 and selecting, 44
 axes, 40
 changing origin, 39
 changing speed, 40
 continuous rotation, 40
 depth cuing, 40
spline function, 161, 162
split-list function, 159
sprintf C function, 121
sqrt function, 16
 stack loss data, 175, 209, 261
 standard graphs, 292
standard-deviation function, 4,
 15, 160
standard-input variable, 123
standard-output variable, 123
:start-buffering message
 for graph windows, 245, 246
 starting Lisp-Stat, 6
 state variables, 101
 static scoping, *see* scoping rules
 statistical environments, 1, 3
 statistical languages, 1
step function, 153
:steps keyword
 for **lowess**, 161
 streams, 123
 string input, 125
 string output, 125
string function, 135, 142, 209
 string streams, *see* streams
stringp function, 135
 strings, 10, 134

studentized residuals, 56
:**studentized-residuals** message
 for regression models, 201
:**style** message
 for menu items, 357
subsets, 32
subst function, 97, 105, 140
sum function, 26, 78, 160
:**sum-of-squares** message
 for regression models, 201
SunView, 213–215, 217, 235, 255,
 355, 365
supplied-p arguments, 127, 128
sv-decomp function, 169
sweep operator, 170
sweep-operator function, 171, 177
symbol values, 11, 14, 25, 35
symbol-function function, 136
symbol-name function, 136
symbol-value function, 136
symbolp function, 111
symbols, 135, 238
 case conversion, 11
sysbeep function, 221
system features, 150
system function, 366
system-edit function, 359

t-cdf function, 163
t-dens function, 163
t-rand function, 31
tail recursion, 84
tenth function, 96
terpri function, 81, 117, 120, 121
:**test** keyword
 for **find**, 143
 for **member**, 139
 for **position**, 143
 for **remove**, 143
text item messages
 :**text**, 229
:**text** keyword
 for **sequence-slider-dialog**,
 225, 226
:**text** message

 for text items, 229
text windows, 215
:**text-ascent** message
 for graph windows, 243
:**text-descent** message
 for graph windows, 243
text-item-proto prototype, 229
:**text-width** message
 for graph windows, 243
theta-hat slot
 for nonlinear models, 202
time macro, 154
timing, 154
:**title** keyword
 for **scatterplot-matrix**, 45
 for **sequence-slider-dialog**,
 225, 226
 for **spin-plot**, 39
:**title** message
 for menus, 219
 for windows, 215
title slot
 for menu items, 219, 221
 for menus, 219, 221
toggle item messages
 :**isnew**, 228
 :**value**, 228
toggle-item-proto prototype, 228
top-level function, 151
trace macro, 152
tracelimit variable, 151
tracenable variable, 151
tracing messages, 368
transcendental functions, 133
transformation matrix, 272
:**transformation** message
 for graphs, 272, 273, 295
transformations, 269, 271
transformed data, 273
transpose function, 165
two-button controls, 317
:**type** keyword
 for :**add-lines**, 266
 for **kernel-smooth**, 162

undef function, 28
uniform-rand function, 31
union function, 139
UNIX operating system, xii, 13,
 29, 150, 356, 363, 364, 366
unless macro, 129
:unselect-all-points message
 for graphs, 278, 279, 338
untrace macro, 152
unwind-protect special form, 148
:update message
 for menu items, 219, 220, 324
 for windows, 359
:use-color message
 for graph windows, 240
user interface guidelines, 213
user interfaces, 6

:v-scroll-incs message
 for graph windows, 255
:value message
 for choice items, 229
 for interval sliders, 226
 for scroll bars, 231
 for sequence sliders, 226
 for toggle items, 228
values, 11, 14, 23, 35
:variable-label message
 for graphs, 262, 320
:variable-labels keyword
 for **scatterplot-matrix**, 45
 for **spin-plot**, 39
:variable-labels message
 for graphs, 262
variables
 bound, 87
 free, 87
 listing, 28
 undefined, 28
variables function, 28, 36
vector function, 140
vector reducing, 160
vectorized arithmetic, 3, 15, 78, 99,
 158, 199
recursive vectorization, 159

result structure, 158, 159, 200
vectorp function, 140
vectors, 140
:verbose keyword
 for **bayes-model**, 72
 for **newtonmax**, 68
 for **nreg-model**, 66
:view-rect message
 for graph windows, 255
:visible-range message
 for graphs, 271

:weights keyword
 for **regression-model**, 53
when macro, 129
which function, 33
:while-button-down message
 for graph windows, 249, 251
 for graphs, 275, 310, 316
white space, 11
:width keyword
 for **kernel-dens**, 304
 for **kernel-smooth**, 162
window layout, 283
window messages
 :activate, 358
 :add-subordinate, 217, 299
 :close, 217
 :delete-subordinate, 218
 :frame-location, 216
 :frame-size, 216
 :hide-window, 217
 :isnew, 215–217
 :location, 216
 :remove, 217, 218
 :show-window, 217
 :size, 215, 216
 :title, 215
 :update, 359
window system interface, 213
window systems, 213
window-proto prototype, 215, 218,
 226, 228, 359
windows, 215
 and objects, 216

closing, 217
frame, 216
hiding, 217
location, 215
removing, 217
size, 215
subordinates, 217
title, 215

with-input-from-string macro,
 125, 259

with-open-file macro, 124, 125

with-output-to-string macro, 125

write-char function, 125

x function, 167

:x message
 for regression models, 201

x slot
 for nonlinear models, 203

:x-axis message
 for graphs, 267, 285

:x-matrix message
 for regression models, 55, 201

X11, xii, 3, 24, 40, 213–215, 235,
 253, 255, 355, 364, 365

x11-options function, 365

XOR drawing, 237, 244, 250
 in color, 237

:xtxinv message
 for regression models, 55, 201

:xvals keyword
 for **kernel-smooth**, 162
 for **spline**, 161

:y message
 for regression models, 201

:y-axis message
 for graphs, 267, 285