XLISP-STAT

A Statistical Environment Based on the XLISP Language

(Version 2.0)

by
Luke Tierney

University of Minnesota
School of Statistics
Technical Report Number 528

July 1988

Contents

Preface

XLISP-STAT is a statistical environment built on top of the XLISP programming language. This document is intended to be a tutorial introduction to the basics of XLISP-STAT. It is written primarily for the Apple Macintosh version, but most of the material applies to other versions as well; some points where other versions differ are outlined in an appendix. The first three sections contain the information you will need to do elementary statistical calculations and plotting. The fourth section introduces some additional methods for generating and modifying data. The fifth section describes some additional features of the Macintosh user interface that may be helpful. The remaining sections deal with more advanced topics, such as interactive plots, regression models, and writing your own functions. All sections are organized around examples, and most contain some suggested exercises for the reader.

This document is not intended to be a complete manual. However, documentation for many of the commands that are available is given in the appendix. Brief help messages for these and other commands are also available through the interactive help facility described in Section 5.1 below.

XLISP itself is a high-level programming language developed by David Betz and made available for unrestricted, non-commercial use. It is a dialect of Lisp, most closely related to the Common Lisp dialect. XLISP also contains some extensions to Lisp to support object-oriented programming. These facilities have been modified in XLISP-STAT to implement the screen menus, plots and regression models. Several excellent books on Common Lisp are available. One example is Winston and Horn [22]. A book on XLISP itself has recently been published. Unfortunately it is based on XLISP 1.7, which differs significantly from XLISP 2.0, the basis of XLISP-STAT 2.0.

XLISP-STAT was originally developed for the Apple Macintosh. It is now also available for UNIX systems using the *X11* window system, for *Sun* workstations under the *SunView* window system, and, with only rudimentary graphics, for generic 4.[23]BSD UNIX systems. The Macintosh version of XLISP-STAT was developed and compiled using the Lightspeed C compiler from Think Technologies, Inc. The Macintosh user interface is based on Paul DuBois' TransSkel and TransEdit libraries. Some of the linear algebra and probability functions are based on code given in Press, Flannery, Teukolsky and Vetterling [14]. Regression computations are carried out using the sweep algorithm as described in Weisberg [21].

This tutorial has borrowed several ideas from Gary Oehlert's *MacAnova User's Guide* [13]. Many of the on-line help entries have been adopted directly or with minor modifications from the *Kyoto Common Lisp System*. Most of the examples used in this tutorial have been taken from Devore and Peck [11]. Many of the functions added to XLISP-STAT were motivated by similar functions in the *S* statistical environment [2,3].

The present version of XLISP-STAT, Version 2.0, seems to run fairly comfortably on a Mac II or Mac Plus with 2MB of memory, but is a bit cramped with only 1MB. It will not run in less than 1Mb of memory. The program will occasionally bomb with an ID=28 if it gets into a recursion that is too deep for the Macintosh stack to handle. On a 1MB Mac it may also bomb with an ID=15 if too much memory has been used for the segment loader to be able to bring in a required code segment.

Development of XLISP-STAT was supported in part by grants of an Apple Macintosh Plus computer and hard disk and a Macintosh II computer from the MinneMac Project at the University of Minnesota, by a single quarter leave granted to the author by the University of Minnesota, by grant DMS-8705646 from the National Science Foundation, and by a research contract with Bell Communications Research.

Using this Tutorial

The best way to learn about a new computer program is usually to use it. You will get most out of this tutorial if you read it at your computer and work through the examples yourself. To make this

easier the named data sets used in this tutorial have been stored on the file `tutorial.lsp` in the `Data` folder of the Macintosh distribution disk. To read in this file select the **Load** item on the **File** menu. This will bring up an **Open File** dialog window. Use this dialog to open the `Data` folder on the distribution disk. Now select the file `tutorial.lsp` and press the **Open** button. The file will be loaded and some variables will be defined for you. [1]

Why XLISP-STAT Exists

There are three primary reasons behind my decision to produce the XLISP-STAT environment. The first is to provide a vehicle for experimenting with dynamic graphics and for using dynamic graphics in instruction. Second, I wanted to be able to experiment with an environment supporting functional data, such as mean functions in nonlinear regression models and prior density and likelihood functions in Bayesian analyses. Finally, I was interested in exploring the use of object-oriented programming ideas for building and analyzing statistical models. I will discuss each of these points in a little more detail in the following paragraphs.

The development of high resolution graphical computer displays has made it possible to consider the use of dynamic graphics for understanding higher-dimensional structure. One of the earliest examples is the real time rotation of a three dimensional point cloud on a screen – an effort to use motion to recover a third dimension from a two dimensional display. Other techniques that have been developed include *brushing* a scatterplot – highlighting points in one plot and seeing where the corresponding points fall in other plots. A considerable amount of research has been done in this area, see for example the discussion in Becker and Cleveland [4] and the papers reproduced in Cleveland and McGill[8]. However most of the software developed to date has been developed on specialized hardware, such as the TTY 5620 terminal or Lisp machines. As a result, very few statisticians have had an opportunity to experiment with dynamic graphics first hand, and still fewer have had access to an environment that would allow them to implement dynamic graphics ideas of their own. Several commercial packages for microcomputers now contain some form of dynamic graphics, but most do not allow users to customize their plots or develop functions for producing specialized plots, such as dynamic residual plots. XLISP-STAT provides at least a partial solution to these problems. It allows the user to modify a scatter plot with Lisp functions and provides means for modifying the way in which a plot responds to mouse actions. It is also possible to add functions written in C to the program. On the Macintosh this has to be done by adding to the source code. On some unix systems it is also possible to compile and dynamically load code written in C or FORTRAN.

An integrated environment for statistical calculations and graphics is essential for developing an understanding of the uses of dynamic graphics in statistics and for developing new graphical techniques. Such an environment must essentially be a programming language. Its basic data types must include types that allow groups of numbers – data sets – to be manipulated as entire objects. But in model-based analyses numerical data are only part of the information being used. The remainder is the model itself. Sometimes a model is easily characterized by specifying a set of numbers. A normal linear regression model with *i.i.d.* errors might be described by the number of covariates, the coefficients and the error variance. On the other hand, in many cases it is easier to specify a model by specifying a function. To specify a normal nonlinear regression model, for example, one might specify the mean function. If our language is to allow us to specify this function within the language itself then the language must support a functional data type with full rights: It has to be possible to define functions that manipulate functions, return functions, apply functions to arguments, etc.. The choice I faced was to define a language from scratch or use an existing language. Because of the complexity of issues involved in functional programming I decided to use a dialect of a well understood functional language, Lisp. The syntax of Lisp is somewhat unfamiliar

[1] On a UNIX system you can use the function `load-data` to load the tutorial data. After starting up XLISP-STAT enter the expression `(load-data "tutorial")`, followed by a *return*.

to most users of statistical packages, but it is easy to learn and several good tutorials are available in local book stores. I considered the possibility of using Lisp to write a top level interface with a more "natural" syntax, but I did not see any way of doing this without complicating access to some of the more powerful features of Lisp or running into some of the pitfalls of functional programming. I therefore decided to retain the basic Lisp top level syntax. To make the manipulation of numerical data sets easier I have redefined the arithmetic operators and basic numerical functions to work on lists and arrays of data.

Having decided to use Lisp as the basis for my environment XLISP was a natural choice for several reasons. It has been made available for unrestricted, non-commercial use by its author, David Betz. It is small (for a Lisp system), its source code is available in C, and it is easily extensible. Finally, it includes support for object-oriented programming. Object-oriented programming has received considerable attention in recent years and is particularly natural for use in describing and manipulating graphical objects. It may also be useful for the analysis of statistical data and models. A collection of data and assumptions may be represented as an *object*. The model object can then be examined and modified by sending it *messages*. Many different kinds of models will answer similar questions, thus fitting naturally into an *inheritance structure*. XLISP-STAT's implementation of linear and nonlinear regression models as *objects*, with nonlinear regression inheriting many of its *methods* from linear regression, is a first, primitive attempt to exploit this programming technique in statistical analysis.

Availability

Source code for XLISP-STAT for *X11*, *Sun*, 4.[23]BSD UNIX and Macintosh versions and executables for the Macintosh are available free of charge for non-commercial use. You should, however, be prepared to bear the cost of copying, for example by supplying a disk or tape and a stamped mailing envelope. You can also obtain the source code and Macintosh executables by anonymous *ftp* over the internet from *umnstat.stat.umn.edu* (128.101.51.1).

A version for the Mac II requiring the MC68881 coprocessor is available as well. This version is compiled to access the coprocessor directly and will therefore not run on a machine without the coprocessor. Numerical computations with this version are about 7 to 10 times faster on a Mac II than without the direct coprocessor access.

Disclaimer

XLISP-STAT is an experimental program. It has not been extensively tested. The University of Minnesota, the School of Statistics, the author of the statistical extensions and the author of XLISP take no responsibility for losses or damages resulting directly or indirectly from the use of this program.

XLISP-STAT is an evolving system. Over time new features will be introduced, and existing features that do not work may be changed. Every effort will be made to keep XLISP-STAT consistent with the information in this tutorial, but if this is not possible the help information should give accurate information about the current use of a commend.

1 Starting and Finishing

You should have the program XLISP-STAT on a Macintosh disk. XLISP-STAT needs to have several files available for it to work properly. These files are [2]:

init.lsp
common.lsp
help.lsp
objects.lsp
menus.lsp
statistics.lsp
dialogs.lsp
graphics.lsp
graphics2.lsp
regression.lsp
xlisp.help

Before starting XLISP-STAT you should make sure that these files are in the same folder as the XLISP-STAT application.

To start XLISP-STAT double click on its icon. The program will need a little time to start up and read in the files mentioned above. When XLISP-STAT is ready the text in its command window will look something like this[3]:

```
XLISP version 2.0, Copyright (c) 1988, by David Betz
XLISP-STAT version 2.0 , Copyright (c) 1988, by Luke Tierney.
Several files will be loaded; this may take a few minutes.


; loading "init.lsp"
; loading "common.lsp"
; loading "help.lsp"
; loading "objects.lsp"
; loading "menus.lsp"
; loading "statistics.lsp"
; loading "dialogs.lsp"
; loading "graphics.lsp"
; loading "graphics2.lsp"
; loading "regression.lsp"
>
```

The final "**>**" in the window is the XLISP-STAT prompt. Any characters you type while the command window is active will be added to the line after the final prompt. When you hit a *return*, XLISP-STAT will try to interpret what you have typed and will print a response. For example, if you type a 1 and hit *return* then XLISP-STAT will respond by simply printing a 1 on the following line and then give you a new prompt:

```
> 1
1
>
```

[2] The file `xlisp.help` is optional. It may be replaced by a reduced file `xlisp.help.small` or it may be omitted entirely. If it is not present interactive help will not be available.

[3] On a Macintosh with limited memory a dialog warning about memory restrictions may be appear at this point. On a Mac II it takes about a minute to load these files; on a Mac Plus or an SE it takes about 3.5 minutes.

If you type an *expression* like `(+ 1 2)`, then XLISP-STAT will print the result of evaluating the expression and give you a new prompt:

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

As you have probably guessed, this expression means that the numbers 1 and 2 are to be added together. The next section will give more details on how XLISP-STAT expressions work. In this tutorial I will always show interactions with the program as I have done here: The "`>`" prompt will appear before lines you should type. XLISP-STAT will supply this prompt when it is ready; you should not type it yourself. In later sections I will omit the new prompt following the result in order to save space.

Now that you have seen how to start up XLISP-STAT it is a good idea to make sure you know how to get out. As with many Macintosh programs the easiest way to get out is to choose the **Quit** command from the **File** menu. You can also use the command key shortcut COMMAND-Q, or you can type the expression

```
> (exit)
```

Any one of these methods should cause the program to exit and return you to the Finder.

2 Introduction to Basics

Before we can start to use XLISP-STAT for statistical work we need to learn a little about the kind of data XLISP-STAT uses and about how the XLISP-STAT *listener* and *evaluator* work.

2.1 Data

XLISP-STAT works with two kinds of data: *simple data* and *compound data*. Simple data are numbers

```
1                   ; an integer
-3.14               ; a floating point number
#C(0 1)             ; a complex number (the imaginary unit)
```

logical values

```
T                   ; true
nil                 ; false
```

strings (always enclosed in double quotes)

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

and symbols (used for naming things; see the following section)

```
x
x12
12x
this-is-a-symbol
```

Compound data are lists

```
(this is a list with 7 elements)
(+ 1 2 3)
(sqrt 2)
```

or vectors

```
#(this is a vector with 7 elements)
#(1 2 3)
```

Higher dimensional arrays are another form of compound data; they will be discussed below in Section 9.

All the examples given above can be typed directly into the command window as they are shown here. The next subsection describes what XLISP-STAT will do with these expressions.

2.2 The Listener and the Evaluator

A session with XLISP-STAT basically consists of a conversation between you and the *listener*. The listener is the window into which you type your commands. When it is ready to receive a command it gives you a prompt. At the prompt you can type in an expression. You can use the mouse or the *backspace* key to correct any mistakes you make while typing in your expression. When the expression is complete and you type a *return* the listener passes the expression on to the *evaluator*. The evaluator evaluates the expression and returns the result to the listener for printing. [4] The evaluator is the heart of the system.

[4] It is possible to make a finer distinction. The *reader* takes a string of characters from the listener and converts it into an expression. The *evaluator* evaluates the expression and the *printer* converts the result into another string of characters for the listener to print. For simplicity I will use *evaluator* to describe the combination of these functions.

The basic rule to remember in trying to understand how the evaluator works is that everything is evaluated. Numbers and strings evaluate to themselves:

```
> 1
1
> "Hello"
"Hello"
>
```

Lists are more complicated. Suppose you type the list `(+ 1 2 3)` at the listener. This list has four elements: the symbol `+` followed by the numbers 1, 2 and 3. Here is what happens:

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

A list is evaluated as a function application. The first element is a symbol representing a function, in this case the symbol `+` representing the addition function. The remaining elements are the arguments. Thus the list in the example above is interpreted to mean "Apply the function `+` to the numbers 1, 2 and 3".

Actually, the arguments to a function are always evaluated before the function is applied. In the previous example the arguments are all numbers and thus evaluate to themselves. On the other hand, consider

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

The evaluator has to evaluate the first argument to the function `+` before it can apply the function.

Occasionally you may want to tell the evaluator *not* to evaluate something. For example, suppose we wanted to get the evaluator to simply return the list `(+ 1 2)` back to us, instead of evaluating it. To do this we need to *quote* our list:

```
> (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. There are a few other special forms that we will need; I will introduce them as they are needed. Together with the basic evaluation rules described here these special forms make up the basics of the Lisp language. The special form `quote` is used so often that a shorthand notation has been developed, a single quote before the expression you want to quote:

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

This is equivalent to `(quote (+ 1 2))`. Note that there is no matching quote following the expression.

By the way, the semicolon ";" is the Lisp comment character. Anything you type after a semicolon up to the next time you hit a *return* is ignored by the evaluator.

Exercises

For each of the following expressions try to predict what the evaluator will return. Then type them in, see what happens and try to explain any differences.

1. `(+ 3 5 6)`
2. `(+ (- 1 2) 3)`
3. `'(+ 3 5 6)`
4. `'( + (- 1 2) 3)`
5. `(+ (- (* 2 3) (/ 6 2)) 7)`
6. `'x`

Remember, to quit from XLISP-STAT choose **Quit** from the **File** menu or type `(exit)`.

3 Elementary Statistical Operations

This section introduces some of the basic graphical and numerical statistical operations that are available in XLISP-STAT.

3.1 First Steps

Statistical data usually consists of groups of numbers. Devore and Peck [11, Exercise 2.11] describe an experiment in which 22 consumers reported the number of times they had purchased a product during the previous 48 week period. The results are given as a table:

0	2	5	0	3	1	8	0	3	1	1
9	2	4	0	2	9	3	0	1	9	8

To examine this data in XLISP-STAT we represent it as a list of numbers using the `list` function:

```
> (list 0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8)
(0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8)
>
```

Note that the numbers are separated by white space (spaces, tabs or even returns), not commas.

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 number of purchases for our sample:

```
> (mean (list 0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8))
3.227273
>
```

The median of these numbers can be computed as

```
> (median (list 0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8))
2
>
```

It is of course a nuisance to have to type in the list of 22 numbers every time we want to compute a statistic for the sample. To avoid having to do this I will give this list a name using the `def` special form [5]:

```
> (def purchases (list 0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8))
PURCHASES
>
```

Now the symbol `purchases` has a value associated with it: Its value is our list of 22 numbers. If you give the symbol `purchases` to the evaluator then it will find the value of this symbol and return that value:

```
> purchases
(0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8)
>
```

[5] `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 tutorial. `def` is closely related to the standard Lisp special forms `setf` and `setq`. The advantage of using `def` is that it adds your variable name to a list of `def`'ed variables that you can retrieve using the function `variables`. If you use `setf` or `setq` there is no easy way to find variables you have defined, as opposed to ones that are predefined. `def` always affects top level symbol bindings, not local bindings. It can not be used in function definitions to change local bindings.

We can now easily compute various numerical descriptive statistics for this data set:

```
> (mean purchases)
3.227273
> (median purchases)
2
> (standard-deviation purchases)
3.279544
> (interquartile-range purchases)
3.5
>
```

XLISP-STAT also supports elementwise arithmetic operations on lists of numbers. For example, we can add 1 to each of the purchases:

```
> (+ 1 purchases)
(1 3 6 1 4 2 9 1 4 2 2 10 3 5 1 3 10 4 1 2 10 9)
>
```

and after adding 1 we can compute the natural logarithms of the results:

```
> (log (+ 1 purchases))
(0 1.098612 1.791759 0 1.386294 0.6931472 2.197225 0 1.386294 0.6931472
0.6931472 2.302585 1.098612 1.609438 0 1.098612 2.302585 1.386294 0
0.6931472 2.302585 2.197225)
>
```

Exercises

For each of the following expressions try to predict what the evaluator will return. Then type them in, see what happens and try to explain any differences.

1. `(mean (list 1 2 3))`
2. `(+ (list 1 2 3) 4)`
3. `(* (list 1 2 3) (list 4 5 6))`
4. `(+ (list 1 2 3) (list 4 5))`

3.2 Summary Statistics and Plots

Devore and Peck [11, page 54, Table 10] give precipitation levels recorded during the month of March in the Minneapolis - St. Paul area over a 30 year period. Let's enter these data into XLISP-STAT with the name `precipitation`:

```
> (def precipitation
       (list .77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 2.20 3.30
             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
>
```

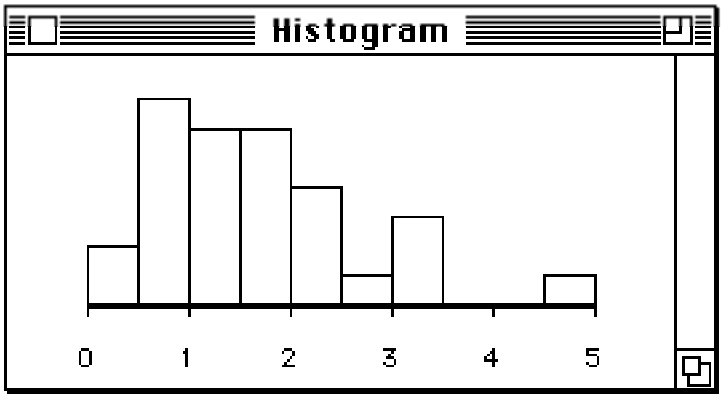

Figure 1: Histogram of precipitation levels.

In typing this expression I have hit the *return* and *tab* keys a few times in order to make the typed expression easier to read. The tab key indents the next line to a reasonable point to make the expression more readable.

The `histogram` and `boxplot` functions can be used to obtain graphical representations of this data set:

```
> (histogram precipitation)
#<Object: 3564170, prototype = HISTOGRAM-PROTO>
> (boxplot precipitation)
#<Object: 3423466, prototype = SCATTERPLOT-PROTO>
>
```

Each of these commands should cause a window with the appropriate graph to appear on your screen. The windows should look something like Figures 1 and 2.

Note that as each graph appears it becomes the active window. To get XLISP-STAT to accept further commands you have to click on the XLISP-STAT listener window. You will have to click on the listener window between the two commands shown here.

The two functions return results that are printed something like this:

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

These result will be used later to identify the window containing the plot. For the moment you can ignore them.

When you have several plot windows open you might want to close the listener window so you can rearrange the plots more easily. You can do this by clicking in the listener window's close box. You can later re-open the listener window by selecting the **Show XLISP-STAT** item on the **Command** menu.

Here are some numerical summaries:

```
> (mean precipitation)
1.685
> (median precipitation)
1.47
> (standard-deviation precipitation)
1.0157
> (interquartile-range precipitation)
1.145
>
```

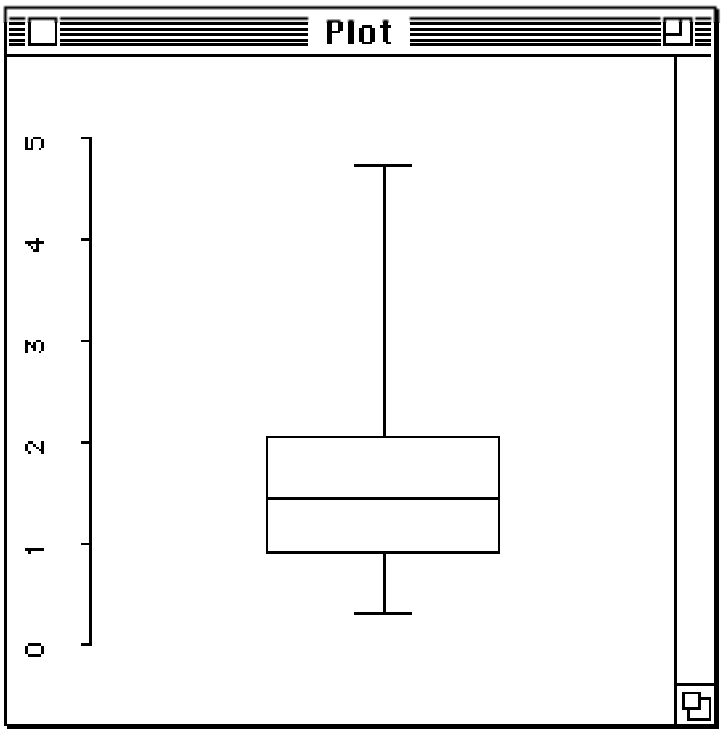

Figure 2: Boxplot of precipitation levels.

The distribution of this data set is somewhat skewed to the right. Notice the separation between the mean and the median. You might want to try a few simple transformations to see if you can symmetrize the data. Square root and log transformations can be computed using the expressions

```
(sqrt precipitation)
```

and

```
(log precipitation).
```

You should look at plots of the data to see if these transformations do indeed lead to a more symmetric shape. The means and medians of the transformed data are

```
> (mean (sqrt precipitation))
1.243006
> (median (sqrt precipitation))
1.212323
> (mean (log precipitation))
0.3405517
> (median (log precipitation))
0.384892
>
```

The boxplot function can also be used to produce parallel boxplots of two or more samples. It will do so if it is given a list of lists as its argument instead of a single list. As an example, let's use this function to compare serum total cholesterol values for samples of rural and urban Guatemalans (Devore and Peck [11, page 19, Example 3]):

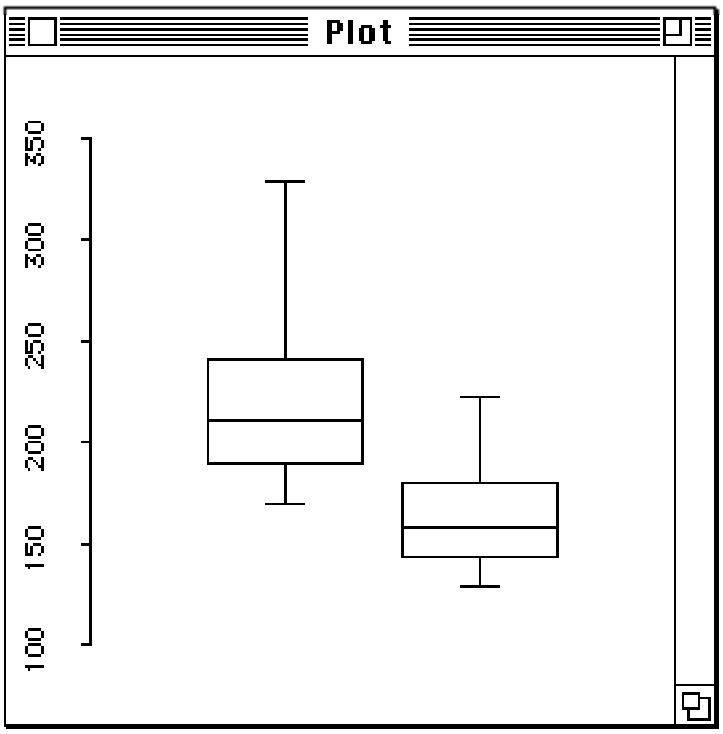

Figure 3: Parallel box plots of cholesterol levels for urban and rural guatemalans.

```
> (def urban (list 184 196 217 284 184 236 189 206 179 170 205 190
                   204 330 217 242 222 242 249 241))
URBAN
> (def rural (list 166 146 144 204 158 143 158 180 223 194 194 175
                   171 155 143 145 131 181 148 144 220 129))
RURAL
>
```

The parallel boxplot is obtained by

```
> (boxplot (list urban rural))
#<Object: 3423466, prototype = SCATTERPLOT-PROTO>
>
```

and is shown in Figure 3; the urban group is on the left.

Exercises

The following exercises involve examples and problems from Devore and Peck [11]. The data sets are in files in the folder **Data** on the XLISP-STAT distribution disk and can be read in using the **Load** item in the **File** menu or using the `load` function (see Section 5.4 below). To use the **Load** item on the **File** menu select this item from the menu. This will bring up an **Open File** dialog window. Use this dialog to open the **Data** folder on the distribution disk. Now select one of the `.lsp` files (`car-prices.lsp` for the first exercise) and press the **Open** button. The file will be loaded and