Recent Developments and Future Directions in Lisp-Stat

By
Luke Tierney[1]
Technical Report No. 608
School of Statistics
University of Minnesota
August 7, 1995

[1]Research supported in part by grant DMS-9303557 from the National Science Foundation.

Abstract

Lisp-Stat is an extensible statistical computing environment based on the Lisp language. The system is currently being revised on the basis of experience gained from several years of use. This paper outlines some of the changes that have been completed and others that are under consideration.

1 Introduction

Lisp-Stat (Tierney, 1990) is an extensible statistical computing environment for data analysis, statistical instruction and research, with an emphasis on providing a framework for exploring the use of dynamic graphical methods. Extensibility is achieved by basing Lisp-Stat on the Lisp language, in particular on a subset of Common Lisp. Lisp-Stat extends standard Lisp arithmetic operations to perform element-wise operations on lists and vectors, and adds a variety of basic statistical and linear algebra functions. A portable window system interface forms the basis of a dynamic graphics system that is designed to work identically in a number of different graphical user interface environments, such as the Macintosh operating system, the *X* window system, and Microsoft Windows. A prototype-based object-oriented programming system is used to implement the graphics system and to allow it to be customized and adapted. The object-oriented programming system is also used as the basis for statistical model representations, such as linear and nonlinear regression models and generalized linear models.

Lisp-Stat was first release in 1989. It has been used for data analysis, as a research tool, and for implementing several larger projects (e. g. Cook and Weisberg, 1994; Young, 1993). Based on experience gained from this use, the system is currently being redesigned. The redesign is evolutionary, with backward compatibility a major objective. The redesign project can be divided into six major segments: the basic Lisp system, data representation and operating system issues, the object system, the graphical system, the statistical component, and the user interface. They will be attacked in this order. The redesign of the basic Lisp system is nearly complete, and some of the changes are outlined in the next section. The third section describes some of the issues involved in the later stages of the revision; this section is more speculative in nature. The final section briefly discusses the importance of extensibility in a statistical software environment.

2 Recent Developments

Lisp-Stat was originally designed as a specification to be implemented on various Lisp systems. The requirement on the Lisp system base is that it support an appropriate subset of the Common Lisp standard (Steele, 1990). The reason for using the Common Lisp specification as a base was that Common Lisp is a rich, high-level language with many features that are already provided and do not need to be designed and documented from scratch. Even though the XLISP language lacked some important Common Lisp features, it was useful as an initial implementation base for Lisp-Stat since it was small and freely available in source form. It was hoped that a transition to a full Common Lisp implementation could be made in the future.

Unfortunately this hope has not been fulfilled. There are a number of reasons, including the continued high cost of commercial Common Lisp implementations, the uncertain future of free and of commercial implementations, and the lack of standardization in window system and foreign function interfaces. Instead, XLISP has been brought closer to a full Common Lisp implementation by adding many Common Lisp functions and some key missing features. The most important added features are multiple values, packages, typed vectors, and a byte

code compiler. Other changes include a new garbage collector and new random number generators. Many of these changes have been folded into the standard XLISP distribution. Other features contributed to the standard distribution by Tom Almy and others have also been or will shortly be incorporated into the XLISP-STAT base. In particular, Tom Almy's unlimited precision integer arithmetic functions will be added in the near future.

2.1 New Common Lisp Features in XLISP

2.1.1 Multiple Values

Multiple values are useful when a function needs to return one primary value and several secondary ones that may be but often are not of interest. Using multiple values avoids the need to make and take apart a list. The hash table lookup function `gethash`, for example, returns the item found as its first value or `NIL` if no item is found. A second value is `t` if an item was found, `NIL` otherwise. This makes it possible to distinguish an item with value `NIL` from an item not found.

Several other high level languages support multiple values. One example is *MATLAB*. For example, the `eig` function in *MATLAB* returns only the eigen values when a single answer is requested; if two values are asked for, it returns the eigenvectors and eigenvalues.

Functions that are only called for their side effects can return no values.

2.1.2 Packages

If a language is to allow the development of substantial subsystems, then it is critical to provide some form of name space management to allow a system to export only its interface and to hide and protect implementation details. Common Lisp manages name spaces by organizing its symbols into collections called *packages*. Each package is divided into internal and external (or exported) symbols. A package can *use* other packages, thus making their symbols accessible within the package. Within a package, only symbols in the package and external symbols of packages used by the package can be referenced directly using their names. As a simple example, if a file contains the code

```
(defpackage "MY-PACKAGE"
  (:use "COMMON-LISP")
  (:export "MY-FUNCTION"))

(defun utility () ...)
(defun my-function () ... (utility) ...)
```

then all symbols in the `"COMMON-LISP"` package and all symbols like `utility` that are in `"MY-PACKAGE"` are accessible in `"MY-PACKAGE"`, but only the exported symbol `my-function` will be available to other packages that use `"MY-PACKAGE"`.

Packages are not modules in the sense of ADA, MODULA-2 or MODULA-3, and they have many shortcomings: They do not allow separate exporting of variables and functions, only symbols; symbols cannot be imported under alternate names; there is no support for organizing separate compilation of system components. But they are a useful first step and

can be used as the basis for more sophisticated module systems. Support for Common Lisp packages is now available in XLISP; a proper module system (e. g. Curtis and Rauen, 1990; Davis *et al.*, 1994) may be added in the future.

2.1.3 Pathnames

The Common Lisp pathname functions allow the portable specification of hierarchical directory structures. For example, the expression

```
(make-pathname :directory '(:relative "a") :name "b")
```

produces `"a/b"` in UNIX, `"a\b"` in MD DOS, and `":a:b"` on the Macintosh. Using these functions it is possible to describe directory structures of a system in a portable way.

2.1.4 Typed Vectors

Vectors and arrays can be restricted to contain only elements of certain specified types. This allows more efficient storage of floating point data and also facilitates the interface to C code by allowing the address of the vector data to be passed directly to a C function. The linear algebra subsystem of Lisp-Stat is being re-implemented to take advantage of this ability. In particular, an interface to Level 1 BLAS and some Level 2 and Level 3 BLAS routines (Anderson *et al.*, 1992) will be provided to allow destructive modification of floating point arrays. The details of the interface are still under development. Once they have been completed, they will allow users to implement efficient linear algebra routines at the Lisp level.

2.2 The Byte Code Compiler

The byte code compiler translates a Lisp function definition into a string of bytes that form an instruction sequence for a *virtual machine* (VM), a fast interpreter for the byte code language. Interpreting byte code is not as fast as executing native machine code, but with a good design the interpreter overhead can be minimized. Byte codes themselves are usually machine-independent, thus making it possible to transfer byte compiled files from one machine to another. In addition, the VM can be implemented in C, thus eliminating hardware dependencies of a native code compiler.

To illustrate what the compiler does, consider the function for adding up a list of numbers shown in Figure 1a. The `dolist` macro is expanded in Figure 1b to show the options for local transfer of control in the loop body (the inner `tagbody`) and for nonlocal exit (the enclosing `block`) that an interpreter has to consider. The compiler recognizes that neither of these is needed and is able to simplify the code down to the set of instructions shown in Figure 2.

Unlike many other byte code VM's, the XLISP VM is not based on a stack model. Instead, the basic instructions are of a three-address-code nature (Aho *et al.*, 1986, Chapter 8). Thus the instruction (`add2` x y z) adds the values stored at offsets x and y and stores the result at offset z from the current frame base. This design seems to produce faster code than a stack-based design for benchmarks that should be representative of statistical

```
(defun f (x)
  (let ((s 0))
    (dolist (y x s)
      (setf s (+ s y)))))
```

(a) before expansion

```
(defun f (x)
  (let ((s 0))
    (block nil
      (let* ((tmp x)
             (y (car tmp)))
        (tagbody
         (unless (consp tmp) (go return))
         loop
         (tagbody (setf s (+ s y)))
         (setq tmp (cdr tmp))
         (setq y (car tmp))
         (when (consp tmp) (go loop))
         return
         (return-from nil s))))))
```

(b) after expansion

Figure 1: A function definition and a (simplified) macro-expanded version

```
(initialize-0 2 1 0 2)
(car 1 3)                          ;(car x) -> y
(test-consp 1 loop return)
loop
(add2 2 3 2)                       ;(+ s y) -> s
(cdr 1 1)                          ;(cdr x) -> x
(car 1 3)                          ;(car x) -> y
(test-consp 1 loop return)
return
(set-one-value-return 0 2)
```

Figure 2: Assembly code for the function in Figure 1

applications. When the example function given here is applied to a list of 1000 integers, the byte compiled code is approximately ten times faster than the interpreted version. Functions in which most iteration is already done in the vectorized code will experience a much smaller improvement.

The use of byte codes has a long history, including, for example, the pcode of the UCSD Pascal system. Recent versions of the Microsoft C compiler have re-introduced the use of byte code as an option to take advantage of the fact that byte code is often more compact than native machine code. Another recent use of byte code is in the *Java* language (Gosling, 1995), where the machine-independence of byte code is used to allow transferring compiled small applications, or applets, for local use by the *HotJava* World Wide Web browser.

The XLISP byte code compiler is based on the design of the ORBIT Scheme compiler (Krantz *et al.*, 1986), which uses conversion to continuation passing style to support a variety of code transformation optimizations (Friedman *et al.*, 1992). The code produced is properly tail recursive; thus iterative computations expressed using recursion will be compiled to iterative code.

Even though the XLISP byte code compiler can already speed up computations considerably, there is still room for improvement. Additional code analysis and support for type declarations will in some cases allow direct use of native machine data types for integers and floating point numbers instead of boxed representations. Optimization strategies designed to improve imperative code, such as static single assignment analysis, which can be related to continuation passing representation (Kelsey, 1995), may also help. It is also possible to replace byte code on a particular machine by threaded code, or to generate C code from the intermediate assembly code and use a local C compiler to produce native code.

The compiler developed up to now is a standard Lisp compiler with only very minimal adaptations to statistical applications. Future work will explore the possibility of incorporating support for vectorized arithmetic and graphical operations at the compiler level in order to optimize performance in statistical applications.

2.3 New Garbage Collection System

The original XLISP memory management system used a mark-and-sweep garbage collector. This collector has the advantage of requiring only two bits of storage per node to implement, but the disadvantage of scanning the entire heap on each collection. With a large heap this can result in pauses long enough to degrade interactive performance. To address this problem, the mark-and-sweep collector was replaced by a simple two-generation generational collector in the spirit of Appel (1989). Generational collectors are based on the assumption that most allocated objects are very short-lived. By distinguishing recently allocated objects from older ones, the collector can usually reclaim adequate space from minor collections in which only the newer nodes are examined. Only rarely is a full collection involving all nodes required. Since the number of active new nodes in the system at any given time is usually very small, the minor collections are very fast and hardly noticeable. Major collections take about as long as mark-and-sweep collections, but occur much less frequently.

Generational collectors are usually implemented as copying collectors, but the resulting data motion would make designing functions that call back to XLISP from C or FORTRAN

quite difficult. A treadmill-type in-place design (Baker, 1992; Wilson, 1992) was therefore used. The nominal space overhead for this approach is considerably larger than for mark-and-sweep: six bytes per node on 32-bit hardware. However on many workstations alignment requirements force enough free space into each node to accommodate this overhead, thus eliminating the space cost on these systems. A compromise that may be worth exploring is to have a first generation that is copied into a fixed second generation. This may provide the advantages of fast allocation achieved by copying collectors without some of the drawbacks that moving data has for call-backs (Doligez and Leroy, 1993).

More work is needed to optimize tuning of the new memory management system to typical statistical activities. The use of adaptive tuning strategies may be explored as well. Support for weak pointers and finalization will also be added.

2.4 New Random Number Generators

The Marsaglia lagged Fibonacci generator used in older versions of XLISP-STAT has been replaced as the default generator by L'Ecuyer's version of the Wichman-Hill generator (L'Ecuyer, 1986; Bratley *et al.*, 1987, Algorithm UNIFL). The original generator is still available, mainly to allow results produced with this generator to be reproduced. Two additional generators are available as well, Marsaglia's Super-Duper generator as used in *S*, and a combined Tauseworthe generator of Tezuka and L'Ecuyer (1991). Random states now contain both generator and seed information. Having several very different generators available is useful for examining the possible sensitivity of simulation results to the generation mechanism. At present the set of available generators is fixed. In the future, a mechanism for adding new generators will be provided.

3 Future Directions

3.1 Additional Data Representations

Until recently data sets of floating point numbers could only be represented in Lisp-Stat as lists or as generic vectors. This requires storing each number in a separate node, and can be quite wasteful. With the addition of typed arrays, it is now possible to use more compact storage. Once typed arrays have been fully integrated, this should increase the size of data sets that can be handled conveniently on standard memory configurations to the level of hundreds of thousands of observations. For larger data sets in the range of several millions of observations, more effective representations will be needed. One possibility is to allow the contents of disk files to be treated as an array. Memory mapped file support may be useful on operating systems where it is available. Since large data sets might only be accessible over a network, remotely stored arrays should be supported as well. To reflect the fact that files may be read-only, it will be necessary to allow arrays to be made read-only as well. It will also be useful to be able to reference smaller subsets of larger arrays indirectly, to support shared sub-arrays.

Once adequate. support for basic handling of larger data sets is available, algorithms for sparse array manipulation will need to be added, and other algorithms will need to be

re-examined to insure that they have adequate numerical properties even for large input arrays. To support adding new algorithms, the current minimal C and FORTRAN interface will need to be improved. Recent developments that have resulted in the inclusion of shared libraries in most operating systems will greatly facilitate this effort.

3.2 Communication and Parallel Processing

The ability to communicate with other applications running locally or remotely is becoming increasingly important. Several new languages have been proposed recently with a structure designed to allow them to take advantage of features of the World Wide Web. Two examples are *Java* (Gosling and McGilton, 1995) and *Obliq* (Cardelli, 1995). Lisp-Stat has already been used as a teaching tool in conjunction with the World Wide Web (Rossini and Rosenberger, 1994). Its use with the Web can be enhanced by adding some of the ideas found in *Java* as well as some lower level communication mechanisms. Security issues that have played a major role in the design of *Java* will also need to be examined to insure that Lisp-Stat can be used safely with the Web.

Adding basic interprocess communication mechanisms such as sockets and X properties for UNIX, Apple events for the Macintosh, and DDE and OLE for MS Windows, will allow Lisp-Stat to take advantage of other applications available in those environments. In addition, in a networked environment these mechanisms can form the basis of a parallel processing environment. The PVM system under UNIX (Geist *et al.*, 1994) is designed around this approach. Either a similar system can be implemented, or an interface to PVM can be provided to allow Lisp-Stat to take advantage of the multiple workstation environments that are now quite common.

Another form of parallelism worth exploring is the use of threads or light-weight processes with shared global memory. Allowing long-running computations to coexist with a graphical user interface is accomplished much more naturally with a threads mechanism than the form of manual implementation that is currently required. In addition, threads allow a system to take advantage of shared memory multiprocessors which are also becoming more common. The SR language (Andrews and Olsson, 1993) provides a useful framework for integrating both separate processes and threads.

One component of Lisp-Stat that is inherently parallel, though the current implementation is serial, is the vectorized arithmetic system. Recent advances in the understanding of nested parallel vector languages (NESL Blelloch, 1994; Proteus Goldberg *et al.*, 1994) may be useful in redesigning this system to be more expressive by making it easier to define vectorized functions at the user level, and more efficient by allowing parallel architecture to be exploited when it is available. One possibility is to re-implement the Lisp-Stat vectorized arithmetic system using the CVL library (Blelloch *et al.*, 1994), which provides implementations for workstations, the Connection Machines CM2 and CM5, the Cray Y-MP and the MasPar MP2.

3.3 The Object System

The Lisp-Stat object system is both unusual and conventional. It is unusual in being based on prototypes rather than classes, and it is conventional in using only single dispatching for handling methods. The use of prototypes instead of classes seems to have been successful, and a number of recent object-oriented languages with a similar emphasis on interactive use have taken this route as well. Many Lisp-based object systems, such as CLOS (Steele, 1990), the EuLisp object system (Padget *et al.*, 1994), and Dylan (Apple Computer, 1994) use multiple dispatching. Other languages that use multiple dispatching are Cecil (Chambers, 1993) and *S*. Multiple dispatching has more expressive power than single dispatching, but also represents a more complex programming paradigm. Most work on object-oriented design (e. g. Rumbaugh *et al.*, 1991) is based on the single dispatch model. Only recently have researchers begun to formulate a framework for understanding multiple dispatch (Chambers, 1992). If these efforts are successful, then it may be worth reconsidering the use of multiple dispatching. For now, single dispatching appears adequate and better understood.

There are situations where it would be useful to develop specialized object-oriented subsystems to support a particular project. This might be to provide increased efficiency or increased expressive power. Such a system can be built from scratch, but would be easier to construct if it could leverage off of the existing system. The need for customized object systems has lead to the development of meta-object protocols (Kiczales *et al.*, 1991; Padget *et al.*, 1994). It may prove useful to design a meta-object protocol for Lisp-Stat as well and to implement the current protocol as a special case.

An area of considerable current research and commercial interest is the development of standards for linking objects in separate applications and on remote systems. Some of the projects with this objective are OpenDoc, OLE, SOM, CORBA, and ILU. Most approaches seem to be working towards compliance with CORBA. ILU (Janssen *et al.*, 1995), which provides a CORBA interface, or Fresco (Linton and Price, 1993), which is based on CORBA, may provide an effective means for integrating object linking into Lisp-Stat. Providing a standard linking mechanism will allow Lisp-Stat to more easily communicate with other programs, either using them as compute engines or serving them as a compute engine. It will also allow Lisp-Stat sessions on separate workstations to communicate with one another in a transparent fashion.

The current Lisp-Stat object system does not provide a standardized broadcasting mechanism for efficiently distributing change notifications to interested objects. At present such broadcasts have to be implemented by hand (Tierney, 1993). An efficient, standard mechanism is needed to adequately support the Model-View-Controller paradigm that has become central to graphical user interface design. A mechanism similar to the one used in Smalltalk will need to be incorporated.

3.4 The Graphics System

The current Lisp-Stat graphics system was designed as a compromise between flexibility, simplicity, and efficiency. The goal of redesigning the graphics system is to increase the flexibility of the system while maintaining or improving on simplicity and efficiency. For example, the original design identifies plots with their containing windows. This simplifies

the user model for dealing with plots, but prevents placing multiple plots in the same window. Similarly, dialog items were considered part of special dialog windows, thus preventing the integration of standard dialog items with plot windows.

The new design will support a hierarchical window structure in which each top level window contains a nested hierarchy of widgets. Each widget can be an elementary item such as a button or a slider, or another collection of widgets. Geometry managers will be provided to facilitate display-independent layout management. The design of the *Tk* toolkit (Ousterhout, 1994) may provide a useful model to explore. With increases in workstation speed experienced in recent years it may also be possible to represent plots as collections of widgets. This will again increase flexibility, but may need to be deferred if it is still too costly in performance on current hardware.

In addition to supporting standard widgets and widgets defined in Lisp-Stat, the graphics system should also support externally defined widgets, such as OLE controls. The ability to embed widgets related to other processes running locally or remotely also needs to be explored. The Fresco toolkit (Linton and Price, 1993), which is based on the CORBA standard for distributed objects, may provide a useful model or a possible basis for this development.

Within the statistical graphs themselves, it would be helpful to provide more programmability to layout features, such as the axes on a plot. It would also be useful to provide primitives for managing symbols or other glyphs that represent groups of points rather than just individual points. This would provide a useful superstructure for histograms as well as for binned scatterplots (Carr, 1991)

Finally, it would be useful to allow closer adaptation to native GUI standards, but without sacrificing code portability. This is difficult to achieve, but is facilitated somewhat by the convergence of features in different GUI's, such as the Macintosh, MS Windows, and Motif, that has occurred over the last few years.

3.5 Models and Data

The current statistical model system has proven quite effective for code re-use, but it does have some design features that are now generally considered to be unfortunate form the point of view of object-oriented design (Rumbaugh *et al.*, 1991). In particular, it would be better to design a nonlinear regression model to have a linear regression model component for code re-use (a *has-a* relationship) and delegate appropriate messages to this component, instead of having nonlinear regression models inherit from the linear regression model (an *is-a* relationship). Using inheritance means that even inappropriate methods are inherited; using containment and delegation provides more reasonable control. To assist with this change, the object system should be modified to provide direct support to delegating messages received by one object to another object, usually a slot value of the original receiver.

It will also be necessary to design a useful data set prototype that is capable of storing attribute information, such as whether the values of data are to be interpreted as numerical values or factor levels. The lack of such a system has forced several users to develop variants of their own.

3.6 Syntax and User Interface Issues

Lisp syntax is often perceived as a bit of an impediment to the use of the language. There is considerable debate about the degree to which this impediment is real or perceived. The success of Lisp-Stat to date suggests it may be less of an issue that is sometimes claimed. Nevertheless, alternate syntaxes, at least for parts of the system, are worth exploring. For example, a simple infix parser may be useful for specifying mathematical formulas. It may also be useful to develop a simple vector subscripting language similar to the ones used in *S* or *MATLAB*.

Developing a textual syntax that is more natural, in some sense, than Lisp's parenthesized prefix syntax while remaining as powerful is a difficult task. Even though discussions of syntax pros and cons often focus on a comparison of infix and prefix notation, probably a more significant aspect of Lisp syntax that can make it hard to follow at times is that there are no syntactic cues to help distinguish special forms, or syntactic keywords, from standard functions – the programmer has to know which symbols refer to special forms. This is a weakness of Lisp syntax, but at the same time it is also a great strength: there are no syntactic impediments to the introduction of new special forms. This makes Lisp a programmable language that can be used to define new, problem-specific languages (Graham, 1994). Achieving this level of flexibility with an infix syntax is extremely difficult; this is reflected by the long delay introduced into the Dylan project by their decision to adopt an infix syntax (Apple Computer, 1994)

A promising alternative to a textual syntax is a visual one. Research on visual languages has met with some successes (Cox *et al.*, 1989; Khoros Rasure *et al.*, 1990; Burnett *et al.*, 1994) and has also seen some application in statistical computing (Oldford and Peters, 1988). Another interesting and related area of research is programming by example, or programming by demonstration (Cypher, 1993). It is still too early to tell whether there are visual paradigms that are sufficiently universal to be intuitive and easy to use, while at the same time retaining the expressive power of their textual counterparts. But even if these approaches cannot entirely replace a textual syntax, it may be possible to develop very useful and effective visual interfaces to significant portions of a statistical system. This could greatly enhance the ease of use for those portions, but it comes at a price: Unless all features are accessible using a visual interface, a barrier is established between those portions that are and those that are not. Learning to use simpler aspects of the system does not provide any assistance at reaching beyond this barrier. The result could be to discourage, rather than encourage, experimentation and development; this would be unfortunate.

4 Discussion

The major objective of Lisp-Stat is to provide a flexible system that can easily be extended both in its numerical and its graphical capabilities. The current system represents a first step; the revisions currently in progress are designed to bring it closer towards this goal. Extensibility is critical for a system to be able to adapt to new statistical problems and ideas. Having an extensible system allows research to progress more rapidly, since new ideas are easier to test and refine. But it also gives a data analyst more flexibility to adapt methods

to a problem instead of having to adapt problems to available methods. In short, having an extensible computing environment helps to reduce the gap between statistical research and practice, which is to the benefit of both.

The Heidelberg Workshop where this paper was presented provided a nice opportunity to illustrate the advantages of extensibility. In an evening session Andreas Buja presented a new idea for interactively controlling a tour of four-dimensional space (Buja and ???, 1996). The idea was clearly excellent, but it was hard to appreciate fully without being able to try it out. It was promised that the idea would be incorporated in a future release of XGobi, but it was not clear when that might be available. Fortunately, by taking advantage of the extensible nature of Lisp-Stat, I was able to put together a simple implementation in an hour or two that evening, and could then begin to experiment with it the next day. The implementation was pedestrian to be sure, but adequate as a prototype. Having the prototype to experiment with helped to underscore the quality of the basic idea.

References

Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. (1992). *LAPACK Users' Guide.* SIAM, Philadelphia, PA.

Andrews, Gregory R. and Olsson, Ronald A. (1993). *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings.

Appel, Andrew W. (1989). Simple generationsl garbage collection and fast allocation. *Software Practice and Experience* **19** (2) 171–183.

Apple Computer (1994). *Dylan (TM) Interim Reference Manual.* Apple Computer, Inc.

Baker, Jr., Henry G. (1992). The treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices* **27** (3) 66–70.

Blelloch, Guy E., Chatterjee, Siddhartha, Hardwick, Jonathan C., Reid-Miller, Margaret, Sipelstein, Jay, and Zagha, Marco (1994). CVL: A C vector library (2.0). Technical report, School of Computer Science, Carnegie Mellon University.

Blelloch, Guy E. (1994). NESL: A nested data-parallel language (3.0). Technical report, School of Computer Science, Carnegie Mellon University.

Bratley, P., Fox, B. L., and Schrage, L. E. (1987). *A Guide to Simulation.* Springer, New York, NY, second edition.

Buja, Andreas and ??? (1996). ???? *Journal of Computational and Graphical Statistics* ??
.

Burnett, Margaret M., Goldberg, Adele, and Lewis, Ted G. (1994). *Visual Object-Oriented Programming.* Prentice Hall, Englewood Cliffs, NJ.

Cardelli, Luca (1995). A language with distributed scope. In *Proc. of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pp. 286–297. ACM SIGPLAN, ACM Press.

Carr, Daniel B. (1991). Looking at large data sets using binned data plots. In *Computing and Graphics in Statistics*, pp. 7–39.

Chambers, Craig (1992). Object-oriented multi-methods in Cecil. In *ECOOP'92 Conference Proceedings*, Utrecht, The Netherlands.

—— (1993). The Cecil language: Specification and rationale. Technical report, Department of Computer Science, Univeristy of Washington.

Cook, R. D. and Weisberg, S. (1994). *An Introduction to Regression Graphics.* Wiley, New York, NY.

Cox, P. T., Giles, F. R., and Pietrzykowski, T. (1989). Prograph: A step towards liberating programming from textual conditioning. In *IEEE Workshop on Visual Languages*, pp. 150–156.

Curtis, Pavel and Rauen, James (1990). A module system for Scheme. In *ACM Conference on Lisp and Functional Programming*, pp. 13–19. ACM SIGPLAN, ACM Press.

Cypher, Allen (1993). *Programming by Demonstration.* MIT Press, Cambridge, MA.

Davis, Harley, Parquier, Pierre, and Séniak, Nitsan (1994). Talking about modules and delivery. In *ACM Conference on Lisp and Functional Programming*, pp. 113–120. ACM SIGPLAN, ACM Press.

Doligez, Damien and Leroy, Xavier (1993). A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. 20th Symp. Principles of Programming Languages.* ACM SIGPLAN, ACM Press.

Friedman, D. P, Wand, M., and Haynes, C. T. (1992). *Essentials of Programming Languages.* MIT Press, Cambridge, MA.

Geist, Al, Beguelin, Adam, Dongarra, Jack, Jiang, Weicheng, Manchek, Robert, and Sunderam, Vaidy (1994). *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA.

Goldberg, Allen, Prins, Jan, Reif, John, Faith, Rik, Li, Zhiyong, Mills, Peter, Nyland, Lars, Palmer, Dan, Riely, James, and Westfold, Stephen (1994). The Proteus system for the development of parallel applications. Technical report, Department of Computer Science, University of North Carolina.

Gosling, James and McGilton, Henry (1995). The Java language environment: A white paper. Technical report, Sun Microsystems, Inc.

Gosling, James (1995). Java intermediate bytecodes. In *SIGPLAN Workshop on Intermediate Representation*, pp. 111–118. ACM SIGPLAN, ACM Press.

Graham, Paul (1994). *On Lisp: Advanced Techniques for Common Lisp.* Prentice Hall, Englewood Cliffs, NJ.

Janssen, Bill, Severson, Denis, and Spreitzer, Mike (1995). ILU 1.8 reference manual. Technical report, Xerox Corporation.

Kelsey, Richard A. (1995). A correspondence between continuation passing style and static single assignment form. In *SIGPLAN Workshop on Intermediate Representation*, pp. 13–22. ACM SIGPLAN, ACM Press.

Kiczales, Gregor, des Rivières, Jim, and Bobrow, Daniel G. (1991). *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA.

Krantz, D. A., Kelsey, R., Rees, J. A., Hudak, P., Philbin, J., and Adams, N. I. (1986). Orbit: An optimizing compiler for Scheme. In *Proc. Symp. on Compiler Construction*, volume 21 of *ACM SIGPLAN Notices*, pp. 219–223.

L'Ecuyer, P. (1986). Efficient and portable combined random number generators. *Communications of the ACM* **31** 742–749.

Linton, M. and Price, C. (1993). Building distributed user interfaces with Fresco. In *Proceedings of the Seventh X Technical Conference*, pp. 77–87.

Oldford, R. W. and Peters, S. C. (1988). DINDE: Towards more sophisticated software environments for statistics. *SIAM Journal on Scientific and Statistical Computing* **9** 191–211.

Ousterhout, John K. (1994). *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA.

Padget, Julian, Nuyens, Greg, and Bretthauer, Harry (1994). An overview of EuLisp. *Lisp and Symbolic Computation* **6** (1/2) 9–98.

Rasure, Williams, Argiro, and Sauer (1990). A visual language and software development environment for image processing. *International Journal of Imaging Systems and Technology* **2** 183–199.

Rossini, A. J. and Rosenberger, J. L. (1994). Teaching statistics and computing via multimedia and the world wide web. *Statistical Computing and Graphics Newsletter* **5** (3) 1, 10–13.

Rumbaugh, James, Blaha, Michael, Premerlani, William, Eddy, Frederick, and Lorensen, William (1991). *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ.

Steele, Guy L. (1990). *Common Lisp the Language.* Digital Press, 2nd edition.

Tezuka, S. and L'Ecuyer, P. (1991). Efficient and portable combined Tauseworthe random number generators. **1** 99–112.

Tierney, Luke (1990). *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics.* Wiley, NewYork, NY.

Tierney, Luke (1993). Announcements in Lisp-Stat. *Bulletin of the International Statistical Institute* **IV** 111–124.

Wilson, Paul R. (1992). Uniprocessor garbage collection. In Bekkers, Yves and Cohen, Jaques, eds., *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science. Springer-Verlag.

Young, Forrest W. (1993). ViSta – the visual statistics system. a research and development test bed for statistical visualization techniques. Technical report, University of North Carolina Psychometrics Lab.