SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(1), 33–48 (JANUARY 1997)

Slisp: A Flexible Software Toolkit for Hybrid, Embedded and Distributed Applications

JAMES F. BRINKLEY AND JEFFREY S. PROTHERO

Structural Informatics Group, Digital Anatomist Program, Department of Biological Structure, Box 357420, University of Washington, Seattle, WA 98195, U.S.A. (email: brinkley@u.washington.edu, jsp@biostr.washington.edu)

SUMMARY

We describe Slisp (pronounced 'Ess-Lisp'), a hybrid Lisp–C programming toolkit for the development of scriptable and distributed applications. Computationally expensive operations implemented as separate C-coded modules are selectively compiled into a small Xlisp interpreter, then called as Lisp functions in a Lisp-coded program. The resulting hybrid program may run in several modes: as a stand-alone executable, embedded in a different C program, as a networked server accessed from another Slisp client, or as a networked server accessed from a C-coded client. Five years of experience with Slisp, as well experience with other scripting languages such as Tcl and Perl, are summarized. These experiences suggest that Slisp will be most useful for mid-sized applications in which the kinds of scripting and embeddability features provided by Tcl and Perl can be extended in an efficient manner to larger applications, while maintaining a well-defined standard (Common Lisp) for these extensions. In addition, the generality of Lisp makes Lisp a good candidate for an application-level communication language in distributed environments.

KEY WORDS: exploratory programming; hybrid programming; Lisp; C; Xlisp; World Wide Web; scripting languages

INTRODUCTION

A constant characteristic of software is change. Even in the earliest days of batch computing, programs continually evolved and grew in response to new requirements. In the current networked and interactive window-based computing environment, evolutionary change, both in new software development and in modifications to existing programs, occurs even faster than before. Thus, there is a trend to replace, or at least to augment, the traditional edit–compile–link cycle with rapid development environments and to provide hooks that let end-users or end-programmers customize and extend existing software tools without having to wait for the next release.

For example, in artificial intelligence research most of the problems are ill-defined[1] and solutions are found by a process of quickly trying out new ideas. Therefore, the language of choice for AI is often Lisp,[2] an extensible language offering sophisticated data structures, automatic storage management, and support for both interpreted and compiled code.

However, the flexibility of rapid prototyping environments usually comes at the price of slow execution and/or large programs. When successful research developments are transferred to a production environment this cost becomes prohibitive, requiring that the program be completely re-written in a language such as C.[3] Because of the high cost of re-coding, much

CCC 0038–0644/97/010033–16 $17·50
©1997 by John Wiley & Sons, Ltd.
Received 20 February 1995
Revised 9 May 1996

application-oriented AI is done in C from the outset, not only to avoid the high cost of porting, but also to allow integration with existing applications. However, C lacks the quick turnaround and automatic storage management of Lisp, while the long compile cycles for C defeat the original advantage of a rapid prototyping programming environment.

In an effort to obtain the best of both worlds, there is a trend for lower-level languages such as C to support more high-level facilities (for example, the C++ extensions to C), and for high-level languages such as Lisp to provide more access to lower-level programming facilities, such as the foreign function support offered by most contemporary Common Lisp systems.[2] However, both of these approaches have important drawbacks: C++ continues to lack automatic storage management, for example,[4] while Lisp systems still fail to deliver small executables, and integration of new low-level functionality into a full-scale Lisp environment remains a tricky task, exacerbated by frequent inaccessibility of the source code.

As a result, an increasing number of systems rely on a hybrid programming approach, in which production and efficiency-critical code is written in C, while rapid prototyping is done in a small, interpreted scripting language. This approach originated at the opposite end of the programming spectrum from the large development environments of AI research labs, where the need arose to write small quick-and-dirty programs for simple tasks that would either be too much effort to code in C, or would be too complex to code in existing systems such as the Unix shell.

One of the most popular examples of the hybrid programming genre is Perl,[5] which started out as a simple report generation language. In addition to providing common programming constructs, Perl supports extension of the interpreter through both statically linked and dynamically loaded libraries, allowing Perl to be interfaced to a variety of systems, ranging from database engines to graphical user interface toolkits, and extended to handle tasks as far removed from simple textual report generation as immersive 3D graphics. The latest version of Perl can also be embedded in existing C programs.

A similar example is Tcl/Tk.[6] Tcl is an interpreted, string-based scripting language that can be extended with new C functions, and embedded in other C programs. This latter capability was used in the development of Tk,[7] a set of Tcl commands that provide access to the X-windows toolkit, thereby allowing both rapid prototyping of graphical user interfaces, and dynamic customization of existing interface widgets by rebinding them to modified Tcl functions.

Both Perl and Tcl/Tk, plus to a lesser extent, a similar scripting package called Python,[8] have become very popular for rapid prototyping of small interactive applications. Perl especially has been widely used for many of the small cgi- bin scripts that have proliferated on the World Wide Web.[9] These languages have become so popular, in fact, that they have been extended and used to build much larger applications than they were originally designed for.

However, when these languages begin to be used for purposes other than they were originally intended, their weaknesses as general programming languages become apparent.[10] For example, both Tcl and Perl lack all simple types other than strings, and all compound types other than hashtables, plus modern amenities such as garbage collection. Because of the lack of generality in the original language design, there is no well-defined standard for the inevitable extensions that make the languages more widely useable, with the result that both Tcl and Perl have the potential to fragment into incompatible development threads.

An alternative to this potential fragmentation is the use of a well-defined general purpose language in the first place, initially implementing only a small subset of the language for the required scripting and communication tasks. Then, when the inevitable extensions are needed, they can be implemented according to the general language standard. In this manner it may be possible to more closely approximate the goal of developing a broad spectrum language that can be used for both small and large programs.

This goal suggests that it may be worth re-evaluating Lisp, and in particular ANSI standard Common Lisp, as a scripting language that has the potential to function across the spectrum of application sizes. Even though full Common Lisp systems are large, subsets of Common Lisp can be small enough to perform many of the functions that Tcl and Perl were designed to do.

One of the main drawbacks of Lisp, and probably the largest reason for its declining popularity, is that it is initially difficult for C or Fortran programmers to understand. However, once this hurdle is surmounted the many advantages of Lisp become apparent, and in fact, dialects of Lisp have been used as scripting languages, including Elisp with Emacs for text-editing,[11,12] Autolisp with Autocad for CAD/CAM,[13] and GCL with GeomView for scientific visualization.[14]

A Lisp interpreter often used for this purpose is Xlisp, a small, portable, Common Lisp subset that is coded in C and runs on many different platforms ranging from PCs to Unix workstations.[15] Because of its relatively small size and easy accessibility, Xlisp has been extended to form the basis for several systems, including Xlisp-Stat for statistical calculations,[16] Winterp for interfacing with Motif Widgets in the X Windows system[17] (with a purpose similar to Tcl/Tk but not currently receiving as much attention as Tcl), and Xlisp-Plus with additional Common Lisp functionality.[18] In most of these systems, additional C-coded Lisp functions are added to the basic Lisp functionality of Xlisp, thereby allowing C-coded primitives to be mixed with Lisp-coded interpreted functions in a hybrid system.

In this paper we describe our experience with Slisp (Skandha Lisp, after our graphics visualization program Skandha),[19] an Xlisp-based programming toolkit that we developed for use in our research. Slisp attempts to generalize the hybrid approach taken by the other Xlisp applications, thereby making it relatively easy for application programmers to extend Xlisp with C-coded functionality according to the Common Lisp standard. The Slisp toolkit allows for the creation of independent C-coded modules implementing primitive Lisp functions. These C-coded modules can be mixed and matched with interpreted Lisp to create applications ranging from small scripts to large, computationally-intensive applications in which Lisp acts as a glue to tie together large C modules. As with Tcl or Perl, the resulting programs may be run in several different modes: (1) from the standard Xlisp command-line prompt; (2) as embedded Lisp within a different C program; (3) as a network server; and (4) as a network client to another Slisp server.

In the remainder of this paper, we describe the architecture of Slisp as well as example applications that we have developed in our lab. We then discuss our experiences using Slisp in comparison with other languages, and in the final section, we present our conclusions as to when Slisp is an appropriate language to use.

SLISP ARCHITECTURE AND MODES OF OPERATION

Figure 1 shows the basic architecture of Slisp. An Slisp *compiled C program* (*P*1. . . *P*3 in Figure 1) includes routines from *xcore*, the Lisp interpreter plus basic Lisp functionality developed by David Betz, plus some number of modules (*x*1 . . .*x*4) containing the C-coded functions specializing it to the task at hand. An Slisp *hybrid program* (*H*1 in Figure 1) consists of an Slisp compiled program *P*3, augmented by Lisp code defined by the `defun` special form, and `load`ed at runtime.

Figure 2 shows more details of the hybrid program *H*1 in Figure 1. The compiled C program

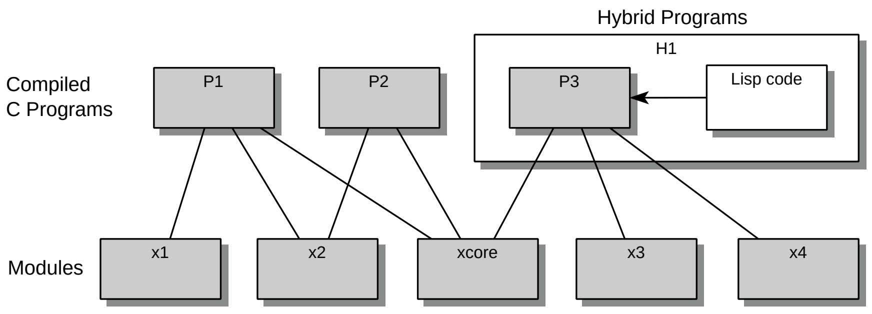

Figure 1. Slisp architecture. Shaded boxes are C code, non-shaded are Lisp code

*P*3 includes the standard Xlisp main program, which implements the Lisp `read-eval-print` loop. The `eval` function calls routines in *xcore* to execute the standard Lisp primitives such as `cons`, `list`, etc. It also calls user-defined routines in modules *x*3 and *x*4 to execute additional Lisp primitives defined in these modules.

As in any Lisp system, the Lisp code itself may be divided into Lisp modules such as *L*1 and *L*2, which may or may not correspond to the C modules *x*3 and *x*4. Thus, an Slisp program is usually a hybrid of C-coded Lisp routines designed for fast execution, and Lisp-coded Lisp routines that glue together the C routines for added functionality.

Any Slisp program may be run in any of four basic modes, shown in Figure 3, which may in turn be combined to create more complex modes: from the standard Lisp command line (Figure 3a), embedded within an arbitrary C program (Figure 3b), from a remote Lisp command line (Figure 3c), or from a different C-based remote client (Figure 3d).

Figure 3(a) shows the basic command-line mode, in which Lisp expressions typed by the user are processed by the standard Lisp `read-eval-print` loop. The program *P*3 is invoked by the name *P*3 rather than Xlisp to reflect the fact that it includes the additional compiled modules *x*3 and *x*4.

Figure 3(b) shows that the same Slisp modules may be embedded within an arbitrary C program *P*4, anything from a tiny stub to a huge application like AVS,[20] by linking it with the Slisp library function `sl_Eval_Str`. This function accepts a string containing a valid Lisp expression, `read`s the string to create an internal Lisp structure, then `eval`uates it in the same way as the command line version. The resulting internal list structure is converted to a string and returned to the calling program. It is also possible to pass more complex data structures between C and Lisp by directly calling the Xlisp data manipulation functions. As in the Tk extensions to Tcl, we find the embeddability feature particularly useful when adding Lisp functionality to a C-based graphical user interface development environment.

As shown in Figures 3(c) and 3(d) any Slisp program *P*3 may be run as a simple server by executing it from the command line with a *-s* switch. This switch suppresses prompting and delimits error messages and return values by configurable control-characters, thereby simplifying client processing of server output. Since the server uses the standard Lisp `read` function, it processes textual input in the same way as the command line version, ignoring newlines and comments, and waiting until a complete s-expression is read before evaluating it. Thus, if the client sends an incomplete s-expression, the server will wait until the client sends the rest of the expression.

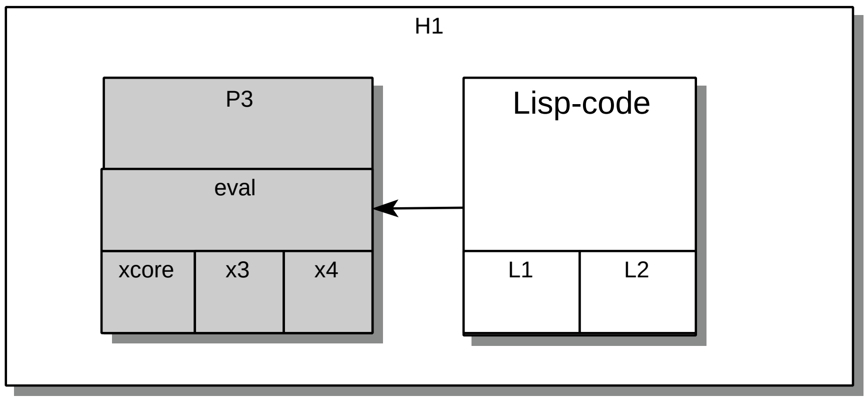

Figure 2. Structure of an slisp program. Shaded boxes are C code, non-shaded are Lisp code

A server set up in this way may then be invoked by a process on the same or another host, for example, via the Internet meta-daemon *inetd* upon connection to a specified port.

If an Slisp program, *P*5 in Figure 3(c), includes module *xnet*, which implements the outgoing telnet-protocol connections, it may act as a client to another Slisp program running in server mode. *xnet* implements a Lisp function `net-eval` which simplifies Lisp-level remote procedure calls by transparently converting the given Lisp expression to text form (taking advantage of the standard Lisp internal/external representation isomorphism), sending it to the (possibly remote) server, and converting the text result back to internal form using the standard Lisp reader. Since any Slisp process that includes *xnet* may play both client and server roles, considerable flexibility is afforded in the construction of distributed programs.

The *xnet* module accesses a library of C functions that establish telnet-protocol connections using Berkeley sockets. The primary function in this library is `sn_Eval_Str`, which has the same syntax as `sl_Eval_Str` in the embedded function library (Figure 3b). An arbitrary remote C client, *P*6 in Figure 3(d), can call `sn_Eval_Str` directly to pass a string to a remote Slisp server for evaluation. Because the embedded and remote client evaluation functions have the same syntax but slightly different names, a simple global replace, followed by linking with a different library, suffices to convert from an embedded program (Figure 3b) to a distributed program (Figure 3d). The different names also allow both functions to be used in the same program.

EXAMPLE SLISP MODULES AND APPLICATIONS

Slisp has been evolving over the past five years within our Digital Anatomist Program,[21,22] the long term goal of which is to develop methods for representing and managing structural biology information within a distributed client-server framework.

Table I shows the Slisp modules that we have implemented to-date. Most modules consist of both C- and Lisp-coded functions. The C-coded functions are included in one of three basic Slisp compiled applications, *Sl*, *Siserver* and *Skandha4*, which are then augmented by interpreted Lisp to produce complete task-specific applications.

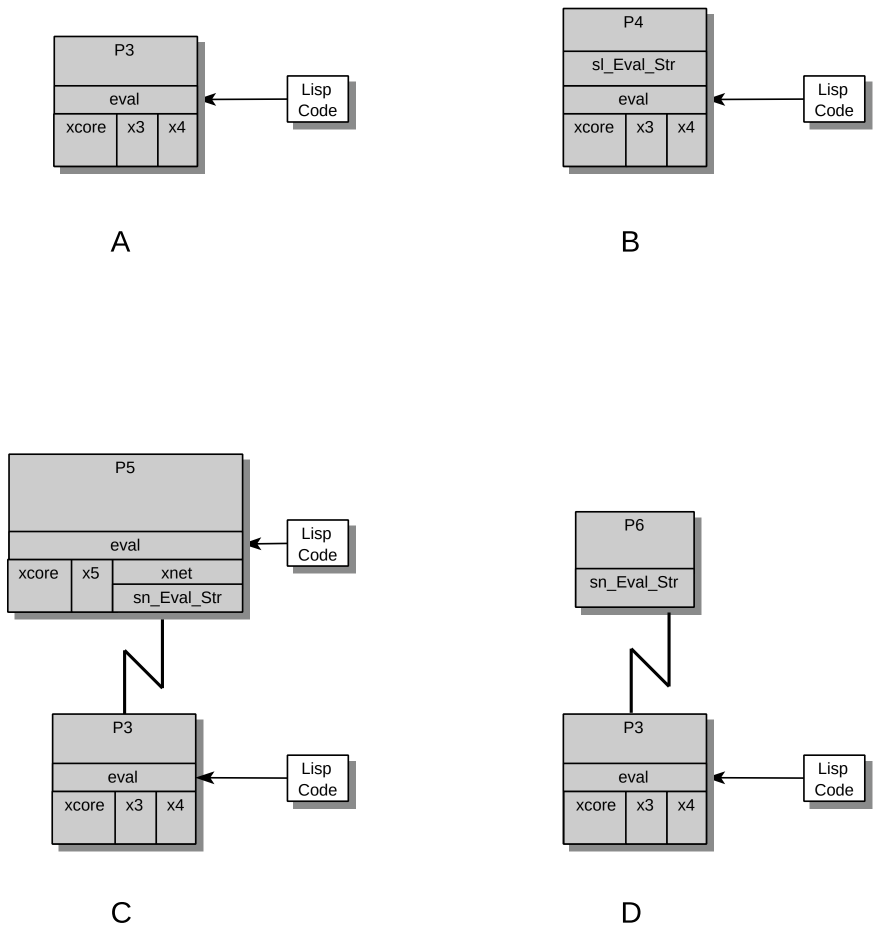

*Figure 3. Slisp modes. Shaded boxes are C code, non-shaded are Lisp code. (a) Command-line, in which the user types Lisp expressions that are evaluated by the Lisp interpreter, (b) embedded within an arbitrary C program P*4*, (c) as a network server P*3 *for an Slisp client P*5*, and (d) as a network server P*3 *for an arbitrary C client P*6

The bottom rows of Table I show the total number of lines of C and Lisp code for each application as the sum of the lines for each included module (as measured by *wc -l*). The number of lines of C is broken into the total lines of C, and the lines of C exclusive of *xcore*, since *xcore* contains the original Xlisp functionality written by David Betz.

Table II shows the size of the executables in kilobytes (by running the *ls -l* command) for the three compiled Slisp applications, compared with executables for three other rapid prototyping languages that we have installed in our lab.

Table I. Slisp modules

Modules[a]		Lines of Code		Compiled Apps		
		C	Lisp	Sl	SiS[b]	Sk4[b]
xcore	Core Xlisp functions	14,047		x	x	x
xetc	Miscellaneous	450		x	x	x
xnet	Network access	267		x		
xsybase	Sybase access	544	758		x	
xhsy	Hershey fonts	722				x
xg	C-based Xlisp objects	1608				x
xg.3d	Basic 3-D graphical objects	37,327	80			x
xg.3d.fileio	File input/output	6370				x
xg.3d.gui	3-D graphical user interface	8976	1173			x
xg.3d.image	2-D image processing	5158	167			x
xg.3d.model	3-D models	562				x
xpvd	Video disk driver	355				x
xbtp	Bitpad driver	3915				x
xavs	Communication with AVS	637				x
xgplotlib	Machine specific graphics	18,817				x
xsk3	3-D surface display	17,780	7884			x
xmorpho	Image segmentation[c]	2853				x
xscanner	Image segmentation[d]	6606	11,668			x
xmri	MR volume visualization	6286	5690			x
xbib	Biliographic retrieval		2274		x	
xkb	Semantic network access		379		x	
xeps	Music sampler database		602		x	
Total lines of Lisp				0	4013	29,515
C excluding xcore				717	994	115,836
C including xcore				14,764	15,041	129,883

[a] Top group are common modules, second is xsybase module for Siserver, third are modules for Skandha4, last group are task-specific modules.

[b] SiS = Siserver, Sk4 = Skandha4.

[c] Manual.

[d] Semi-automatic, model-based.

Sl

Sl is a minimal Slisp that includes the core Xlisp functionality in the *xcore* module, a few miscellaneous Xlisp primitives in *xetc* and the client capabilities in *xnet*. *Sl* has been compiled and tested under NeXTStep, Silicon Graphics Irix, and HP U/X. It should run on most Unix platforms because Xlisp has been widely used for years, and in fact is included as part of the standard X-Window distribution.

Sl by itself contains no Lisp code, and almost all of the C code is that contained in the original Xlisp distribution. The executable size of 456K is not much larger than the minimal Tcl distribution, smaller than Perl version 5, and much smaller than Common Lisp.

Table II. Executable file sizes

Program	Executable (K)
Sl	456
Siserver	664
Skandha4	3603
Tcl 7.5 (tclsh)	328
Perl 5	923
Allegro Commonlisp	4606

Siserver

Siserver includes the *xsybase* module in addition to *xcore* and *xetc*, thereby providing access to the commercially-available Sybase relational database. Since *xsybase* requires the Sybase client library, which is currently only available to us on NeXT, *siserver* only runs on NeXT. The xsybase module includes 544 lines of C to implement the primitive Sybase access functions, and 758 lines of Lisp to implement a Lisp-level object-oriented layer over the database.

The following modules, shown in the lower rows of Table I, are dynamically loaded into the compiled *Siserver* executable in order to customize *Siserver* for specific tasks:

(a) The *xbib* module provides additional Lisp code to implement a personal bibliographic management system that is stored in Sybase. The database may be accessed via Lisp-level command line functions, by a NeXTStep graphical interface, or by a Web-based interface.

(b) The *xkb* Lisp module calls *xsybase* C-coded functions to access a semantic network of anatomic terminology and symbolic relationships (`part-of`, `is-a`, etc). Lisp functions, such as (`kb-get-children`) and (`kb-get-ancestors`), traverse the semantic network and perform simple inferences. *Siserver*, augmented by the `xkb` lisp functions, is provided as a server process under *inetd*, and is accessed by a Macintosh program called the Digital Anatomist Interactive Atlas, an on-line atlas of anatomy.[22] More recently, we have begun accessing the *xkb* functions via a Web-based version of the interactive atlas.[23]

(c) The *xeps* Lisp module, as an example of an application done just for fun, provides a different set of Lisp functions that call *xsybase* to access a sound sample database for a music synthesizer called the Ensoniq EPS. As in *xbib*, this module is embedded in a NeXTStep front end via the `sl_Eval_Str` library function. At some point, a separate module *xmidi* could be included to control the synthesizer from the Lisp level, and to link that in turn to interfaces such as NeXTStep, X-Windows or Windows, or to substitute a different database by re-writing the functions in the *xsybase* module.

The bottom of Table I shows that the total number of lines of C added by the custom modules in *Siserver* is 994, whereas the number of lines of dynamically-loadable Lisp is 4013. Thus, for *Siserver* most of the additional code is written in Lisp, with only the minimal C needed to access the database. The expressive power of Lisp permits complex object-oriented programs to be developed without the need for *ad hoc* extensions to the language, yet, like Tcl or Perl, the small size and accessibility of the Xlisp functions permits the application to be embedded in a graphical user interface.

Skandha4

The third compiled application, *Skandha4,* is an example of large system development that would be difficult if not impossible to do in a small scripting language such as Tcl or Perl. As shown in Tables I and II, the *Skandha4* executable size of 3.6 M approaches the executable size of Allegro Commonlisp (4.6 M), and the number of added lines of C (115,836) is much greater than the number of added lines of Lisp (29,515). Thus, for *Skandha4,* Lisp is used more as a scripting language than as the major development language.

The purpose of *Skandha4* is to provide a customizable toolkit of efficient functions for modelling, display and animation of biological structures. The structures are often represented by several hundred thousand polygons, which must be rendered efficiently to achieve reasonable display rates. *Skandha4* meets this efficiency requirement with a set of fundamental C-based objects and rendering methods, defined in the *xg* and *xg.3d* modules shown in Table I, that can store and process large amounts of floating point and integer data. These objects and methods are integrated into standard Xlisp data structures, and are thereby treated as first class Lisp objects that are automatically managed by the Xlisp garbage collector.

The C-based objects are dynamically combined at the Lisp level to create structural models that can be manipulated to generate 3D animated displays. The fundamental objects can also be used as the basis for other graphics or image processing tasks. Given these C-based primitives, a relatively small amount of dynamically-loaded Lisp code can customize *Skandha4* to meet a variety of application-specific needs.

The lower rows of Table I show, in addition to the *Siserver* modules previously described, the Slisp modules that were designed to customize *Skandha4* in various ways:

(a) *Xsk3* implements Lisp-based hierarchical surface modelling and display functions on top of the basic *Skandha4* datatypes, and is the basis for the graphics production environment used by the anatomists in our group. The module is called *xsk3* because it emulates the look and feel of our earlier program *Skandha3,* which was written entirely in C.[19]

(b) *Xmorpho* retrieves and displays 2D images of biological sections, and supports hand-tracing of anatomical structures from on-screen images in order to produce 3D reconstructions.

(c) *Xscanner*[24] is a re-implementation of an earlier program called *Scanner*, developed in Objective C on the NeXT computer,[25] that uses anatomic shape knowledge to semi-automatically segment (find) structures in medical images.

(d) *Xmri* adds 3D voxel manipulation and visualization to Skandha4. In this case foundation objects defined in *xg* and *xg.3d* are augmented to process and render 3D volume data as obtained from magnetic resonance or other medical imaging modalities. *Xmri* is one component of a larger project for human brain mapping,[26] and as such it employs the *xavs* module to communicate with the commercial visualization package AVS for pixel and voxel processing operations that we did not want to re-invent. Skandha4 has also been embedded into a C-based AVS module by means of the `sl_Eval_Str` function.

Skandha4 has been compiled for Silicon Graphics machines and for the IBM RS6000. Low level graphics primitives are implemented in the SGI GL language, and are confined to the *xgplotlib* module. Therefore, porting to machines based on other graphics standards should be straightforward.

SLISP EXPERIENCE

Slisp has been in development in our lab since 1990, before Tcl and Perl attained their current popularity. Given the widespread use of these languages, the availability of large Common Lisp systems and the C++ language, and the impending stabilization of Java[27] for client-side computing on the World Wide Web, it is important to re-assess whether we or anyone else should continue development in Slisp, or whether we should move to one of these other languages. Our current conclusion is that we will continue to develop in Slisp where appropriate, but will also continue to develop in other languages. This conclusion is based on our experience with Slisp over the past five years, as well as our experience with other languages such as C, Perl and Tcl.

Historical rationale for Slisp

Slisp evolved from our collective experience with the slow development cycles of C, particularly with previous versions of the Skandha program for 3D reconstruction and display of anatomic objects.[19] Experience with the earlier programs convinced us that requests for new functionality were the norm rather than the exception, at least in a research environment. Since the compiled programs were fairly large, addition of the new functionality often required long compile cycles to develop and debug, and as a result, many requested features were never implemented.

We therefore decided to incorporate a scripting language in the next version of Skandha. However, because of the excessive storage and processing requirements of image and graphics data, we decided that much of the code would remain in C, and that the scripting language would serve primarily as a glue to tie together the C-based functions. We decided not to use Common Lisp because previous experience with Lisp suggested that it is very difficult to exert precise low-level control over code and data structures, and to efficiently interface them to C-coded graphics libraries.[28]

The choice of a rapid prototyping language was dictated by the desire to use a complete language that could be extended in a well-defined manner. Although Perl and Tcl were available, they were not as popular as they are now, so the choice was not influenced by the large number of user-contributed programs that are currently available for these languages.

Xlisp was chosen as the basic language because of its small size, completeness, extensibility in a standard manner, garbage collector, and the availability of the source code. Garbage collection by itself was and is one of the main reasons for choosing Lisp over languages like Perl or Tcl. Xlisp itself, however, was not easily modifiable to include new C functions, which is why we developed the Slisp toolkit.

Development environment

Our software development group currently includes nine people who program at least some of the time, five of whom have programmed at least some in Slisp, and three of whom (including the authors of this paper) have programmed extensively in Slisp. Other languages in use in our group include C, C++, Perl, Allegro Common Lisp, and the visual programming language of AVS. A few of the programmers in the group have extensive experience with Lisp, but most have a C background.

The diverse backgrounds and opinions of our developers have led to discussions about the relative merits of various languages, including Slisp. Rather than impose a single development language, we allow each person to choose the language they think is most suited to the task, but we require the individual applications to communicate via a standard application programmer's interface (API), which we have chosen to be Lisp because of the desire to use a complete language for communication as well as scripting.

This diverse development environment has allowed us to compare different approaches along several dimensions that are presented in the next sections.

Software development times

Most of the earlier development in our group was done in C. For example, a NeXTStep image database system accessed Sybase directly through the Sybase client library,[29] and required over six man-months to build, as compared to about two man-months for the Slisp-based bibliographic retrieval system, even though the functionality is similar. Similarly, once the basic Skandha4 C functions were available, the Skandha3 emulator was programmed in much less time than it took to program the C version, as was the *xscanner* image segmentation module.

In general we have found that, after an initial learning curve, Slisp Lisp-level programming is several times faster than C programming for mid-size programs, where a relatively small number of C primitives can be combined in many ways at the Lisp level. If a large amount of C must be written, then it is faster to program in straight C, since there is overhead involved in integrating C with the Xlisp data structures.

For graphics and imaging in particular we have found that relatively few fundamental C-based data structures and methods are enough to provide a powerful set of tools for rapid prototyping of efficient graphics applications. Although both Perl and Tcl can also be extended, the use of strings as the fundamental data types makes it very difficult to generate these kinds of efficient graphics primitives.

Small versus large systems

We use C, Perl and Slisp for the rapid development of small programs. We increasingly use Perl for Web-accessible executable scripts, and for system administration tasks. We also use Slisp for many small programs, particularly hybrid programs which call *Siserver* functions to access Sybase. We are just beginning to look at Tcl/Tk and WAFE[30] for rapid prototyping of X interfaces.

We use C, Common Lisp and Slisp for the development of large programs. Allegro Common Lisp and C are used to modify programs that are already written in those languages, whereas Slisp is used to modify and extend the *Skandha4* program for graphics and imaging. We generally no longer use straight C for large systems because of the long compile cycles, and because we can achieve comparable results in Slisp.

The *Siserver* program is near the small end of the size spectrum, and *Skandha4* is nearer the large end. Thus, we have found that Slisp can be used at both ends of the spectrum, even though there are alternative but different languages available at each end.

Efficiency

An important advantage of a hybrid language like Slisp is that new ideas may be rapidly prototyped in the interpreted language, then speed-critical functions may be re-coded in C for efficiency. In our experience with this kind of environment, we have generally been able to determine ahead of time what will work in Lisp and what needs to be coded in C. Thus, we generally have not recoded much in C, even though we could expect some speedup if we did so.

One example where re-coding made a dramatic difference in efficiency is the *xscanner* module. Test runs identified two hotspots in this module: one, a function sampling a 2D image along an arbitrary line segment; and the other, evaluating the gradient on the resulting 1D data. Recoding these two Lisp functions in C dropped evaluation times on an 80 contour dataset from 116 minutes to 25 minutes, a 4.6-fold speedup.

Client-server operations

We currently use Slisp in several client-server configurations. *Siserver* is run on one Tcp port with the *xbib* Lisp code, and on another port with the *xkb* Lisp code. In both cases the program is started anew by *inetd* each time a new connection is made on the specified port, which means that all the Lisp functions must be loaded. For *xbib* this load time is significant enough that the response time is too slow for use as a back end for a Web-based bibliographic management system, since a new connection is made for each request. For *xkb*, which loads only a few Lisp functions, the response time is fast enough that we currently use it for browsing anatomic terminology and definitions in our Web-based anatomy atlas.[23]

The speed limitations could be greatly reduced by modifying the main Xlisp routine so that an Slisp program could run as a server independently of *inetd*. In this case the Lisp code would be loaded by the parent process, after which child processes would be forked in response to client connections. Another approach we have tried is to run *Skandha4* in server mode via a persistent named pipe on the same machine as a Web browser, and to control *Skandha4* from a Web interface. However, this approach requires that the server be on the same machine as the client.

The server mechanism currently does not include more advanced features such as multi-threading for efficiency, and distinctions between actual communication (i.e., data) and meta-data (e.g. authentication). These limitations mean that the servers are primarily useful for communication among a small number of secure applications, or for access from a Web server, which is the envisioned mode of operation in our group. However, more advanced network features could be added if further experience suggests the need.

Re-usability

One of the advantages of a machine-independent scripting language such as Lisp, Tcl or Perl is that it promotes re-usability by providing an intermediate layer between low level machine-specific code, and higher-level user interface code. In the case of Slisp, if a significant amount of code is written at the intermediate Lisp level and a smaller amount is written at both the higher and lower levels, then previously-written code can be re-used. This re-usability is one of the main attractions of Tcl and Perl, since user-contributed extensions can be incorporated into new programs.

Slisp also promotes the development of re-usable code, as illustrated by the bibliographic management program embodied in the *xbib* module. At the low level, the foundation of *xbib* is a small set of C-coded Lisp functions that access a relational database. These functions are built upon by Lisp-level code that represents the relational database as a set of objects. Because of this layered approach it would not be difficult to replace the low-level C-functions with a different module that accesses a different relational database, or even an object-oriented database, while preserving the same higher-level methods.

At the high level the *xbib* objects and methods present an interface to the programmer or user that is more application-specific than low level SQL statements, and is therefore more stable as the database is changed. At the same time, the embedding and remote execution facilities of Slisp allow this API to be accessed from multiple types of user interface: the standard Lisp command line (Figure 3a), embedded within a C program (Figure 3b), from a remote Lisp command line (Figure 3c), or from a different C-based remote client (Figure 3d).

Thus, one way to use the Slisp toolkit is to develop an intermediate Lisp layer that provides at the high end a relatively domain-specific API, defined in a complete language (Lisp), and which accesses a set of interchangeable C-coded modules at the low end. If the application is designed correctly, both the low-level modules and the interfaces which call the API can be interchanged without requiring major re-writes of the Lisp level code.

Ease of learning

The biggest barrier to using Slisp in our group is the intimidation that C programmers seem to experience when encountering Lisp code for the first time. Once this barrier is overcome, Slisp Lisp-level programming is much like Common Lisp programming, except there are fewer functions to learn.

Slisp programming at the C level is more difficult than straight C programming for several reasons: (1) it is necessary to understand a fair number of the Xlisp C routines in order to effectively integrate new functions into Xlisp; (2) it is necessary to protect pointers so they are not unexpectedly destroyed by the Xlisp garbage collector; and (3) the Xlisp C functions are only documented in the source code. However, simple functions can be written without too much difficulty once a few examples are available.

In practice, we have found that a useful division of labor is for one knowledgeable programmer to develop C-level Lisp code, while the other programmers program in Lisp to quickly try out new ideas. When C functions are needed they are often coded by the single programmer in response to requests from the Lisp level programmers. This division of labor seems to have occurred in the Tcl and Perl communities as well, since only a few people are developing the language and adding C-based extensions, whereas many more people are using the languages at the interpreted level.

Integration with other languages

Slisp is currently only used within our own group. Therefore, one of the main reasons we are using Perl and possibly Tcl is the availability of a large amount of user-contributed code. There is also a large amount of code available in C and Common Lisp. Thus, even though Lisp might be a better language for many tasks, it is usually not worth re-implementing extensions that have already been developed elsewhere, especially when modern hardware and networks permit heterogeneous applications to communicate.

For programs written in C, or a language such as Perl or Tcl that can be embedded in C, Slisp provides a convenient wrapper, since these languages can be embedded in Slisp. In cases where another language must be at the top level (e.g. Common Lisp or Tk), Slisp can be embedded in these languages via the C library functions. Slisp can also be run as a client or server, communicating via Lisp with applications that only need to parse Lisp commands. Thus, Slisp should work well in heterogeneous environments, and could provide a common API for communication among diverse applications.

Slisp Limitations

As with any other environment Slisp has limitations. Slisp Lisp-level code cannot be compiled, so speedups can only be achieved by re-coding in C or by porting to Common Lisp. However, the ability to port to Common Lisp compiled code is an important advantage that is not available with *ad hoc* scripting languages.

C-level code also cannot yet be dynamically loaded in Slisp. The version of Xlisp on which Slisp is based (version 2.1) is only a subset of Common Lisp, so many of the standard Common Lisp functions are not available, although they may be coded in C or DEFUN'd in Lisp when necessary. The current version of Slisp is not compatible with later versions of Xlisp developed by Tom Almy,[18] Luke Tierney[16] and Neils Mayer,[17] although these versions could be made compatible since they all stem from the same root, Xlisp. A more serious limitation is that the Xlisp object system is not compatible with the Common Lisp Object System (CLOS), so CLOS code cannot be ported directly to Slisp. On the other hand the Xlisp object system is very straightforward to use, and appears to be semantically equivalent to Objective C.

All these limitations could be overcome if it seemed worthwhile. Since we are currently using Slisp only within our own research group, and since Slisp is adequate for our purposes, we have not needed to expend the additional effort towards improving the program. However, if a sufficiently large user community were to develop then it would be worth addressing some of these issues. A major advantage of Lisp as a language is that there are well-defined paths for these extensions and optimizations, paths that are not defined at all for *ad hoc* languages.

CONCLUSIONS

Our experience with Slisp as well as other languages has shown that, if only because of personal preference, no one software or hardware architecture is adequate for all tasks. In the light of this fact, and assuming that the learning barrier has been overcome, the following are some conclusions we have made about the utility of different languages for various task sizes:

(a) Small tasks can be done in whatever language is most convenient, whether that be C, Perl, Tcl, Lisp or Java. Perl is particularly convenient for Unix system administration, Tcl/Tk should be useful for quick X-based programs, and Java will most likely become the language of choice for Web client-side computing.[27]
(b) Large experimental AI-like programs are best done in one of the Common Lisps because of the generality of the language and because of the large amount of source code contributed by the Lisp community.
(c) Existing code should be used as is whenever possible, as long as it can be made to communicate with other programs. Slisp can be useful as a wrapper in these cases.
(d) For new applications, Slisp seems to fill a niche somewhere between very small tasks and very large tasks, that is for medium size programs in which the same customizability and embeddability that is available for small scripting languages can be extended to larger programs.

Our evolving lab policy takes these considerations into account, respecting strong 'religious' attachments of individuals to particular languages when present, and otherwise encouraging use of each language in its area of strength, with (whenever practical) Lisp as the *lingua franca* connecting the components.

We believe that this policy is a practical one given that contemporary software development, both at our lab and in the world at large, is becoming steadily less a matter of stand-alone monolithic application solutions than of heterogeneous software resources, developed on a variety of platforms, in a variety of languages, interacting via the network to provide end-user solutions. In this context, a standard API is essential if we are to have smoothly meshing tools. We believe Lisp, based on its unexcelled history of stability, extensibility, generality and smooth evolution over time, is uniquely suited to this critical role, and that Slisp is an effective and efficient framework for the construction of such cooperative software components.

ACKNOWLEDGEMENTS

This work was funded by National Library of Medicine Grant LM04925, National Library of Medicine Contract LM13506, National Cancer Institute Grant CA59070, and Human Brain Project Grant DC/LM02310, co-funded by the National Library of Medicine and the National Institute on Deafness and Other Communication Disorders. We would like to thank the following individuals for their help in various aspects of this work: Scott Bradley for the Web interface to the *bib* program, Kraig Eno for the *xkb* module; Kevin Hinshaw for extensive comments on the manuscript and for the *xscanner* module with its speedup experiments; anatomists David Conley, John Sundsten and Peter Ratiu for actually using and providing valuable feedback about the Slisp tools we produce; and Cornelius Rosse, Greg Heil, Steve Tanimoto, Bharath Modayur and Rex Jakobovits for discussions and feedback regarding this manuscript. We would also like to thank David Betz for developing the Xlisp program and for making it widely available, and the reviewers of the original version of this manuscript, who provided very helpful suggestions for its improvement.

APPENDIX: AVAILABILITY

The basic Slisp framework is freely available for both commercial and non-commercial use, with no restrictions other than those described in the general copyright notice associated with the Slisp distribution. However, this general copyright specifically does not apply to individual modules developed within the Slisp framework, which may be copyrighted in a manner determined by the authors. We hope that this approach will result in many freely available modules (perhaps protected by the GNU Public Library license), but other modules may be protected for commercial use, and distributed only as executable libraries. How well this will work is not yet clear, and we have not yet implemented a mechanism for distributing some modules only as libraries, though it should be relatively straightforward to do so.

The current version of the Slisp framework is available from ftp://ftp.biostr.washington.edu in the directory pub/slisp/slX.tar.Z, where X is the current version number. This file contains the Slisp framework, the basic Slisp program Sl, and documentation that should make it possible to build additional Slisp programs.

REFERENCES

1. R. Fikes, ‘Ai and software engineering-managing exploratory programming’, *AAAI-90 Proceedings. Eighth National Conference on Artificial Intelligence,* Cambridge, MA., July 29-August 3 1990, pp. 1126–1127. MIT Press.
2. D. K. Layer and C. Richardson, ‘Lisp systems in the 1990s’, *Communications of the ACM,* **34**(9), 48–57 (1991).

3. J. Eccles, ‘Porting from commonlisp with flavors to c++’, *USENIX Proceedings. C++ Conference*, Denver, CO., October 17–21 1988, pp. 31–40. USENIX Assoc., Berkeley, CA.
4. M. Sakkinen, ‘The darker side of c++ revisited’, *Structured Programming*, **13**(4), 155–177 (1992).
5. L. Wall and R. L. Schwartz, *Programming Perl*, O’Reilly and Associates, Sebastopol, CA., 1992.
6. J. K. Ousterhout, ‘Tcl: an embeddable command language’, *Proceedings of the Winter 1990 USENIX Conference*, Washington, DC, January 22–26 1990, pp. 133–146. USENIX.
7. J. K. Ousterhout, ‘An x11 toolkit based on the tcl language’, *Proceedings of the Winter 1991 USENIX Conference*, Dallas, TX, January 21–25 1991, pp. 105–115. USENIX.
8. A. R. Watters, ‘The what, why, who, and where of python’. UnixWorld Online: Tutorial Article No. 005, http://www.wcmh.com/uworld/archives/95/tutorial/005.html, September 1995.
9. R. J. Vetter, C. Spell and C. Ward, ‘Mosaic and the world-wide web’, *Computer*, **27**(10), 49–57 (1994).
10. R. Stallman, ‘Why you should not use tcl’. http://minsky.med.virginia.edu/sdm7g/LangCrit/Tcl/RMS-Why-you-should-not-use-Tcl, September 1994.
11. R. M. Stallman, *GNU Emacs Manual*, Free Software Foundation, 675 Massachusetts Ave., Cambridge, MA 02139, 1987.
12. H. Halme and J. Heinanen, ‘Gnu emacs as a dynamically extensible programming environment’, *Software – Practice and Experience*, **18**(10), 999–1009 (1988).
13. M. B. McGrath, ‘Autocad release 11 and autoshade’, *Proceedings National Computer Graphics Association 1991*, Chicago, IL, April 22–25 1991, pp. 70–74.
14. University of Minnesota Geometry Center, ‘Geomview’. Available by anonymous ftp from geom.umn.edu, 1994.
15. D. Betz, ‘Xlisp: an object-oriented lisp’. Unpublished reference manual for version 2.1, available from ftp.biostr.washington.edu and other Xlisp ftp sites, 1989.
16. L. Tierney, *LISP-Stat: an object-oriented environment for statistical computing and dynamic graphics*, Wiley, New York, 1990.
17. N. P. Mayer, ‘The winterp widget interpreter: an application prototyping and extension environment for osf/motif’, *Motif ’91, First Annual Motif Users Meeting*, 1991. Available via World Wide Web from http://www.eit.com/software/winterp.html.
18. T. Almy, ‘Xlisp-plus’. Available from ftp.biostr.washington.edu and other Xlisp ftp sites, 1994.
19. J. S. Prothero and J. W. Prothero, ‘A software package in c for interactive 3d reconstruction and display of anatomical objects from serial section data’, *NCGA Proceedings*, 1989, pp. 187–192.
20. Advanced Visual Systems, *AVS User’s Guide*, 300 Fifth Ave.,Waltham, MA 02154, 1994.
21. J. F. Brinkley, J. S. Prothero, J. W. Prothero and C. Rosse, ‘A framework for the design of knowledge-based systems in structural biology’, *Proceedings 15th Annual Symposium on Computer Applications in Medical Care*, Baltimore, MD, 1989, pp. 61–65.
22. J. F. Brinkley, K. Eno, and J. W. Sundsten, ‘Knowledge-based client-server approach to structural information retrieval: the digital anatomist browser’, *Computer Methods and Programs in Biomedicine*, **40**, 131–145 (1993).
23. S. W. Bradley, C. Rosse, and J. F. Brinkley, ‘Web-based access to an online atlas of anatomy: the digital anatomist common gateway interface’, *19th Symposium on Computer Applications in Medical Care*, New Orleans, October 30–November 1 1995, pp. 512–516.
24. K. P. Hinshaw, R. B. Altman, and J. F. Brinkley, ‘Shape-based models for interactive segmentation of medical images’, *SPIE Medical Imaging 1995: Image Processing*, San Diego, February 26-March 2 1995, pp. 771–780.
25. J. F. Brinkley, ‘A flexible, generic model for anatomic shape: application to interactive two-dimensional medical image segmentation and matching’, *Computers and Biomedical Research*, **26**, 121–142 (1993).
26. J. F. Brinkley, L. M. Myers, J. S. Prothero, G. H. Heil, K. R. Maravilla, G. A. Ojemann and C. Rosse, ‘A structural information framework for brain mapping’, in S. H. Koslow and M. F. Huerta (eds), *Progress in Neuroinformatics*. Lawrence Erlbaum. In press.
27. B. Carlson, ‘A jolt of java could shake up the computing community’, *IEEE Computer*, **28**(11), 81–82 (1995).
28. J. F. Brinkley, R. B. Altman, B. S. Duncan, B. G. Buchanan and O. Jardetzky, ‘Heuristic refinement method for the derivation of protein solution structures: validation on cytochrome b562’, *J. Chem. Inf. Comput. Sci.*, **28**(4), 194–210 (1988).
29. J. F. Brinkley, ‘A distributed, object-oriented framework for medical image management and analysis: application to evaluation of medical image segmentation techniques’, *Proceedings IEEE Workshop on Biomedical Image Analysis*, Seattle, June 24–25 1994, pp. 194–203. IEEE Press.
30. G. Neumann and S. Nusser, ‘Wafe – an x toolit based frontend for applications in various programming languages’, *Proceedings USENIX Winter Conference*, January 1993.