Appendix C
Lisp Environment Specification

KSL Lisp Environment Requirements

*** *DRAFT* ***

Richard Acuff

Knowledge Systems Laboratory

Computer Science Department, Stanford University

May 11, 1988

1. Introduction

The Knowledge Systems Laboratory (KSL) is an artificial intelligence research laboratory at Stanford University that has been developing systems and tools in Lisp environments for more than 20 years. These systems have been implemented in the InterLisp, MACLisp, ZetaLisp, and more recently, CommonLisp dialects predominantly. Beginning in the early 1980's, our work moved from mainframe Lisp environments to workstation environments for many reasons, principally involving powerful tools for system development and debugging and graphical interfaces. Commercial versions of these tools, that evolved over many years in the Xerox D-Machine, Symbolics 36xx, LMI, and TI Explorer systems, have become an indispensible part of our work environment. Newer Lisp systems for workstations not specifically developed for Lisp have lacked many important features of these environments. This document attempts to summarize the key features of the Lisp machine environments that would be needed in "stock" machine implementations in order to make them attractive in a development setting.

There are several overall points to be emphasized about this write-up:

1) These requirements represent a snapshot of the tools and technology available on today's machines. AI has historically and will continue to ride the crest of the wave of new computing technologies for the forseeable future, which enable ever more complex systems. Thus, these are not static requirements and we expect to be able to take advantage of the future improvements in hardware, graphics, and software as they are generated by computer science research and industry.

2) It has been hard to describe concisely many aspects of the Lisp environments because they involve visual interactions and the "feel" of the way systems are organized and interconnected. The write-up assumes a general familiarity and experience with the Lisp environments Xerox, Symbolics, TI Explorer systems.

3) We have tried to sort out the key features of current systems that are important to our research work. Except where explicitly stated, everything in this document describes this "core" of functionality. Some items are clearly more important than others, but all represent needs that really guide our decisions about which new systems can be broadly used in the KSL.

4) The discussion is organized according to a "layered" view of Lisp environments shown in Figure 1, beginning with the upper levels. This organization is a conceptual framework within which to describe the various parts of the environment but may not correspond in full detail to the way system modules are actually organized. While most of the discussion focuses on the higher layers in this diagram, unavoidably some issues involving lower level or more general issues such as address space, system speed, and graphics facilities have to be mentioned.

5) While this description is based on what we know, use, and understand today, we have attempted to allow for innovation by describing the functionality that we require in a fairly abstract way wherever possible, rather than specifying, for example, the "Symbolics XYZ feature". This may result in

some ambiguities that will need to be addressed by appropriate discussion and iteration.

6) We have two overriding goals in adopting future computing environments (which may seem to be or actually are in some conflict). We want the most powerful development environments we can get to facilitate the building of complex AI programs. But at the same time, we want to be able to share (import and export) research results and tools with colleagues in other labs and so must maximize the portability of code among systems. We believe that these goals can be approached jointly by the establishment and careful adherence to standards where possible, while continuing systems development where necessary.

7) Because of the evolving nature of these research environments, no vendor's system can ever be "finished". While we expect reasonably professional standards of robustness and reliability in the systems we use, we also expect to have special needs and to work closely with the vendors of products we use to adapt, extend, and debug the environment and tools. Our experience has been that in order to do this effectively, it is essential that we have broad access to system source codes.

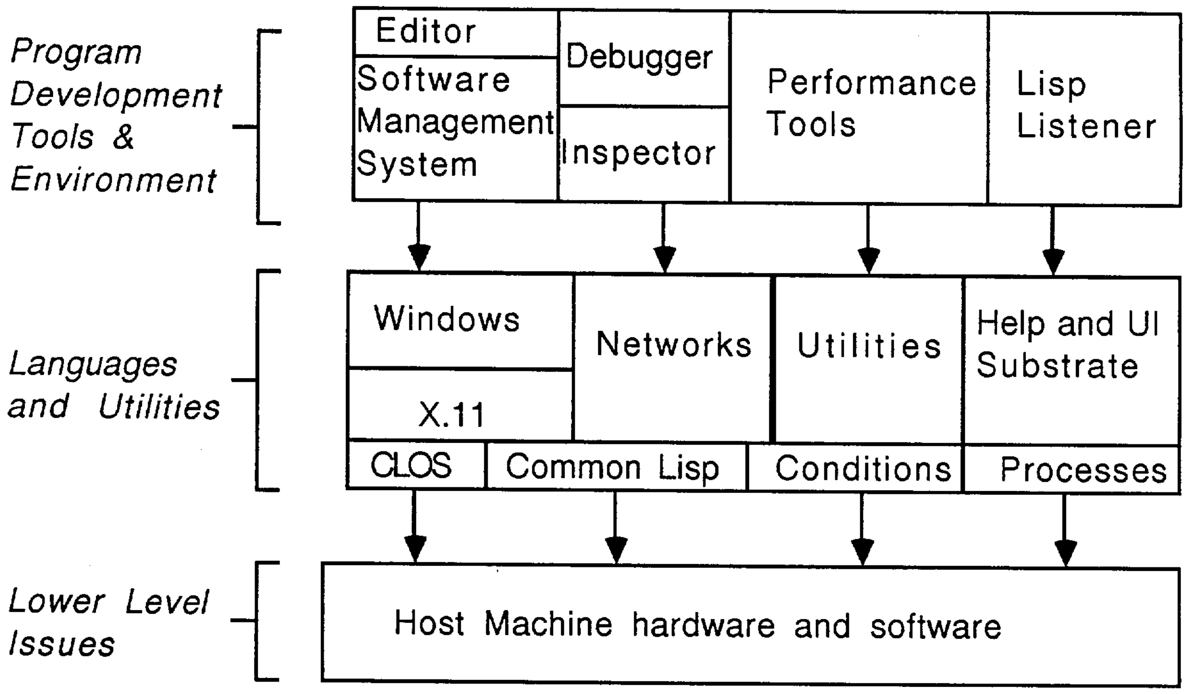

Figure 1

2. Program Development Tools & Environment

The quality of the development tools and environment is what has been the primary strength of Lisp machines, allowing rapid design, implementation, and debugging of complex programs. We believe the key to good development tools is integration, both in terms of consistency of interface, and in the ability to move seamlessly from tool to tool, carrying along appropriate data and state information. These qualities must be manifest in any KSL research computing system.

2.1. Editor

The hands of the development environment is the editor. There has been a great deal of experimentation with various styles of editing, most significantly text-based, as in Zmacs on the TI Explorer and Symbolics machines, versus structure-based, as in Xerox Lisp. We are inclined to believe that an editor rooted in a text-based approach but having understanding of the structural content of code being edited is the best approach as it allows full base-level generality for dealing with all kinds of text but, if well implemented, can be specialized for various types of editing. Given this, we feel the editor in the Lisp systems should have the following features:

- since it is common to build tools that utilize editing it is important to have a complete programatic interface to the editor
- able to use Emacs-like commands for hands-on-keyboard control as in Zmacs
- also uses pointer for moving editing focus, selection, some command selection, etc.
- fully extensible in Lisp
- uses the pretty printer described above to allow code reformatting (eg. for narrower/wider windows)
- minimally includes some source libraries to be used as examples
- supports keyboard macros
- integrated with Lisp such that edit definition, incremental compilation, documentation string viewing, argument list viewing, macro expansion, evaluation, etc. are available easily from the editing environment
- knows lisp syntax such that users can manipulate Lisp expressions (eg. move forward on s-expression, select an s-expression, etc.)
- has a complete edit definition facility such that the source of a DEFSTRUCT, DEFUN, DEFCLASS, or other definition of symbol will be automatically loaded if available without the need for explicit cross referencing
- allows user-defined modes for non-Lisp
- if text can be selected and operated on the selected text should be highlighted as in region marking in Zmacs on the TI Explorer
- if the matching grouping character is visible when the editing focus is on a grouping character, it should be indicated; a "grouping character" is a parentheses, curly brace, square brace, angle bracket, double quote, or other user-specified character; a "match" is found when a symmetric character is found in a syntactically legal place (ie. not in a different context than the focus character, such as in a comment or literal string when the focus

character is in code), thus requiring that the editor have "understanding" of the syntax being used

- allows multi-fonting and font shift stripping (eg., writes #2\a to files)
- has the ability to automatically place code into different fonts depending on context
- if the editor allows code with unbalanced parentheses to be entered, it should be possible to check the code for unbalanced parentheses, and such a check should be done when the code is saved in a file (eg. x-X Find Unbalanced Parentheses in TI and Symbolics' Zmacs)
- can be instantiated multiple times (multi-window and multi-process)
- completion of commands and file names
- hooks for buffer switching, buffer creation, mode changes
- per buffer/window editory control variable bindings
- ZetaLisp style attribute lists or some other mechanism for telling the editor what packages, fonts, base, etc. are used with data

2.2. Debugger

The interactive debugger is also a critical part of the Lisp environment. In many ways, it can be viewed as an extension of the inspector. It should have the following features:

- "Terminal" based and window based versions for times when the window system fails
- ability to force entry to debugger from keyboard, or abort execution of running program
- ability to see and modify arguments and locals of active stack frames and closures, as well as evaluating expressions in the lexical context of stack frames running compiled code
- ability to peruse the stack easily and quickly
- ability to return from or restart an arbitrary stack frame
- able to have multiple concurrent instantiations
- *must* be robust in the face of errors that occur during the operation of the debugger (eg. can't print some datum)
- arglists and docstrings must be around and accessible
- fast startup

One of the most difficult aspects of debugging is understanding where in a program the error is occurring. To assist with this, we require a good disassembler and some level of source code debugging. The disassembler must be able to give information about operations being performed including names of variables, indication of arguments being set up and functions being called, and current execution location.

The source code debugger should indicate either the current source form being executed in a given stack frame or the most recently exited form, and allow entering the editor on that source code. It is acceptable to have to compile code with a special flag on or with special declarations in effect in order to achieve source code debugging, and a moderate (approximately twofold at the most) performance penalty in code compiled with source code debugging in effect is acceptable. We have implemented such a system at the KSL by storing the program location counter (PC) before and after the execution of each form and using this to index back to the source code. It is *not* required that the source code debugger (SCD) work with all optimizations, but it should be possible to disable those optimizers to use the SCD and still run effectively.

2.3. Inspector

The backbone of the development environment is the Inspector. This tool must be quite flexible and user customizable. Given a datum the Inspector should display it in a window in such a way as to make the structure and contents quickly apparent to the user. For instance, a DEFSTRUCT structure might be displayed with a column with field names on the left and values on the right. It should further be possible to ask for alternative perspectives so that the user could view a list as a simple list of items, an ALIST, or a PLIST, and similarly for structures. The mouse should be used to traverse data structures by further inspecting. The display should not be strictly tabular to admit to nested data, graphs, etc.. The Inspector should also have the following attributes:

- there should be a well-defined protocol to allow instances to display themselves and have non-standard mouse sensitivity in the inspector
- able to work with all CL types, CLOS objects, compiled code, stack groups, and other objects that can be found in the system.
- startup quickly
- format only what is visible so that users don't have to wait for formatting of large data structures of which they wish to view only a small part
- allow fields to be modified easily
- handle circular data structures, preferably using the Common Lisp #1=(a b #1#) notation; (a b ...) is not acceptable.
- use multiple windows (one per datum) with a separate history window
- support concise and verbose modes so that, for instance, an instance being viewed as part of another structure could display itself briefly, but be more detailed when being viewed directly

2.4. Software Management Tools

An often neglected component of the development environment is a tool to manage software systems, keeping track of versions, patches, compilation and load dependencies, etc. This has frequently been handled in the past with simple command procedures and the file system. These primitive mechanisms fail to handle many cases that are becoming more and more important such as version checkpointing and multiple programmers, as well as needing version numbers in the host file system to support backup versions. We have explicitly not required version numbers on files, but only under the condition that the usefulness of the version numbers is addressed in the software management system (SMS). The SMS should have the following features:

- in the SMS, even more than usual, quick response and non-intrusive function are critical so that users aren't tempted to circumvent the system, thus ruining its integrity
- allow multiple versions of objects to be kept for both backup and for release cycling
- allow for patches to "released" systems
- allow partial or complete recompilation of systems, automatically taking care of dependencies
- allow transitive dependencies so that if system A depends on system B and system C depends on system A, manipulations of C cause both A and B to be affected appropriately
- support team programming via object or module "check out" (ie. only one programmer is able to write a module, and ideally audit trails are kept); the the smaller the module size, the less chance for two programmers requiring write access to a module at the same time
- can be either file or object based

2.5. Performance Monitoring and Analysis

An important aspect of writing software is the ability to find out where programs are spending their time so that tuning work can be applied appropriately to sluggish programs. Thus we require the following performance measurement facilities:

- stack sampling wherein a record of what functions are active on the stack is recorded at small intervals
- function entry counting
- accurate meters; microsecond precision desired

2.6. Lisp Listener

There must be a "listener" or "top-level" which is how the user interacts with the read-eval-print loop of Lisp. Along with the terminal oriented listener there should be a window based implementation (ideally based on the system editor) with the following features:

- a history mechanism allowing access to past typein and results (in at least a text form)
- editing ability, including using the pointer

3. Languages and Utilities

3.1. Windowing

Currently, the best way for a computer to present information to an interactive user seems to be via digital images presented on CRT displays. The display is divided into sub-displays called "windows" allowing various pieces of information to be presented at once. The programmatic and user interfaces to this mechanism is called the "window system".

It is very difficult to fully specify a window system complete, flexible, and efficient enough to be what everybody needs and will need. Therefore, the most important aspect of the desired window system is that it be able to evolve as more is learned about user interface and data presentation. Therefore the window system must be well layered and modular to support incremental mutation and experimentation.

In particular, we expect to see tools on top of an application toolbox, on top of window system primitives, on top of a window transport protocol (such as CLX with X.11), with the inter-layer communication passing through well-defined CLOS protocols such that new layers can be implemented with a minimum of trouble. In particular, we also expect to routinely use remote windowing capabilities, so it's very important that the windowing protocol be a standardized one accessible from many machines, such as X.11. Standards in other layers should be used as they become available and appropriate.

The layering approach also allow flexibility in display devices, and if well implemented, would allow easy redirection of output to a "display" device that happens to be a printer to give high-resolution hard copy, such as seen in Xerox Lisp's ImageOp facility, as well as color displays, higher or lower resolution displays, files, etc.

We expect people to be developing tools under at least two different windowing paradigms, including the "messy desk" metaphor which is characterized by many small (relative to the display size) windows each of which is an application or piece thereof, like a desk with many papers on it, and the "display swapping" metaphor in which there are a few applications, each of which typically takes the whole screen when active, though the individual applications usually have smaller "panes" in the display-filling "frame", and the user swaps which application is on the display via some keystroke sequence. There are arguments for both styles, and we would like our next generation system to support both styles to the extent possible. Generally, we feel the "messy desk" approach is the more general of the two, and the more sought after, and so should receive the most attention.

With that general framework, here are some specific requirements for the window system (Note: the "as in" comments below are intended to give examples of current systems with the type of functionality we wish to describe, and are not indended as strict specification of how the functionality should be implemented):

- since we can't predict the needs of our programmers in this time of rapidly changing user interface technology, we have to insist on access to the source code for at least the higher levels of the window system, and that the inter-layer protocols be well documented and flexible

- while we expect almost all interactions to be window-based, it will be necessary at times to access the Lisp from a non-windowing system or terminal so we need terminal-like interfacing to be available on at least a rudimentary basis making it possible, for instance, to check on a long-running program from a remote location over a crude link

- provide programmer specifiable on-screen mini-doc about available mouse actions, and use this facility in system tools
- there must be a way to reestablish connections to the window system from remote hosts without restarting Lisp so that if, say, a network connection fails the user can reconnect and reattach to the Lisp session if the operating system hasn't killed it
- horizontal and vertical scrolling based on customizable redisplay (wherein the user programs how to fill in the newly exposed window area) such as in Symbolics Lisp
- hierarchical (nested) window structuring as in TI Explorer Lisp
- high tolerance for "logical" errors, so that minor operational errors, such as incorrectly reshaping a window, don't cause the window system to crash
- non-restrictive parameters such the maximum number of windows, depth of the window hierarchy, size of a window, etc.
- possible to write to non-exposed windows as in the Apple Macintosh
- a window title mechanism (with option of attaching mouse handlers to titles) as in TI Explorer Lisp
- customizable scroll bars (eg. size, location, color, pattern, pop-up, mouse-capturing, mouse-click actions, extra "buttons", etc.)
- constraint frames that allow automatic configuration of inferior windows when the superior is altered as in TI Explorer Lisp
- fast opening windows, with no more than fraction of a second delay for typical windows in typical circumstances
- scrolling and character drawing rate of at least 1,500 char/sec in a full-screen window
- operations that work in color and B/W
- optional color
- use n-dimensional abstract positions (not X and Y coordinates), even if the window system only uses 2-d points in order to allow for future display devices that may well be able to deal in 3-d
- documented font formats so users may define new fonts
- ability to use any font, any time without using a "font map" (requiring separate operations to ensure proper baseline calculations is ok)

- customizable, titled pop-up menu styles with 2 standard styles: roll-out (the menu stays in place with no buttons down until an item is selected or the mouse is moved away from the menu as in TI Explorer Lisp) and button-up (the menu stays up while a mouse button is held down, selecting the item the mouse is over when the button is released or none if the mouse is outside the menu as in Xerox Lisp); "pull down" menus should be implementable

- a "snapshot" facility to allow a section of the screen image to be recorded in another window for later viewing, printing, or saving on a file in a published raster format

- ability to shrink application windows into smaller icons as in Xerox Lisp

- customizable event distribution (mouse clicks, etc.)

- user-extensible pop-up windows used for entering data ("dialog box") such as seen on the Apple Macintosh dialog boxes or the TI Explorer Choose Variable Values menus.

- allow CLOS instances to display themselves, allowing graphical menu items, etc.

- there must be a rich set of facilities for running user specified code when the pointing device enters or exits a region, and when a region is "touched" by the pointing device; these "active regions" must:

    - be able to be non-rectangular, though they may be constrained to be rectilinear and congruent to the X and Y axis of the window system

    - have built in ways to work in scrolling windows

    - have built in ways to be highlighted via boxing, inverting, and color washing on color displays when the pointer is inside the region

- the ability to channel mouse events and keystrokes through streams

- a modular way to add items to menus in tools

- the mechanism for redisplaying windows that are being uncovered or moved should be flexible and programmable so that, for instance, windows need not have a data structure that stores the pixels associated with the window when they are not visible, a technique that can be expensive when there are many large windows or many bits per pixel

- mechanism for synchronizing keyboard and mouse so switching windows works smoothly

- have available variable width fonts

The following items are features which would be very useful, but are not required:

- optional larger or multiple displays as in the Macintosh II

- have available primitive drawing operations that are very low overhead (non-consing and fast)
- coordinate transforms
- allow access to low level color/frame buffer control for some applications
- kerning (changing the position a character is drawn in when it is next to certain other characters) of certain character combinations or at least pseudo-kerning (offsetting certain characters in a font by a small amount to improve the aesthetics of the resulting text

3.2. Multiple Processes

A critical part of any powerful software development environment is the ability to run multiple tasks simultaneously. In Lisp it is particularly important that one have access to multi-tasking within the single address space of the running Lisp environment so that cooperating agents can freely access shared data. These "processes" should have the following properties:

- inexpensive in terms of system resources and time to create/use (lightweight) (to assist this, it's advisable that a process not have things like an I/O window until it's needed)
- processes should be preemptable and should be scheduled under a priority and quantum based scheme
- locks and events should be available to control synchronization of cooperating agents so that, for instance, a data-producer could signal an event that would re-activate a consumer process
- the scheduler should maintain queues rather than typically calling a "are you runnable" routine to avoid high overhead when there are many processes and events should be used to move processes onto run queues
- a complete set of operations to control processes (eg. [un]arrest, kill, change priority/quantum, inspection of statistics/top-level functions/etc., and interrupt)
- the keyboard attaches to processes, not windows
- the notion of a stack or stack group should be separate from the actual process
- processes should be CLOS objects

It is conspicuous that unless process switching is a very, very low overhead operation, of the order of 2-4 function calls, the scheduler shouldn't run in it's own stack, but run on the last running processs' stack so that undue overhead isn't introduced.

It is desirable to have but potentially difficult to implement a system wherein when one process waits for an event like I/O or a page fault other processes are able to continue. We would very much like this feature, but do not require it.

3.3. Common Lisp

Lisp is the computing language of choice in the KSL and is likely to remain so due to it's utility in the type of programming required for the KSL's research, as well as the tremendous amount of available experience in building strong computing environments in Lisp and the strong commitment already in place.

We feel it is critical that the Lisp be Common Lisp, as described in Guy Steele's "Common Lisp the Language" today, and as specified by ANSI's X3J13 in the future.

It is essential that with the Lisp there is a strong object oriented programming system. In particular, the Common Lisp Object System {ref} (CLOS) should be fully supported, being well integrated with the Lisp support environment, and used where appropriate in system software. The CLOS specification has not yet been completed, but major parts of it are essentially complete and have been accepted by X3J13 and so, given the critical role of the object system, we feel that even an implementation of the partial specification is important, along with further implementation as the specification matures.

Similarly, we require a condition handling system, and in particular the Common Lisp Condition System {ref}, under conditions similar to CLOS. The condition system should also provide a mechanism to catch and abort "trivial" errors committed during top-level typein such as unbound variables and undefined functions and a mechanism allowing searching for functions or symbols in other packages when they are undefined (package DWIM).

The system must also support the following:

- Large FIXNUMs (at least 24 bits)
- IEEE Floating Point Numbers {what's the IEEE spec name?}

3.4. Compilation

Mondern Lisp systems almost always "compile" the Lisp source code into instructions more suited to the architecture of the machine in use. As execution speeds increase and the size of problems being tackled increases it's important that compilation time not introduce painful delays into the development loop, ruining the quick edit-compile-run cycle characteristic of Lisp. The compilation delay for "small" code units (approximately 10 to 50 lines of code not involving heavy macro expansion), should be negligible, while mechanisms should be available for larger "batch" compilations as part of the software management system.

It is acceptable to have mutiple ways to execute the same source code, such as an interpreter, a compiler that executes quickly but produces slower code, and a compiler that executes slowly but produces faster code. However, it is absolutely essential that all of these have indistinguishable semantics.

The following features should be present with compilation:

- Compilation of individual top-level forms (incremental compilation)
- Complete compilation of all forms, including closures and other lexical functions
- Ability to cause code to be compiled in-line via the use of declarations

- Ability to do unboxed floating point operations if appropriate declarations are present

- Documentation on built-in compiler optimizers

- The ability for the user to define new compiler optimizers

- Optimization of tail recursive calls

- Automatic compilation of "encapsulation" code such as ADVISE or TRACE (described elsewhere) so that at the time of the encapsulation the associated code is compiled

To achieve further portability, we would like to see cross compilation or multi-targeting capability, though this is not strictly required.

3.5. Input/Output

Moving data into and out of Lisp is something that is done quite a bit during the normal operation of the machine, so I/O performance must be on par with the rest of the system. In particular, loading compiled files (FASLOAD), reading source files into the editor, loading source files (READ), file probes (OPEN and FILE-WRITE-DATE) (typically done in the software management system) must be quick. As a guideline, we would expect read/write speeds into and out of the Lisp world of close to 40 kilobytes (kB) per second to a network file server, or 150 kB/sec to a local disk, as seen on the Explorer II. Additionally, the system should support a large number of simultaneously open files (certainly 30 or more), as well as multiple streams (input and output) to the same file. Access to remote files should be transparent to the Lisp user (ie. no special "copy to the local system" step should be needed to access data available via filing protocols including at least NFS.)

3.6. Utilities

There are a number of environmental utilities needed to use the system effectively, including:

- a way to save a lisp image for later reuse, as in TI's DISK-SAVE and Xerox's SYSOUT

- a mechanism for "advising"; wrapping code around entry points to affect the behavior of code as in TI Explorer Lisp's ADVISE

- routines for accessing network facilities (TCP, etc.)

- a WITH-TIMEOUT routine that would allow execution of some code to be aborted if it does not complete within the alloted time

- trace, including internals (FOO-in-BAR, LABELS, FLET, closures)

- a facility for searching the world for symbols (APROPOS)

- ways to determine what functions call another function or use a particular special variable, as well as to determine what functions are called by a (and specials used by) a particular function

- routines for laying out and drawing hierarchies and graphs where the nodes and edges can be instances that draw themselves and define their own mouse sensitivity
- a foreign language interface on machines which support non-Lisp languages
- remote procedure call (Sun RPC)
- stream interfaces to various facilities such as networks, windows, and printers
- a single stepper, probably only on interpreted code
- file properties including write-date, author, security status, locking, and properties native to the operating system; ideally the user would be able to define and use arbitrary file properties
- ability to restart Lisp in the same address space, allowing one to reinitialize windows, processes, etc. without losing edits and other work
- routines for manipulating time values
- a mechanism for reconstituting structure definitions (DEFSTRUCT)

3.7. Interface Toolkit

While there is a good deal of disagreement about what user interfaces should look like it is generally accepted that consistency within a system is worth working for when other considerations aren't overriding. Thus, to encourage overall consistency, we require that the higher level programming tools be built on a single interface subtrate such that they have consistent use of menus, typein, display format, etc., and that this tool be available to the user. An examples of this type of tool is TI's Universal Command Loop.

3.8. Help System

Important to the overall usability of the system is good "novice" support in the form of some combination of on-line help (eg. general help files), on-line tutorials (eg. the Machintosh Guided Tour), context sensitive help (including menus of commonly used commands and completion) (eg. TI's Suggestions Menus), on-line documentation (eg. doc strings in functions and variables), primer documentation, and informative error messages. The help system should be consistent, used in all the system tools, and useable in user written programs.

3.9. Status Information

It's important for the user, and especially, the programmer to get good information about the current operational status of the machine. The TI and Symbolics WHO-LINE and Sun's Perfmeters are examples of this sort of facility. We feel that some information about the following should be includable on the screen at all times:

- gc activity
- cpu used by lisp

- paging
- consing
- system load
- current package
- status of process owning the keyboard
- what process owns the keyboard
- lisp file activity

3.10. Printing

While the capabilities of the system being specified will encourage a paperless office, hardcopy printing is still an important part of our activities for debugging, passing along information, and keeping records. The printing system should:

- use generic operations as in Xerox Lisp (see window system discussion)
- allow users to add new printer types/drivers
- support at least PostScript initially
- allow printing of unformatted files, formatted files, and window images

3.11. Pretty Printing

The ability of the system to format output, especially in a window based environment, is important to the user's ability to understand the data being displayed. Thus, a "pretty printing" facility must be included with the following features:

- user customizable
- has a protocol to interact with instances so that they can make formatting decisions
- interprets arg lists for macros and formats accordingly so that, for instance, &BODY arguments get formatted as code
- works with intermediate data structures so that entire expressions need not be printed for efficiency with scrolling windows, especially in the inspector
- can format user-defined mouse sensitive data

4. Lower Level Issues

4.1. Address Space

As the size of problems being addressed increases so does the need for address space in Lisp systems. It is difficult to quantify address space requirements as they are affected by other facets of the system including effectiveness of the memory management system and the space required by data strucutures, but we can say that we need the potential for a very large address space, such that the address space is typically limited by how much disk it's feasible to have rather than the number of bits used in addresses. Good examples of today's systems are the Symbolics 3600 {number} or the TI Explorer with Extended Address Spaces {number}. Note that systems that migrate unused objects out of the primary address space should allow a primary address space of at least 100 megabytes. We expect this requirement to expand in the future.

Conditions should be signalled when address space gets to a user-definable minimum, with default handlers that will notify the user of the low address space condition.

A parallel to address space is stack size. Recursive or other deeply nested programs must work without modification and so we require that the execution stack be expandable at run time as with the TI Explorer and Symbolics systems, or very, very large. Minimally the stack should be large enough to run 5,000 function call levels with an average of 4 arguments and 4 locals per level in the most stack-hungry execution mode (usually interpreted).

4.2. Memory Management

A frequent thorn in the side of Lisp programmers is the reclamation of allocated but no longer used memory, or garbage collection (GC). Therefore, we require that:

- in general, GC take no more than 10% total overhead, with less being very desirable
- no programmer/user intervention be required in normal operation
- the amount of time that the machine is made unavailable to the user is limited to a few seconds at a time either by time limiting the amount of work done at once or by using a concurrent system, with either solution implying a dynamic algorithm
- the working set not be unduly expanded by GC operation to avoid thrashing
- there be controls available to the programmer to tune the GC to a particular program, or to inhibit it at times for real-time program segments or for timing
- there be finalization code associated with some objects like CLOS instances for cases like the need to release resources that aren't resident in the Lisp address space

In addition to the automatic memory management software there should be tools that allow a programmer explicit control over storage allocation with notions similar to "area"s for new allocation which can be declared exempt from GC or to be deallocated in bulk. Also allocation aids such as RESOURCE structures should be provided.

4.3. Dedicated Versus Shared Systems

Timesharing, as opposed to having a processor dedicated to a single user, is acceptable in principle, though it is important that it be well done in the sense that users not step on each others' toes by doing simple things like running programs. In particular the scheduler and paging algorithms should be such that if the system is claimed to support N users, all N users should simultaneously see the kind of minimum responsiveness and performance we require.

4.4. Hardware Capabilities

The display should be able show approximately 70 lines of monochrome text with 130 columns each and still be read comfortably, such as the approximately 1024 x 768, 72 dot/inch black and white displays found on TI Explorers and Symbolics 3600 class machines, with a strong desire for being able to display two 80 character wide windows side by side, as Sun workstations with hi-resolution displays and Xerox 1186's with 19" displays. The display must be stable and crisp, which means it should probably be non-interlaced. It must also be possible to get a video output for demonstrating software to large audiences.

The data input mechanism should support a rate at least equal to that found in accomplished touch typists (approximately 70 words per minute), such as a keyboard with 2 key or more rollover, as well as a fast pointing device equipped with at least two, and preferably three kinds of "touches", such as a 3 button mouse, a way of programming idioms into short cut sequences, such as programmable function keys on a keyboard, and a way of sending non-text commands which would mean at least two (Control and Meta) modifier keys on a keyboard.

4.5. Overall System Integration in the KSL

Any new computing systems in the KSL must be able to fit in to the existing environment and interact with pre-existing systems to facilitate sharing data, moving users from system to system, and system administration. Incoming systems should have the following properties to integrate well into the KSL environment:

- good networking including filing (Sun IP/UDP/NFS and IP/TCP/FTP minimally), virtual terminal service (IP/TCP/TELNET), remote procedure call (Sun IP/UDP/RPC), and name service (IP/UDP/DOMAIN)

- provisions for file backup, possibly via NFS

- a large limit, if any, on file names, with 40 characters per field minimum

- if the system is a workstation, the user must be able to reboot from the console and be able to run first-order diagnostics to determine in most cases what major component is responsible for any failure

- the element of the system that sits in offices should have minimal power requirements, thus requiring no additional air conditioning capacity, and should not generate distracting noise; if the system is a workstation unit, it is probably necessary to remote the processor from the display to achieve this requirement and we would need at least 500 feet of potential separation to reach from our office spaces to our machine room spaces; for example, we consider the TI Explorer too noisy and hot to have the system unit in most offices, and consider most Apple Macintosh II's to be just under the acceptable noise level

5. Acknowledgement

The summary presented here is the result of extensive discussions among the following group: Bruce Delagi, Christopher Lane, James Rice, Thomas Rindfleisch, and Eric Schoen.

Appendix D

AIM Management Committee Membership

Following are the current membership lists of the various SUMEX-AIM management committees:

AIM Executive Committee:

SHORTLIFFE, Edward H., M.D., Ph.D. (Chairman)
Principal Investigator - SUMEX
Medical School Office Building, Rm. X271
Stanford University Medical Center
Stanford, California 94305
(415) 723-6970

FEIGENBAUM, Edward A., Ph.D.
Co-Principal Investigator - SUMEX
Heuristic Programming Project
Department of Computer Science
701 Welch Road, Building C
Stanford University
Stanford, California 94305
(415) 723-4879

KULIKOWSKI, Casimir, Ph.D.
Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903
(201) 932-2006

LEDERBERG, Joshua, Ph.D.
President
The Rockefeller University
1230 York Avenue
New York, New York 10021
(212) 570-8080, 570-8000

LINDBERG, Donald A.B., M.D. (Past Adv Group Chrmn)
Director, National Library of Medicine
8600 Rockville Pike
Bethesda, Maryland 20814
(301)496-6221

MYERS, Jack D., M.D.
School of Medicine
Scaife Hall, 1291
University of Pittsburgh
Pittsburgh, Pennsylvania 15261
(412) 648-9933

AIM Advisory Group:

MYERS, Jack D., M.D. (Chairman)
School of Medicine
Scaife Hall, 1291
University of Pittsburgh
Pittsburgh, Pennsylvania 15261
(412) 648-9933

AMAREL, Saul, Ph.D.
Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903
(201) 932-3546

COULTER, Charles L., Ph.D. (Exec. Secretary)
Bldg 31, Room 5B41
Biomedical Research Technology Program
National Institutes of Health
9000 Rockville Pike
Bethesda, Maryland 20892
(301) 496-5411

FEIGENBAUM, Edward A., Ph.D. (Ex-officio)
Co-Principal Investigator - SUMEX
Heuristic Programming Project
Department of Computer Science
701 Welch Road, Building C
Stanford University
Palo Alto, California 94305
(415) 723-4879

KULIKOWSKI, Casimir, Ph.D.
Department of Computer Science
Hill Center Busch Campus
Rutgers University
New Brunswick, New Jersey 08903
(201) 932-2006

LEDERBERG, Joshua, Ph.D.
President
The Rockefeller University
1230 York Avenue
New York, New York 10021
(212) 570-8080, 570-8000

LINDBERG, Donald A.B., M.D.
Director, National Library of Medicine
Building 38, Rm. 2E-17B
8600 Rockville Pike
Bethesda, Maryland 20814
(301) 496-6221

MINSKY, Marvin, Ph.D.
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139
(617) 253-5864

MOHLER, William C., M.D.
Associate Director
Division of Computer Research and Technology
National Institutes of Health
Building 12A, Room 3033
9000 Rockville Pike
Bethesda, Maryland 20892
(301) 496-1168

PAUKER, Stephen G., M.D.
Department of Medicine - Cardiology
Tufts New England Medical Center Hospital
171 Harrison Avenue
Boston, Massachusetts 02111
(617) 956-5910

SHORTLIFFE, Edward H., M.D., Ph.D. (Ex-officio)
Principal Investigator - SUMEX
Medical School Office Building, Rm. X271
Stanford University Medical Center
Stanford, California 94305
(415) 723-6979

SIMON, Herbert A., Ph.D.
Department of Psychology
Baker Hall, 339
Carnegie-Mellon University
Schenley Park
Pittsburgh, Pennsylvania 15213
(412) 578-2787, 578-2000