XLISP-STAT 2.1 Release 3
Beta Release Notes

Luke Tierney
School of Statistics
University of Minnesota

January 4, 1996

Contents

1 Introduction

This note outlines some of the changes in the new XLISP-STAT release from Release 2. There are a large number of minor changes, mostly in the basic XLISP system. These changes move the basic XLISP closer to Common Lisp, and include the addition of packages, multiple values, other improvements from XLISP-PLUS 2.1g, and a byte code compiler. Wherever possible, changes have been made in such a way as to maintain backward compatibility. In a few, hopefully minor, cases this was not possible; these cases should be noted below.

> *If you notice anything that has changed and is not mentioned here, please let me know. If any change causes exceptional grief, please let me know as well.*

2 Changes to System Features

2.1 Debugging and Evaluation

The error handling in XLISP-STAT is now based on the Common Lisp condition system. When debugging is turned on, an error enters a break loop. The break loop presents a list of the available restarts, and the function `continue` can be used to select a restart:

```
Error: illegal zero argument
Break level 2.
To continue, type (continue n), where n is an option number:
 0: Return to break level 1.
 1: Return to Lisp Toplevel.
2>
```

When debugging is not turned on, an indication of the function in which the error occurred is printed before the system returns to the top level:

```
Error: illegal zero argument
Happened in: #<Subr-/: #c49e20>
>
```

The `baktrace` function now accepts a second optional argument that determines whether it prints function arguments or not. Supplying `nil` as this argument suppresses argument printing. The default value for this argument is the value of the special variable `*baktrace-print-arguments*`.

The back trace does not show entries for byte compiled functions or internal functions called from byte compiled code.

The old `step` function has been replaced by a new one from the the stepper file included in XLISP-PLUS 2.1g. Detailed documentation for this new `step` is given in Appendix A. The Macintosh version of `step` no longer has a dialog interface.

The action taken in response to a user interrupt is now customizable. If the value of the special variable `*interrupt-action*` is `nil`, interrupts are ignored. Otherwise, the value should be a function of no arguments that is called in response to an interrupt. The default value is the `top-level` function that returns to the top level.

Debugging may be affected by the handling of macros. If the special variable `*displace-macros*` is non-`nil`, then macros are spliced into the code after they are expanded. This means macros are only expanded once and can significantly improve speed of interpreted code. But it may make code used in stepping less readable, so it may be useful to set this variable to `nil` during debugging.

XLISP can now be more strict in checking for keywords. If `*strict-keywords*` is non-`nil`, then specifying keyword arguments not defined for a function signals an error unless the function is defined using `&allow-other-keys` or the keyword arguments include `:allow-other-keys` with a non-`nil` value. If the value of `*strict-keywords*` is `nil`, then unmatched keywords are always ignored. For now, this is the default setting.

Some graphical control over the system option variables, choosing restarts, etc. is clearly needed.

2.2 Top Level and Saved Workspaces

It is now possible to save workspaces and customize the initialization and top level functions. The function `save-workspace` takes a single argument, the name of the saved workspace file. It closes all windows and menus, creates the saved workspace, and exits from the system. The workspace file is created with a `.wks` extension.

On UNIX, and eventually MS Windows, a saved workspace can be loaded by specifying `-w`*file.wks* (with no spaces) as a command line argument. On a Macintosh, you can double click a workspace. If no workspace argument is given, the workspace named `xlisp.wks` in the startup directory is used if one is available. Otherwise, a file `init.lsp` is loaded. (So `init.lsp` is not loaded if a workspace is found). It is currently not possible to load a new workspace once the system has started up.

After loading a workspace or the `init.lsp` file, files on the command line (or files that where double clicked on the Macintosh) are loaded. The command line used to start up XLISP-STAT is also made available as a list of strings in the variable `*command-line*`. (On the Macintosh this variable currently does not contain anything useful.) Next, the functions of no arguments in the list that is the value of `*startup-functions*` is called. One of the functions in the default value of this variable is responsible for loading the `statinit.lsp` file. Finally, the function `*top-level-loop*` is called. This function is called again each time the system is reset to the top level.

The installed versions of xlispstat are now set up to start from a default `xlisp.wks` saved workspace rather than by loading a set of `.lsp` files. Compiled versions of files needed less frequently and help information are still kept in a library and loaded on demand.

> *The basic objective here is to make startup and the top level customizable. The details are likely to change somewhat as I get things straightened out on the Mac and Windows versions.*

> *The ability to save workspaces (and to byte compile files) creates a problem for using* `#+` *and* `#-` *to test for runtime properties. These tests occur when a file is read. But a file can be read on a computer with a color display and the definitions can then be used from a saved workspace on a monochrome display, for example. As a result,* `#+` *and* `#-` *should only be used for properties, such as the operating system type, that will not change from one invocation to another. Runtime tests,*

such as the function `screen-has-color` *should be used instead. All system files should have been rewritten with this in mind.*

2.3 IEEE NaN's and infinites

Three constants are defined to support IEEE NaN's and infinities:

`positive-infinity` `negative-infinity` `not-a-number`

When printing an infinity or a NaN when `*print-escape*` is `nil`, the appropriate symbol is printed using the `#.` read macro. For example,

```
> positive-infinity
#.POSITIVE-INFINITY
```

This insures that all numbers are printed in a readable form, even infinities. All NaN's are printed identically, there is no provision for preserving variations among NaN's.

There is currently no provision for checking for a NaN or infinity. There probably ought to be lisp equivalents of the IEEE-recommended functions `isnan` *and* `finite`*.*

At present the symbols used here are external in the XLISP package.As they are nonstandard, they should probably be internal in a system package.

2.4 New Garbage Collector and Memory Requirements

The original mark-and-sweep collector of XLISP has been replaced by an in-place two-generation generational collector. This means that there are now two types of garbage collections, minor and major ones. Minor collections occur relatively frequently but are very fast. Major ones occur rather rarely in typical usage, and are only marginally slower than mark-and-sweep collections. The net effect should be to reduce garbage collection pauses significantly.

In addition, the system is more aggressive about allocating more space when a substantial fraction of the allocated space is in use. This should also reduce the number of collections, but may lead to greater process memory growth. The impact on limited memory systems such as a Macintosh with a small partition is not clear at this point.

The new garbage collector requires more storage space per Lisp node. The overall requirements are increased by around 30%-50%. The default partition on the Macintosh will therefore be increased to 3M.

2.5 Missing and Non-Numerical Data in Graphs

It is now possible to give plotting functions data sequences that contain non-numerical values, such as missing value codes. Points with one or more non-numerical coordinates are marked internally as masked. This means they are not drawn and are not considered for the calculation of scaling information.

Any non-numerical lisp item can be used as a missing data code, but `nil` should be avoided since it also represents the empty list and can cause confusion in vectorized operations.

2.6 Working Directories

The function `get-working-directory` returns a string naming the current working directory. The function `set-working-directory` sets the working directory to the directory specified by its string argument. If the change of directory succeeds, `t` is returned. Otherwise, `nil` is returned.

On the Macintosh these functions will fail if the length of the directory name is greater than 255.

2.7 Version Information

In addition to the feature `xlisp`, starting with release 3.39 the new version also defines `xlisp-plus`. The constants `xls-major-release` and `xls-minor-release` contain the major and minor release numbers of the executable. For release 3.39, for example, these are 3 and 39, respectively.

3 Changes in Specific Implementations

3.1 Changes in the UNIX/X11 Version

The initial workspace and the default directory can be specified on the commandline as `-w`*file.wks* and `-d`*dir*, respectively.

Upon startup of the *X11* version of XLISP-STAT the system merges resources found in file

```
/usr/lib/X11/app-defaults/Xlisp
```

then ones found in the file `Xlisp` in the directory named by the environment variable `XAPPLRESDIR`, and finally the resources of the display. Resources can thus be specified in all three places, with the display's resources having highest precedence.

The functions `popen` and `pclose` from `winterp` have been added to the UNIX version to allow streams connected to processes to be created.

The interface to these functions may change.

3.2 Macintosh Changes and System 7 Support

Several options can now be set in a preferences file. This file is a text file, is must be called **XLS Preferences**, and it must be located in the same folder as the application. Each line in the file represents an option specification. The first word identifies the option, the remaining words make up the option data. The option names currently supported are

EditFontName	**EditFontSize**	**EditFontStyle**
ListenerFontName	**ListenerFontSize**	**ListenerFontStyle**
GraphFontName	**GraphFontSize**	**GraphFontStyle**
Workspace	**Color**	

the **Workspace** entry specifies a default workspace to use. Unless specified separately, listener font properties are identical to editor font ones. As an example, a preferences file containing the entries

```
EditFontName Courier
EditFontSize 12
Workspace Alliance Drive:MPW:Programs:New XLS:xlisp.wks
Color off
```

specifies a 12-point bold italic Courier font for editor windows, an alternative workspace, and turns color off. Turning color off may be useful if the screen size and number of colors used would require too much memory for the background buffer.

The dashed line separating system apple menu items from application items is now supplied internally – a dash item is now no longer necessary. This should result in a single dash under both System 6 and System 7.

Under Macintosh System 7 XLISP-STAT now continues to operate while in background. If the variable `*use-notifier*` is not `nil`, then notification occurs when a modal dialog is to be opened or when the system tries to read from the listener. The notification consists of a beep, a flashing icon on the system menu, and a diamond marking the application name in the system menu.

Minimal support for examining processes on the current computer and for sending local or remote apple events is now available. The function `launch-application` takes either a file name string argument or an application signature argument and launches the corresponding application. By default the application is launched in foreground; specifying an optional argument as non-`nil` causes the application to be launched in background. The value returned is a process information record if the launch succeeds, or `nil` if the launch fails. Process information is currently encoded as a list of the process name, process signature, process serial number, time of process start and run time of the process. This representation may change.

Currently applications can only be found by signature if they are located on the boot volume.

Application signatures used by `launch-application` are integers representing four-character character constants. The function `encode-signature` converts strings with the corresponding string of characters to the signature code. For example,

```
> (encode-signature "X1St")
1479627636
```

computes the signature code for `X1St`, the signature for XLISP-STAT.

A list of process information records for all running processes is returned by `get-process-list`, and `get-front-process` returns the process information for the current front process. The `set-front-process` function takes a process name string, signature code, or process information record and makes the corresponding process the front process. If the argument is a string or integer, then the first process, if any, with the corresponding name or signature is used.

XLISP-STAT now responds to the four required Apple events (Open Application, Open Documents, Print Documents, and Quit Application). The Print event is ignored. Currently new workspaces can only be opened at startup (i. e. as part of the initial high level event), but text files can be opened by double clicking them or dragging them onto the XLISP-STAT icon at any time. In addition, the Do Script event is supported (class `'misc'`, event `'dosc'`). This event takes a direct parameter that is a string and interprets it as a lisp expression. The expression is evaluated, the result is printed to a string, and the string is returned as the direct parameter of the result. For example, if you have the `Alpha` shareware editor, then from a Tcl shell window you can send an apple event to a running XLISP-STAT process asking XLISP-STAT to make a system beep sound:

```
Alpha.5.55> dosc -c 'X1St' -s "(sysbeep)"
NIL
```

You can also send Apple events to processes on the same or on remote computers using the function `send-apple-event`. At the moment this function allows you to send events that contain parameters that are strings, integers, or floating point numbers. If a result is expected it should contain no required parameters or a required direct parameter that can be coerced to a string. In particular, this allows Do Script events to be sent to other XLISP-STAT processes. The `send-apple-event` function has three required parameters, two four-character strings specifying the event class and type, and an address specifier for the event's receiver. The address can be

- The symbol `t`, which means the current application is the target, a self send.
- A string representing a process name on the local computer.
- An integer representing the signature code of a process on the current computer.
- A process information record as returned by `launch-application` or by `get-process-list`.
- A target specifier as returned by `browse-apple-event-targets` or by `get-apple-event-target`.

Several keyword arguments are also accepted:

`:data`: Either `nil`, the default, or a string to be used as the direct parameter of the event, or a list of lists. If a list of lists is used, then the sublists must contain two items, a key string identifying the parameter (such as `"----"` for a direct parameter) and the data item. Data items can be integers, floating point numbers or strings.

`:wait-reply`: Wait for a reply if non-`nil`, the default.

`:timeout`: The symbol `t` for the default timeout (about a minute), an integer for a specific value, or `nil` for no timeout. If not supplied, the default timeout is used.

`:can-switch-layer`: Whether or not the receiver should be allowed to come to the front if it requires interaction.

As an example,

```
(send-apple-event "aevt" "abou" "Finder")
```

sends the Finder an apple event asking it to present its **About This Macintosh** dialog box.

To locate a target on the local machine or a machine on the network you can use the interactive `browse-apple-event-targets` function. This function presents a dialog box that lists available processes for all Macintoshes on a local network; if the local network is connected to an internet the dialog also presents a list of AppleTalk zones to choose from. Several keyword arguments are available for controlling the appearance of the dialog:

`:prompt`: A prompt string to show at the top of the dialog.

`:application-list-label`: A string to show over the program list item.

`:type`: The connection type, which defaults to `"PPCToolBox"`.

By default, all targets with suitable permissions are shown. The set of targets can also be restricted with two additional keyword arguments:

`:name`: A string specifying the name of the application to link to. Only applications with this name are shown.

`:signature`: An integer signature code. Only applications with this signature are shown.

You can also locate applications to link to without using a dialog box. The `get-apple-event-target-list` function returns a list of all targets available on a particular machine. By default the local machine is examined. Alternatives can be specified by keyword arguments:

`:object`: The name of a macintosh.

`:type`: The connection type, which defaults to `"PPCToolBox"`.

`:zone`: An AppleTalk zone name.

The function `get-apple-event-target` returns a single target, or `nil` if none is found. It requires one argument, an application name string or signature code, and also accepts the same keyword arguments as `get-apple-event-target-list`. If there are several possible matches for the name or signature given, an arbitrary choice is returned.

As an example, suppose an XLISP-STAT process is running on another Macintosh named `"Fred's Macintosh"` and that all permissions are suitably set. We can send a `sysbeep` command to this Macintosh using the expression

```
(let* ((sig (encode-signature "X1St"))
       (target (get-apple-event-target :object "Fred's Macintosh")))
  (when target
        (send-apple-event "misc" "dosc" target :data "(sysbeep)")))
```

If successful, the expression returns the string `"NIL"`.

> *The support included for Apple events is rather minimal. The main reason I have not yet made it more extensive is that I don't know what kinds of events will be most useful for communicating with other applications. If anyone has any ideas or would like to help extending this, please let me know.*

One problem that comes up on the Macintosh is the need to account for the screen size. Currently a background bitmap the size and depth of the primary screen is allocated within the partition. The size of this bitmap is

$$\frac{\text{width} \times \text{height} \times \text{depth}}{8}.$$

For a 512×320 monochrome display this is fairly trivial, and a 3M partition should be adequate for most purposes (though this of course depends on what is loaded). But for a 1024×1024 display using 32-bit color, this is 4M. At the moment, this calculation needs to be done manually and you need to make adjustments to your partition accordingly. I will try to come up with a better solution in the future.

3.3 Changes in the Microsoft Windows Version

The new standard version for Microsoft Windows now requires at least a 386 processor and Windows version 3.1. This version is still based on 16 bit windows and hence still has the same limitations on vector sizes as previous versions. In addition, a Win32 version is now available that removes some of these limitation.

In line with the use on Windows 3.1 as the base, accelerators for the edit menu items in WXLS and in LSPEDIT have changed to the ones used in the 3.1 accessories Write and Notepad: *Ctrl-X* for Cut, *Ctrl-C* for Copy, etc.. The interrupt key combination is now *Ctrl-Break*. The functions `open-file-dialog` and `set-file-dialog` now use the new standard Windows open and save dialogs and also optionally set the working directory to the directory containing the selected file.

The listener has been modified so that is can now hold a more reasonable amount of text and should no longer run out of memory.

Modeless dialogs, such as slider dialogs, are now MDI clients rather that popup windows.

The need to set environment variables has been eliminated. Instead, information about the location of startup files can now be placed in a private initialization file, `wxls.ini` for the 16-bit version and `wxls32.ini` for the 32-bit version, in the Windows system directory. This file can contain several variables in sections `[Xlisp]`, `[Listener]`, and `[Graphics]`:

Section	Variables
`[Xlisp]`	`Libdir`, `Workspace`
`[Listener]`	`Font`, `FontSize`
`[Graphics]`	`Font`, `FontSize`, `Color`

The `Libdir` variable should be set to the directory containing the executable and runtime files; this should be done at installation time by loading `config.lsp`. The `Workspace` variable allows an alternate initial workspace to be specified. Both can be overridden on the command line by specifying `-d` or `-w` options, respectively. The `Color` variable can be used to turn color use off if allocating a color background buffer would require too much memory. As an example, a `wxls.ini` file containing

```
[Xlisp]
Libdir=C:\WXLS
[Listener]
Font="Courier New Bold"
FontSize=13
[Graphics]
Color=off
```

specifies a library directory, a listener font, and turns color off.

The initial workspace and the default directory can also be specified on the commandline as `-w`*file.wks* and `-d`*dir*, respectively.

A very minimal DDE interface is provided. The interface is very simple and based on DDEML. As a server, WXLS allows to connections to the topic `XLISP-STAT` under the service name `XLISP-STAT`. It accepts two kinds of transactions:

- Execute transactions in which the command is a sequence of lisp expressions. The LSPEDIT application sends the current selection as an execute transaction when the **Eval Selection** menu item is chosen.
- Request transactions or the item `VALUE`. This returns a string with a printed representation of the value returned by the last expression in the most recent execute transaction of the conversation.

As a client, there are three functions you can use that correspond fairly closely to their DDEML equivalents: `dde-connect`, `dde-disconnect`, and `dde-client-transaction`.

`dde-connect` takes two arguments, strings naming a service and a topic. The topic string is optional; it defaults to the service string. The return value is a descriptor of the conversation if the connection is established, otherwise it is `nil`. At the moment conversation descriptors are integer indices into a table, and the number of concurrent conversation is limited to 30. This may change.

`dde-disconnect` takes a conversation descriptor and attempts to close the conversation. If successful it returns `t`, otherwise `nil`. If dde-disconnect is called with no arguments then all currently active conversations are terminated. In this case the return value is `t` if any are terminated, `nil` if not.

`dde-client-transaction` requires a conversation descriptor as its first argument. The remaining arguments are keyword arguments:

`:type`: should be `:request` or `:execute`; default is `:execute`.

`:data`: a string, currently only used by execute transactions.

`:item`: an item name string, currently only used by request transactions.

`:timeout`: a positive integer specifying the timeout in milliseconds. The default is 60000.

The return value is `t` if the transaction is successful, `nil` if not.

As an example, you could have WXLS communicate with WXLS by DDE (why you would want to I do not know, but it works):

```
> (dde-connect "xlisp-stat")
0
> (dde-client-transaction 0 :data "(+ 1 2)")
T
> (dde-client-transaction 0 :type :request :item "value")
"3"
> (dde-disconnect 0)
T
```

You can also communicate with the program manager:

```
> (dde-connect "progman")
0
> (dde-client-transaction 0 :data "[ShowGroup(Main,1)]")
T
> (dde-client-transaction 0 :type :request :item "Groups")
"Main\r
Accessories\r
Applications\r
Microsoft C/C++ 7.0\r
Software Development Kit 3.1\r
Win32 Applications\r
Adobe\r
Borland C++ 4.0 Online Books\r
Borland C++ 4.0\r
StartUp\r
Games\r
Borland C++ 3.1\r
Pocket Tools\r
XLISP-STAT Programs\r
"
> (dde-disconnect 0)
T
```

The first transaction is an execute transaction that opens the Main group's window. The second is a request that obtains a list of the current groups.

> *This DDE interface is experimental and may change. For the moment it seems adequate for providing configuring the integration of WXLS into Windows during setup.*

The functions `msw-get-profile-string` and `msw-write-profile-string` can be used to access and modify user preference information. They require three and four arguments, respectively. The first and second arguments specify the section and item names as strings, and the last argument specifies the preference file name. A preference file name of `nil` refers to the system preference file. For `msw-write-profile-string` the third argument is a new value. This can be a string or `nil`; if it is `nil`, the entry is deleted. This function deletes a section if the item argument is `nil`.

The function `msw-win-exec` is a synonym for the `system` function. On failure it now returns two values: `nil` and a numerical error code. The `msw-win-help` function has been modified to use keyword symbols to identify help request types.

Both 16-bit and 32-bit versions can use DLL's, but only 16-bit and 32-bit DLL's, respectively.

4 Improvements in Common Lisp Support

This section outlines changes in Common Lisp compatibility. The subsections are numbered according to the chapters of Steele [7]. Detailed descriptions of standard functions and macros are given in Steele [7].

4.1 Introduction

No changes.

4.2 Data Types

Vectors and arrays containing typed elements, such as fixnums or floats, are now partially supported. This support will be completed in a future release.

Other new data types are hash tables and packages.

4.3 Scope and Extent

The handling of `block/return-from` and `tagbody/go` has been fixed to give the block names and go tags lexical scope, in compliance with the Common Lisp specification. In the previous implementation they had dynamic scope.

4.4 Type Specifiers

The `typep` function accepts specialized type specifications, such as `(member 1 2 3)` or `(vector t *)`. New type names can be defined using the `deftype` macro.

The function `coerce` has been modified to be more Common Lisp compliant. In particular, it is no longer possible to use it to coerce an array to a list. The effect of the old coercion can be achieved with the new `array-to-nested-list` function.

4.5 Program Structure

Special variables are now available. Variables that have been proclaimed special using the `proclaim` function or that were defined using `defvar`, `defparameter`, or `defconstant` are dynamically scoped in all uses. It is not possible to declare variables special locally, but `progv` can be used for making variables locally special.

The macro `defvar` now leaves unbound variable unbound if it is not given a value argument.

Proper keyword argument handling is now possible; see Section 2.1 for more details.

The special form `eval-when` for controlling time of evaluation is now defined. It is defined in accordance with the revised definition of Steele [7].

4.6 Predicates

The `functionp` now follows the new specification in Steele [7]. Only internal functions (`SUBR`'s), byte compiled functions, and function closures result in a non-`nil` value. In particular, `nil` is returned for symbols and lambda expressions.

The function `bit-vector-p` has been added for compatibility with some software; it always returns `nil`.

The macros `and` and `or` now return multiple values if the final expression does.

4.7 Control Structure

As mentioned in Section 4.3, block names and go targets are now properly lexically scoped.

A number of functions, macros and special forms have been modified to return multiple values when appropriate. These are

apply	block	case	catch	cond	do	do*	dolist
dotimes	flet	funcall	if	labels	let	let*	loop
macrolet	prog	prog*	progn	progv	unless	unwind-protect	when

Functions can now be defined to return multiple values using the `values` function. If a function is defined as

```
(defun f (x y) (values x y))
```

then it returns two values:

```
> (f 1 2)
1
2
```

Values beyond the first are ignored when the value of a function is passed as an argument:

```
> (list (f 1 2))
(1)
```

It is also possible to return no values; the implicit first value when used as a function argument is `nil`:

```
> (values)
> (list (values))
(NIL)
```

Returning no values is one way to suppress printing if no meaningful value is returned; for example, the `pprint` function returns no values.

Several macros and special forms are available for capturing multiple values. You can collect them in a list with `multiple-value-list`,

```
> (multiple-value-list (f 1 2))
(1 2)
```

access one specific value by position with `nth-value`,

```
> (nth-value 0 (f 1 2))
1
> (nth-value 1 (f 1 2))
2
> (nth-value 2 (f 1 2))
NIL
```

bind them to variables with `multiple-value-bind`,

```
> (multiple-value-bind (a b) (f 1 2) (list b a))
(2 1)
```

or pass them as arguments to a function with `multiple-value-call`,

```
> (multiple-value-call #'+ (f 1 2))
3
```

Multiple values can also be produced with `values-list` and captured with `multiple-value-prog1` and `multiple-value-setq`.

The assignment macros `psetf` and `rotatef` and the macro `typecase` are now available.

The assignment macro `setf` has been re-implemented as a macro, both in the interpreter and a compiler. The function `get-setf-method` has been added and is used both by `setf` and by new versions of macros like `push` and `incf`. The lmacsetf macro should now work with place forms that use `apply`.

The standard macro version of `setf` introduces a number of temporary variables to avoid multiple evaluation of forms in a setf expression. This can slow down interpreted code (it does not affect compiled code since unnecessary bindings are optimized out). To reduce the impact of this, the special variable `*simplify-setf*` can be set to a nonnil value. When this variable is not `nil`, `setf` substitutes the value expressions for these variables. In principle this can cause multiple evaluation of some subform, but it will not for any of the standard setf methods. This behavior is equivalent to the behavior of the previous internal version of `setf`. The setault value of `*simplify-setf*` in the interpreter it `t`. The compiler should set it to `nil`, but does not do so yet.

The function `special-form-p` has been added. It returns `t` if its symbol argument has a global function binding that is of type `FSUBR`; otherwise it returns `nil`.

4.8 Macros

Macros can now be defined to use the `&environment`, `&whole` and `&body` lambda list keywords. The functions `macroexpand` and `macroexpand-1` accept an optional environment argument, which should only be an environment captured with an `&environment` argument to a macro or an environment passed to an evalhook function. Expansion is done using this environment if supplied; otherwise, the null environment is used.

Macros allow destructuring in their required arguments, not in any other arguments. The macro `destructuring-bind` is also available. The `&whole` lambda keyword may not be used in this macro. Steele [7] says it may, but I think this is an error.

The `define-compiler-macro` macro is available for defining macros to be expanded only at compile time.

The function `macro-function` is available. Macros have been modified to use a macro function of two arguments, the form to be expanded and the environment. `macro-function` returns this function when its symbol argument names a macro.

The functions `variable-information` and `function-information` for accessing environment information and `augment-environment` for changing environment information have been added. The functions `parse-macro` and `enclose` are also available.

A simple implementation of `symbol-macrolet` has been added. It has not been extensively tested. It is currently not part of the standard initial workspace but set up to be autoloaded when called. Code using `symbol-macrolet` will be completely macro expanded, which means some standard macros implemented as special forms in the interpreter will be replaced by macros and expanded. The code may therefore be a bit slow when interpreted, but speed of compiled code will not suffer.

4.9 Declarations

The special form `declare` is available, but all declarations are currently ignored by the interpreter and the compiler, including special declarations.

> *It may be fairly easy to add proper handling of special declarations to the byte code compiler, but adding it to the interpreter will cut speed unless the code is rewritten in the spirit of macro displacement.*

The function `proclaim` is available. Currently only special proclamations have an effect. There is no portable way to take back a special proclamation.

Macros for the special forms `locally` and `the` are available. They currently ignore declaration and type information.

4.10 Symbols

The functions `keywordp`, `gentemp`, and `get-properties` have been added. The `getf` function has been fixed to search for property identifiers only in even positions in a property list, and the `(setf getf)` form has been added. The macro `remf` has also been added.

The function `gensym` has not yet been changed to follow the new specification, but probably will be.

4.11 Packages

XLISP now uses the Common Lisp package system for name space management. The functions `apropos`, `apropos-list`, `intern`, and `unintern` have been modified accordingly.

A package is a collection of symbols, with some symbols considered internal and others external, or exported. The system is always "in" some package, the current package, which is the value of the variable `*package*`. Packages can "use" other packages. When in a package, all symbols of that package, internal or external, and all external symbols of other packages used by that package are considered accessible. Accessible symbols can be referenced by their name and are printed using only their name. Inaccessible symbols can still be referenced and printed using a package name qualifier and a colon, `:`, or double colon, `::` separator. An external symbol in a package is referenced or printed as

<package>`:`*<name>*

and an internal symbol as

<package>`::`*<name>*

Thus `foo:bar` is the external symbol with name `"BAR"` in the package named `"FOO"`, and `foo::baz` is the internal symbol with name `"BAZ"` in the `"FOO"` package.

The default package is the package named `"USER"`. This package has several nicknames that can be used to refer to it:

```
> (package-nicknames "USER")
("CL-USER" "COMMON-LISP-USER")
```

It uses the `"XLISP"` package,

```
(package-use-list "USER")
(#<Package XLISP>)
> (package-nicknames "XLISP")
("CL" "COMMON-LISP" "LISP")
```

Another standard package is the `"KEYWORD"` package. Keywords are placed in this package as internal symbols and are made constants with values equal to themselves. Symbols in this package are always referenced and printed as a colon followed by the symbol name. So `:test` is the symbol with name `"TEST"` in the keyword package.

Packages are usually constructed using the `defpackage` macro. The current package is usually set with the `in-package` macro. Both typically appear in a file, followed by some export commands. For example, a file containing the lines

```
(defpackage "MYPACK" (:use "XLISP"))

(in-package "MYPACK")
```

```
(export '(a b))

(defun a () ...)
(defun b () ...)
(defun c () ...)
(defun d () ...)
```

constructs a new package `"MYPACK"` with external symbols `a` and `b` and internal symbols `c` and `d`. It is not necessary to save and restore the old package since the `load` function restores the current package to the value it had before the load.

When a package other than the `"USER"` package is the current package, the standard listener top level loop adds the package name to the prompt:

```
> (in-package "XLISP"")
#<Package XLISP>
XLISP> (in-package "USER")
#<Package USER>
>
```

Other functions and macros to support the use of packages are

`delete-package`	`do-all-symbols`	`do-external-symbols`	`do-symbols`
`find-all-symbols`	`find-package`	`find-symbol`	`import`
`list-all-packages`	`make-package`	`package-name`	`package-shadowing-symbols`
`package-used-by-list`	`package-valid-p`	`packagep`	`rename-package`
`shadow`	`shadowing-import`	`symbol-package`	`unexport`
`unuse-package`	`use-package`		

> *Details of the assignment of system and statistical symbols to packages and of package naming are likely to change in the near future and should not be relied upon.*

4.12 Numbers

The functions `ceiling`, `floor`, `round`, and `truncate` now return two values in accordance with the Common Lisp specification. The second value is the remainder of the operation.

> *The vectorized version only returns one value for a compound argument. This is consistent with the idea that the vectorization can be defined with a mapping function. It is not clear whether this is the right decision, and it may change.*

The functions `make-random-state` and `random-state-p` have been changed to use new random number generators; see Section 5.1

The following vectorized arithmetic functions have been added:

`ash` `asinh` `atanh` `cosh` `cis` `lcm` `logtest` `signum` `sinh` `tanh`

The macros `decf` and `incf` have also been added.

4.13 Characters

The function `alpha-char-p` has been added.

4.14 Sequences

Sequence functions have been improved and expanded. The following functions have been modified to operate on lists and all vector types, including strings:

`concatenate`	`copy-seq`	`count`	`elt`	`find`	`every`	`map`	`notany`
`notevery`	`some`	`position`	`reduce`	`remove-duplicates`	`subseq`		

The following functions have been added:

`count-if`	`count-if-not`	`delete-duplicates`	`fill`	`find-if`	`find-if-not`
`map-into`	`nreverse`	`position-if`	`position-if-not`	`replace`	`search`

Most keyword arguments specified in Steele [7] are supported, except that not all functions that should support the `:from-end` keyword yet.

The `complement` function for negating a predicate has been added.

4.15 Lists

The functions `mapcan` and `mapcon` are now functions instead of macros.

The following list functions have been added:

`copy-alist`	`copy-tree`	`list-length`	`list*`	`nintersection`
`nreconc`	`nset-difference`	`nset-exclusive-or`	`nsublis`	`nsubst`
`nsubst-if`	`nsubst-if-not`	`nunion`	`pairlis`	`set-exclusive-or`
`eighth`	`fifth`	`ninth`	`seventh`	`sixth`
`tenth`				

Setf forms for the positional accessors `fifth`, ..., `tenth` have also been added.

The macro `pop` is now available.

4.16 Hash Tables

Hash tables are now provided, with the interface functions

`clrhash`	`gethash`	`hash-table-count`	`hash-table-p`
`hash-table-rehash-size`	`hash-table-rehash-threshold`	`hash-table-size`	`hash-table-test`
`make-hash-table`	`maphash`	`remhash`	

Any test predicate is allowed, but the standard ones, `eq`, `eql`, and `equal` have internal implementations that should make them reasonably fast.

4.17 Arrays

Arrays can now be restricted to certain element types. This allows for more compact storage of arrays of specialized types. The standard element types supported are

`character`   `fixnum`   `float`   `(complex fixnum)`   `(complex float)`

In addition, the following nonstandard types are supported for communicating with C programs:

`c-char`	`c-short`	`c-int`	`c-long`	`c-float`	`c-double`
`c-complex`	`c-dcomplex`				

Other types are considered equivalent to `t`.

The `array-element-type` function returns the element type of an array, The `:element-type` keyword can be used with `make-array` to construct an array of the specified type.

> *The support for typed vectors is not yet complete. It is more or less complete for the basic XLISP system, but has not yet been completely integrated in the statistical code. Eventually the linear algebra routines and the foreign function interface will be changed to rely on these routines in order to minimize conversion to and from C data. The details of the supported types may need to be changed as this evolves.*

The functions `row-major-aref` and `svref` have been added.

> *At present, vectors with fill pointers and adjustable arrays are not supported. One or both may be added.*

4.18 Strings

The functions `nstring-capitalize`, `schar`, `string-capitalize`, and `string-search` have been added.

4.19 Structures

The `defstruct` macro allows constructor, predicate and print functions to be specified. Inheritance of other structures through the `include` keyword is supported, and produces proper subtype relationships. BOA constructors are not supported.

4.20 The Evaluator

The `applyhook` function and the `*applyhook*` variable are now available. Both `eval` and `evalhook` now use the null lexical environment for evaluation.

4.21 Streams

The `force-output` function now allows arbitrary stream arguments. Simple versions of `clear-output` and `finish-output` are now available. The functions `fresh-line` and the corresponding format directive are now available. The functions `input-stream-p`, `open-stream-p`, `output-stream-p`, and the macro `with-open-stream` have been added.

4.22 Input/Output

Printing of floating point numbers by the `print` function now follows the Common Lisp standard. This means floating point numbers always contain a decimal point and are generally printed with around 18 digits on a system with 64-bit double precision floating point numbers. This insures that printed numbers can be read back in to produce essentially identical values (slightly more digits would be needed to insure absolute equality).

> *This change may produce output that some find to unreadable. It would be good to be able to control the number of digits used for floating point printing. At present there is a back door mechanism: the variable* `*float-format*` *can be set to a C format string, which will then be used. The old printing approach is equivalent to setting this variable to* `"%g"`. *I do not know how to implement such an option portably in Common Lisp and I am not sure how important it is, since all model summaries and things like* `print-matrix` *use* `format`. *If this turns out to be useful and important, I will change the mechanism to use a Common Lisp format string instead of a C one.*

Several new print and read control variables are now used. These are

`*print-array*`	`*print-case*`	`*print-circle*`
`*print-escape*`	`*print-gensym*`	`*print-length*`
`*print-level*`	`*print-readably*`	`*read-suppress*`

Two nonstandard variables are also used. If `*print-symbol-package*` is non-`nil`, all symbols are printed with package qualifiers. The variable `*readtable-case*` can be used to set the case of the readtable.

> *The readtable case should be part of the readtable; this will probably be changed to comply with Common Lisp.*

The functions `read`, `read-char`, `read-byte`, and `read-line` now support `eof-error-p` arguments.

The reader now uses the keyword package as the default package while processing a features expression specified with the `#+` or `#-` read macros. This means that `#+fred` and `#+:fred` are equivalent. The standard symbols in the `*features*` list have been changed to keyword symbols.

The `format` function has been changed to handle the `~E`, `~F`, and `~G` directives in accordance with the Common Lisp specification. A number of format directives have been added. The new format directives are

`~O`, `~X`	octal and hexadecimal output
`~&`	fresh line
`~T`	tabulate
`~(`, `~)`	case conversion
`~*`	argument skipping
`~?`	indirection
`~[`, `~;`, `~]`	conditional expression
`~{`, `~}`	iteration
`~|`	page separator

Very minimal implementations of the following functions have been added:

`write`	`write-line`	`write-string`	`write-to-string`
`parse-integer`	`prin1-to-string`	`princ-to-string`	`pprint`
`read-from-string`			

4.23 File System interface

The functions `delete-file`, `file-length`, `probe-file`, `truename`, `rename-file`, and `file-write-date` have been added.

There is no separate pathname type; pathnames are currently just strings. Minimal versions of the following pathname functions have been added:

`make-pathname`	`merge-pathnames`	`namestring`	`parse-namestring`
`pathname`	`pathname-device`	`pathname-directory`	`pathname-host`
`pathname-name`	`pathname-type`	`pathname-version`	

The variable `*default-pathname-defaults*` is used for default values.

4.24 Errors

Error handling now uses the condition system; see Section 4.29 for details.

4.25 Miscellaneous Features

The `step` function has been replaced by a new implementation. Details are given in Appendix A.

The standard function `function-lambda-expression` replaces `get-lambda-expression`.

A very rudimentary `describe` functions is available; it will be improved in the future.

A byte code compiler for XLISP is included in Release 3. For code that spend much time in tight loops, the compiler can lead to significant speed improvements.

The interface to the compiler is through two functions, `compile` and `compile-file`. Suppose `f` is defined as

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

Then

```
> #'f
#<Closure-F: #c6741c>
> (compile 'f)
F
> #'f
#<Byte-Code-Closure: #c73644>
```

`compile` can also be used to compile a lambda expression:

```
> (compile nil '(lambda (x) (+ x 1)))
#<Byte-Code-Closure: #d77350>
```

The function `compile-file` takes a file name string as its required argument. If the file has no extension, a `.lsp` extension is added. It compiles the file into a file with a `.fsl` extension. When `load` is given a string `"fred"` as its argument, it first looks for `"fred.fsl"` and then for `"fred.lsp"`. If both are present, the `.fsl` files is used if it is newer than the `.lsp` file; otherwise the `.lsp` file is used.

Currently the compiler ignores all declarations, including special declarations, and all proclamations other than special ones. Future versions will use inline and optimize declarations to choose among code generation strategies.

The `.fsl` files produced by the compiler contain ordinary lisp expressions that are read in by the reader. Constants are printed out with `*print-circle*` and `*print-readably*` turned on. This should be sufficient, but if things get confused by a lot of messing with packages, it may help to also turn on `*print-symbol-package*`. This can be done by supplying the `:print-symbol-package` keyword argument to `compile-file` with a non-`nil` value.

The function `compiled-function-p` returns true for internal compiled functions (`SUBR`'s) or byte compiled functions.

The compiler is based on CPS conversion (see, for example, Friedman, Wand and Haynes [3]). The design is based on the *ORBIT* compiler as described in Krantz et al. [4] and on Brooks, Gabriel and Steele [2].

At this point the compiler does not do anything special for vectorized arithmetic or anything else statistical. In the future I will explore adding optimizations designed to deal with problems specific to statistical usage. The basic design should make this reasonably easy.

4.26 Loop

No changes – not implemented. The subset of loop from Peter Norvig's [6] book should work with at most minor modifications.

4.27 Pretty Print

No changes – not implemented. The XP pretty pringing package from the CMU lisp archives can be made to work with minor modifications.

4.28 CLOS

No changes – not implemented. The closette subset of CLOS from the AMOP book can be made to work with minor modifications.

4.29 Conditions

XLISP-STAT now uses an implementation of the Common Lisp condition system for error handling. The functions `error`, `cerror`, `warn`, and `signal` signal errors, continuable errors, warnings, or general conditions, respectively.

The macro `ignore-errors` takes an expression argument and returns either the values of that expression in the current environment if there is no error, or the values `nil` and the error condition signaled:

```
> (ignore-errors (values 1 2))
1
2
> (ignore-errors (error ``an error''))
NIL
#<Condition SIMPLE-ERROR: 13023072>
```

The macros `assert` and `check-type` for type and predicate checking and the macros `ccase`, `ctypecase` signal continuable errors. The macros `check-type`, `ccase`, `ctypecase`, `ecase`, and `etypecase` signal errors of type `type-error`.

At this point the implementations of these macros are bare bones and may not actually provide continuable errors or the right error type but that should be changed soon.

Conditions handlers are set up and used with the functions and macros

`define-condition`	`make-condition`	`handler-bind`
`handler-case`	`with-condition-restarts`	

Restarts can be manipulated using the following functions and macros:

`compute-restarts`	`find-restart`	`invoke-restart`	`invoke-restart-interactively`
`muffle-warning`	`restart-bind`	`restart-case`	`restart-name`
`store-value`	`use-value`	`with-simple-restart`	

The function `invoke-debugger` provides a low level interface to the debugger.
All standard predefined condition types are implemented, including the accessor functions

`cell-error-name`	`simple-condition-format-arguments`
`simple-condition-format-string`	`type-error-datum`
`type-error-expected-type`	

Conditions are currently implemented as structures and therefore do not support multiple inheritance.

For the most part internal errors still signal errors of type `simple-error`*. This will be changed eventually. One exception is unbound variable and unbound function errors – these are already of types* `unbound-variable` *and* `unbound-function`*, respectively. This may eventually be used to define a more sophisticated autoload facility.*

Currently stack overflow errors are not signalled at the system state where they occur, because trying to handle them without any stack space would lead to an infinite error recursion. Instead they are signaled from the most recent `handler-bind`*, in such a way as to insure that a stack overflow in a handler is caught in the next most recent one.*

In the future, I will try to generate low stack errors that allow a limited amount of exploring the system state before a real overflow occurs. I think this can be done fairly easily without a performance penalty, but I'm not sure yet.

5 Changes in the Statistical System

5.1 New Random Number Generators

Three new generators in addition to the original lagged Fibonacci generator are now available. The generators are identified by an integer:

0 The original XLISP-STAT generator, Marsaglia's portable generator from CMLIB. This is a lagged Fibonacci generator.

1 L'Ecuyer's [5] version of the Wichmann-Hill [9] generator, also used in Bratley, Fox and Schrage, [1, program UNIFL].

2 Marsaglia's Super-Duper, as used in *S*.

3 Combined Tausworthe generator of Tezuka and L'Ecuyer [8].

The default generator is generator 1. Generator 0 has a period of 2^{32}. All three new generators have periods on the order of 2^{60}.

Random states are now printed as

```
> *random-state*
#$(1 #(2147483562 0 11716 54063))
```

The function make-random-state produces a new seed from the current generator when called with the argument t. An alternate generator can be specified by supplying an appropriate integer as a second argument:

```
> (make-random-state t)
#$(1 #(2147483562 0 11716 54088))
> (make-random-state t 2)
#$(2 #(2147483647 0 0 11715 0 54105))
```

When make-random-state is called with an old state vector, a new state for generator 0 is returned.

5.2 Changes in the Object System

The object system now uses a method cache that should significantly improve dispatch for methods in deep object hierarchies.

To help with compilation, the way in which the current object is passed to functions slot-value, call-next-method and call-next-method has changed. These functions should only be called in the dynamic extent of a method call. Their use outside of this scope, for example in a function closure returned by a method, is undefined.

> *The current implementation uses a dynamic binding to store the current object. This works, but makes tail recursion optimization on methods impossible. I may need to make a major change in which things like* slot-value *and* call-next-method *become macros. The main implication of this is that* (apply #'call-next-method ...) *constructs won't work and will have to be handled by something like an* apply-next-method *macro. Since this is likely to break a fair bit of code I will think about it before doing it, but at this point a change along these lines look fairly likely.*
>
> *A more substantial revision of the object system may occur soon in which the current system remains more or less unchanged but becomes just one possible system among many supported by a metaobject protocol. This should allow experimentation with variations in the object system that might be helpful.*

5.2.1 Changes in the Linear Algera System

Substantial internal changes have been mad in the linear algebra system. These should result in improved performance in fitting large regression and generalized linear models.

The overall plan is to eliminate allocation of data structures at the internal C level. Instead, data structures for efficient linear algebra computations are allocated at the Lisp level as typed arrays. These are operated on by low level functions, which in many cases are simple front ends to LINPACK or BLAS routines. Standard functions, such as qr-decomp are built as Lisp-level wrappers around these lower level routines. Once the lower level is cleaned up it will be documented so that users have the option of managing their own allocation and directly using the low level routines.

5.3 Other Changes

The generalized linear model system is now part of the standard distribution. The :display methods for all model objects and the print-matrix function have been changed to use the new ~G format implementation.

> *At the moment the number of digits printed is fixed at 6. I may make this a user-settable option.*

The function eigen is now based on the EISPACK rs routine.

The function reset-system can be used to reset the state of various internal system parameters. If you write your own top level loop, you can call this when your loop is restarted if it looks like it is needed.

> *The need for this function is not quite clear yet – it may be dropped.*

The function system-has-windows returns non-nil if a window system is available at runtime. This should be used on UNIX systems rather than a readtime method, such as #+windows.

6 Deleted Functions

The function `load-help` and the *X11* support function `make-fake-menu-bar` are no longer generally available. Their symbols are internal to the XLISP package.

> *The assignment of internal symbols to packages is likely to change in the near future and should not be relied upon.*

The functions `get-lambda-expression`, `num-to-string`, and `strcat` have been removed since similar Common Lisp functions are available. You should use `function-lambda-expression`, `prin1-to-string` and `concatenate` instead.

A The Step Function

> *This is a slightly modified version of the file* `stepper.doc` *from the XLISP-PLUS 2.1g distribution.*

The new step debugger, written by Ray Comas (`comas@math.lsa.umich.edu`) and modified by Tom Almy, was inspired by the `step.lsp` stepper included with XLISP 2.1, originally written by Jonathan Engdahl (`jengdahl` on *BIX*). This version has the ability to set/reset breakpoints, and a few bells and whistles.

To invoke the stepper:

```
(step (form with args))
```

The stepper will stop and print every form, then wait for user input. Forms are printed compressed, i.e. to a depth and length of 3. This may be changed by assigning the desired depth and length values to `*stepper-depth*` and `*stepper-length*` before invoking the stepper, or from within the stepper via the `.` and `#` commands.

For example, suppose you have the following defined:

```
(defun fib (n)
  (if (or (eql n 1) (eql n 2))
      1
      (+ (fib (- n 2)) (fib (- n 1)))))
```

Then `(step (fib 4))` will produce the following:

```
0 >==> (fib 4)
 1 >==> (if (or (eql n 1) (eql n 2)) 1 ...) :
```

The colon is the stepper's prompt. For a list of commands, type `h`. this produces:

```
Stepper Commands
----------------
 n or space - next form
 s or <cr>  - step over form
 f FUNCTION - go until FUNCTION is called
 b FUNCTION - set breakpoint at FUNCTION
 b <list>   - set breakpoint at each function in list
 c FUNCTION - clear breakpoint at FUNCTION
 c <list>   - clear breakpoint at each function in list
 c *all*    - clear all breakpoints
          g - go until a breakpoint is reached
          u - go up; continue until enclosing form is done
          w - where am I? -- backtrace
```

```
        t - toggle trace on/off
        q - quit stepper, continue execution
        p - pretty-print current form (uncompressed)
        e - print environment
 x <expr> - execute expression in current environment
 r <expr> - execute and return expression
     # nn - set print depth to nn
     . nn - set print length to nn
        h - print this summary
```

Breakpoints may be set with the b command. You may set breakpoints at one function, e.g. b FOO<cr> sets a breakpoint at the function FOO, or at various functions at once, e.g. b (FOO FIE FUM)<cr> sets breakpoints at the functions FOO, FIE, and FUM. Breakpoints are cleared with the c command in an analogous way. Furthermore, a special form of the c command, c *all* <cr>, clears all previously set breakpoints. Breakpoints are remembered from one invocation of step to the next, so it is only necessary to set them once in a debugging session.

The g command causes execution to proceed until a breakpoint is reached, at which time more stepper commands can be entered.

The f command sets a temporary breakpoint at one function, and causes execution to proceed until that function is called.

The u command continues execution until the form enclosing the current form is done, then re-enters the stepper.

The w command prints a back trace.

The q command quits and causes execution to continue uninterrupted.

Entry and exit to functions are traced after a g, f, u, or q command. To turn off tracing, use the t command which toggles the trace on/off. Also, with trace off, the values of function parameters are not printed.

The s command causes the current form to be evaluated.

The n command steps into the current form.

The . and # commands change the compression of displayed forms. E.g. in the previous example:

```
1 >==> (if (or (eql n 1) (eql n 2)) 1 ...) : . 2
1 >==> (if (or (eql n ...) ...) ...) :
```

changes the print length to 2, and

```
1 >==> (if (or (eql n ...) ...) ...) : # 2
1 >==> (if (or #\# ...) ...) :
```

changes the print depth to 2.

To print the entire form use the p command, which pretty-prints the entire form.

The e command causes the current environment to be printed;

The x command causes an expression to be executed in the current environment. Note that this permits the user to alter values while the program is running, and may affect execution of the program.

The r command causes the value of the given expression to be returned, i.e. makes it the return value of the current form.

> *The stepper will not produce proper printout for* go *if the jump is outside the most enclosing* tagbody*, and the tag arguments of* catch/throw *must either be symbols or quoted symbols. No attempt is made here to correctly handle tracing of* unwind-protect*, either.*

References

[1] Bratley, P., Fox, B. L., and Schrage, L. E. (1987), *A Guide to Simulation* (2nd ed.), New York, NY: Springer-Verlag.

[2] Brooks, R. A., Gabriel, R. P, and Steele, G. L. (1982), "An optimizing compiler for lexically scoped LISP," *Proc. Symp. on Compiler Construction, ACM SIGPLAN Notices* 17, 6, 261–275.

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

[4] Krantz, D. A., Kelsey, R., Rees, J. A., Hudak, P., Philbin, J., and Adams, N. I. (1986), "Orbit: An optimizing compiler for Scheme," *Proc. SIGPLAN '86 Symp. on Compiler Construction, SIGPLAN Notices* 21, 7, 219–223.

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

[6] Norvig, P. (1992), *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, San Mateo, CA: Morgan Kaufmann.

[7] Steele, Guy L. (1990), *Common Lisp: The Language*, second edition, Bedford, MA: Digital Press.

[8] Tezuka, S. and L'Ecuyer, P. (1991), "Efficient and portable combined Tauseworthe random number generators," *ACM Transactions on Modeling and Computer Simulation* 1, 99-112.

[9] Wichmann, B. A. and Hill, I. D. (1982) "An efficient and portable pseudo-random number generator," (Corr: V33 p123), *Applied Statistics* 31, 188–190.

Index