A New Autoload System for XLISP-STAT

By
Luke Tierney [1]
Technical Report No. 623
School of Statistics
University of Minnesota
December 29, 1997

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

1 Introduction

An autoloading system allows infrequently used data or procedures to be stored on disk until they are needed. At that time they are automatically loaded without requiring user intervention. This report describes a new autoloading system for XLISP-STAT [4]. It also introduces an enhancement to the `require` function that allows a search path for `require` to be specified.

The new autoloading system and the modified `require` function are included in the current development snapshot[1] and will be part of the next release of XLISP-STAT.

This report is a literate literate program[1]. The file used to typeset this report also contains the source code. The `noweb` literate programming system [3, 2] was used to produce the manuscript and the source files.

2 The New System

Under the previous autoloading system, when XLISP-STAT started a session it would execute the code in *<library>*`/Autoload/autoload.lsp`. This file defined some utility functions and then provided definitions for the symbols to be autoloaded. These definitions consisted of macro calls of the form

```
(autoload foo "bar")
```

This call would expand into (in simplified form)

```
(defun foo (&rest args)
  (load "bar")
  (apply foo args))
```

This approach has several drawbacks. It works for functions, could be modified to work for macros, but does not work for variables. Also, adding new code for autoloading requires editing the `autoload.lsp` file.

The new approach uses the `unbound-variable` and `undefined-function` errors signaled when a symbol's value or function cells are accessed and found to be unbound.[2] On startup, XLISP-STAT searches for files named `_autoidx.lsp` (or `_autoidx.fsl` if compiled, but there is no need to) in a specified set of directories and all its subdirectories and loads them. The default search path contains only the *<library>*`/Autoload` directory. These files should

- define any packages that are needed
- export symbols as needed
- register symbols to trigger autoloading when their value or function cells are accessed.

[1] See URL `http://www.stat.umn.edu/~luke/xls/projects/Snapshot`.

[2] The `undefined-function` error was previously incorrectly named `unbound-function`; this has been changed.

It may also be useful to include a provide call for the module.

A new macro, system:define-autoload-package[3] is provided for registering the function and value cells of symbols that are to trigger autoloading. The macro is called with a string naming the module and clauses listing the variables and functions/macros that are to trigger autoloading. For example, if an _autoidx.lsp file contains the expression

```
(system:define-autoload-module "foo"
  (function bar1 bar2)
  (variable baz))
```

then an attempt to access the function cells of bar1 and bar2 or the value cell of baz causes the file foo.lsp or foo.fsl to be loaded from the directory containing the index file.

An important point to note is that symbol references are still constructed by standard reader rules. Thus if a symbol is referenced as foo it will be looked up in the current package. If a symbol is referenced as bar:foo then the package bar must already exist and contain the exported symbol named foo, even if the function definition of the symbol is to be autoloaded. This is why index files must contain appropriate package definition and export commands.

Here are some examples. The autoload index for a regular expression library might contain

2a ⟨*autoloads for a regular expression library* 2a⟩≡

```
(defpackage "REGULAR-EXPRESSIONS"
  (:use "COMMON-LISP")
  (:nicknames "REGEXP"))

(in-package "REGEXP")

(export '(regexp regsub url-decode))

(system:define-autoload-module "regexp"
  (function regexp regsub url-decode))
```

The autoload specification for the glim module in the standard distribution is

2b ⟨*autoload specification for the glim module* 2b⟩≡ (8)

```
(in-package "USER")
(system:define-autoload-module "glim"
  (variable glim-link-proto identity-link log-link inverse-link sqrt-link
            power-link-proto logit-link probit-link cloglog-link glim-proto
            normalreg-proto poissonreg-proto binomialreg-proto gammareg-proto)
  (function normalreg-model poissonreg-model loglinreg-model binomialreg-model
            logitreg-model probitreg-model gammareg-model indicators
            cross-terms level-names cross-names))
```

The remainder of the autoload specifications for the standard autoloaded modules is given in the appendix.

[3]New system features will be placed in the SYSTEM package. At the moment, this is just a nickname for the XLISP package, but this is likely to change. Exported symbols from the system package should thus always be referenced with a system: prefix unless the current package explicitly uses the SYSTEM package.

The easiest way to register a new set of functions for autoloading is to add a subdirectory to *<library>*`/Autoload` that contains an appropriate `_autoidx.lsp` file. A more complex alternative is to redefine the function `system:create-autoload-path` to add a new directory to the search path. A third option is to directly call `system:register-autoloads` with a directory containing an index file, or subdirectories with index files, as argument. When a session is initialized, autoloading registration is handled by the expression

3a ⟨*register standard autoloads* 3a⟩≡
```
(mapc #'register-autoloads (create-autoload-path))
```

The `require` function plays a similar role to the autoloading process. It allows modules to specify additional modules they need if they are loaded. The first argument to `require` is a module name string that is looked up in the `*modules*` list. If the name is not registered in the list then the optional second argument specifies a file or a list of files to load. The default value for the second argument is the module name. The loading process searches for the specified files by merging the pathnames for the files with the path names in the variable `system:*module-path*`. This variable is initialized by

3b ⟨*initialize module search path* 3b⟩≡
```
(setf *module-path* (create-module-path))
```

The `system:create-module-path` function creates a path consisting of the current directory, the standard library directory and the `Examples` subdirectory of the standard library directory. You can change this definition in a `statinit.lsp` file or by redefining `create-module-path`. Assigning a new value to `*module-path*` and saving the workspace will not work since this variable is reset at session startup. This allows the library directory to be changed without requiring a new workspace to be built.

3 Implementation

3.1 The Autoload System

Autoloading is done by handling the `unbound-variable` and `undefined-function` errors. There are two possible approaches. One is to handle them at the bottom of the handler stack by redefining the default handler. This is less dependent on the details of the condition system, but it means `ignore-errors` will not allow autoloading to work in its body. The alternative is to handle these errors at the top of the handler stack by redefining the condition hook function. This is the approach I have used.

The new condition hook function is `autoload-condition-hook`.

3c ⟨*definition of* `autoload-condition-hook` 3c⟩≡ (5d)
```
(defun autoload-condition-hook (&rest args)
  (handler-bind
   ((⟨unbound variable handler clause 3d⟩
    ⟨undefined function handler clause 4a⟩)
   (apply #'condition-hook args)))
```

The handler clause for unbound variables is

3d ⟨*unbound variable handler clause* 3d⟩≡ (3c)

```
(unbound-variable #'(lambda (c)
                      (autoload-variable (cell-error-name c))
                      (apply #'condition-hook args)))
```

and the undefined function handler clause is

4a ⟨*undefined function handler clause* 4a⟩≡ (3c)

```
(undefined-function #'(lambda (c)
                        (autoload-function (cell-error-name c))
                        (apply #'condition-hook args)))
```

The calls of `condition-hook`, the standard condition hook function, in the handlers handle unbound variable or undefined function cases that are not resolved by autoloading. This code uses `handler-bind`, not `handler-case`, since the handlers have to be called inside the restart context established by the implicit `cerror` call that signaled the error.

The hook is installed by

4b ⟨*install the new condition hook* 4b⟩≡

```
(setf *condition-hook* 'autoload-condition-hook)
```

To load an undefined function, `autoload-function` looks up a module path in a database and finds the `continue` restart that should have been established by the implicit `cerror` that signaled the error. The `*load-verbose*` variable is bound to `NIL` to suppress loading messages. If the module path and the restart are found, then the file is loaded. If the symbol has a function definition after the load, then the restart is invoked. If any of these conditions fails, then `autoload-function` returns and normal error processing resumes.

4c ⟨*definition of* `autoload-function` 4c⟩≡ (5d)

```
(defun autoload-function (name)
  (let ((modpath (find-function-module-path name))
        (restart (find-restart 'continue))
        (*load-verbose* nil))
    (when (and modpath restart)
          (load modpath)
          (when (fboundp name)
                (invoke-restart restart)))))
```

Undefined variables are handled analogously by

4d ⟨*definition of* `autoload-variable` 4d⟩≡ (5d)

```
(defun autoload-variable (name)
  (let ((modpath (find-variable-module-path name))
        (restart (find-restart 'continue))
        (*load-verbose* nil))
    (when (and modpath restart)
          (load modpath)
          (when (boundp name)
                (invoke-restart restart)))))
```

The autoload database is maintained in two hash tables,

4e ⟨*autoload database* 4e⟩≡ (5d)

```
(let ((function-modules (make-hash-table))
      (variable-modules (make-hash-table)))
  (defun find-function-module-path (name)
    (gethash name function-modules))
  (defun find-variable-module-path (name)
    (gethash name variable-modules))
  (defun add-function-module (name module)
    (setf (gethash name function-modules) module))
  (defun add-variable-module (name module)
    (setf (gethash name variable-modules) module)))
```

The macro for installing symbols in this table is

5a ⟨*definition of* `define-autoload-module` 5a⟩≡ (5d)

```
(defmacro define-autoload-module (module &rest clauses)
  `(let* ((dir (pathname-directory *load-truename*))
          (mname (make-pathname :name ',module :directory dir))
          (clist ',clauses))
     (dolist (c clist)
       (ecase (first c)
         (variable (dolist (n (rest c)) (add-variable-module n mname)))
         (function (dolist (n (rest c)) (add-function-module n mname)))))))
```

The `register-autoloads` function recursively traverses the directory structure starting at the specified argument and reads in any index files it finds.

5b ⟨*definition of* `register-autoloads` 5b⟩≡ (5d)

```
(defun register-autoloads (dir)
  (let ((idx (merge-pathnames "_autoidx" dir))
        (dirlist (base-directory dir)))
    #+(or unix msdos) (setf dirlist (delete "." dirlist :test #'equal))
    #+(or unix msdos) (setf dirlist (delete ".." dirlist :test #'equal))
    (load idx :verbose nil :if-does-not-exist nil)
    (dolist (d dirlist)
      (let ((dpath (make-pathname :directory (list :relative d))))
        (register-autoloads (merge-pathnames dpath dir))))))
```

This function is called during system startup for each directory in the list returned by the function `create-autoload-path`. The default definition of this function produces a list that contains only only the `Autoload` subdirectory of the system library,

5c ⟨*definition of* `create-autoload-path` 5c⟩≡ (5d)

```
(defun create-autoload-path ()
  (list (merge-pathnames (make-pathname :directory '(:relative "Autoload"))
                         *default-path*)))
```

Currently this code[4] is included in `pathname.lsp`.

[4] See URL `http://www.stat.umn.edu/~luke/xls/projects/autoload/pathname.lsp.frag`.

5d ⟨*pathname.lsp code* 5d⟩≡
```
(in-package "SYSTEM")
(export '(define-autoload-module register-autoloads
          create-autoload-path))
```
⟨*definition of* autoload-condition-hook 3c⟩
⟨*definition of* autoload-function 4c⟩
⟨*definition of* autoload-variable 4d⟩
⟨*autoload database* 4e⟩
⟨*definition of* define-autoload-module 5a⟩
⟨*definition of* register-autoloads 5b⟩
⟨*definition of* create-autoload-path 5c⟩

3.2 Modified require Function

The modified require function uses the *module-path* variable in the system package to hold the module search path.

6a ⟨*definition of* *module-path* *variable* 6a⟩≡ (7)
```
(defvar *module-path* nil)
```

The default value of this variable is computed by create-module-path.

6b ⟨*definition of* create-module-path 6b⟩≡ (7)
```
(defun create-module-path ()
  (list (make-pathname :directory '(:relative))
        *default-path*
        (merge-pathnames (make-pathname :directory '(:relative "Examples"))
                         *default-path*)))
```

Given a pathname from the second argument to require (supplied or default), the function find-require-path searches the module path until it finds a file that matches the path, possibly after adding a .lsp or .fsl extension. The path returned does not have an added extension. If no file is found, NIL is returned.

6c ⟨*definition of* find-require-file 6c⟩≡ (7)
```
(defun find-require-file (path)
  (let ((type (pathname-type path)))
    (dolist (dir *module-path*)
      (let ((p (merge-pathnames path dir)))
        (when (or (probe-file p)
                  (and (not type)
                       (or (probe-file (merge-pathnames p ".lsp"))
                           (probe-file (merge-pathnames p ".fsl")))))
          (return p))))))
```

The require function uses find-require-file to locate the files to load. Loading is done by calling the load function on the path. This allows the standard load code to examine modification dates and determine whether a .lsp or a .fsl file should be loaded if both are present and the path does not specify an extension. If no file is found by searching the path, load is called with the original path argument and the :if-does-not-exist flag set to NIL. This is to maintain backwards compatibility with the previous definition of require.

6d ⟨*definition of* require 6d⟩≡ (7)

```
(defun require (name &optional (path (string name)))
  (let ((name (string name))
        (pathlist (if (listp path) path (list path))))
    (unless (member name *modules* :test #'equal)
            (dolist (pathname pathlist)
              (let ((rpath (find-require-file pathname)))
                (if rpath
                    (load rpath)
                  (load pathname :if-does-not-exist nil)))))))
```

This code[5] is included in `common.lsp` in place of the previous definition of `require`.

7 ⟨*common.lsp code* 7⟩≡

```
(export '(system::*module-path* system::create-module-path)
        "SYSTEM")
```

⟨*definition of* `*module-path*` *variable* 6a⟩
⟨*definition of* `require` 6d⟩
⟨*definition of* `find-require-file` 6c⟩
⟨*definition of* `create-module-path` 6b⟩

4 Discussion

At present the index files for autoloading need to be prepared manually. It should be possible to modify the `compile-file` top level to attempt to generate these files automatically. This can't be done perfectly, but it should be possible to handle most cases.

It would be useful to explore adding more features to the minimal module system that `require` and `provide` make available. One useful addition would be versioning, perhaps along the lines of the versioning system in Tcl 8.0 [5]. Integrating name space management and modules would also be useful, as would better support for separate compilation and syntax management. Some of the newer Scheme module systems need to be examined.

It might also be useful to allow search paths to be initialized from environment variables on systems where those make sense (i.e. UNIX and Windows).

References

[1] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[2] Norman Ramsey. Noweb home page.

[3] Norman Ramsey. Literate programming simplified. *IEEE Software*, 13(9):97–105, September 1994.

[5] See URL `http://www.stat.umn.edu/~luke/xls/projects/autoload/common.lsp.frag`.

[4] Luke Tierney. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics.* J. Wiley & Sons, New York, NY, 1990.

[5] Brent B. Welch. *Practical Programming in Tcl and Tk.* Prentice-Hall, Upper Saddle River, NJ, 2nd edition, 1997.

A Standard Autoloads

The file `_autoidx.lsp`[6] in the `Autoload` directory provides for autoloading of certain modules in the standard distribution.

8 ⟨*_autoidx.lsp* 8⟩≡

```
(in-package "USER")
(system:define-autoload-module "nonlin"
  (variable nreg-model-proto)
  (function nreg-model))

(in-package "USER")
(system:define-autoload-module "oneway"
  (variable oneway-model-proto)
  (function oneway-model))

(in-package "XLISP")
(export '(numgrad numhess newtonmax nelmeadmax))
(system:define-autoload-module "maximize"
  (function numgrad numhess newtonmax nelmeadmax))

(in-package "USER")
(system:define-autoload-module "bayes"
  (function bayes-model)
  (variable bayes-model-proto))

(in-package "XLISP")
(export 'step)
(system:define-autoload-module "stepper"
  (function step))

(in-package "XLISP")
(export '(compile compile-file))
(system:define-autoload-module "cmpload"
  (function compile compile-file))
```

⟨*autoload specification for the glim module* 2b⟩

```
(in-package "XLISP")
```

[6] See URL `http://www.stat.umn.edu/~luke/xls/projects/autoload/_autoidx.lsp`.

```
(export 'xlisp::symbol-macrolet "XLISP")
(system:define-autoload-module "symaclet"
  (function symbol-macrolet))
```

B Indices

Chunk Index

Identifier Index