AUTOLOAD MANUAL

Contents

[in package AUTOLOAD]

1 Links and Systems

Here is the official repository and the HTML documentation for the latest version.

- **[system]** "**autoload**"
  - *Version:* 0.1.0
  - *Description:* An ASDF autoloading facility. See Autoload Manual.
  - *Licence:* MIT, see COPYING.
  - *Author:* Gábor Melis
  - *Mailto:* mega@retes.hu
  - *Homepage:* https://github.com/melisgl/autoload

- *Bug tracker:* https://github.com/melisgl/autoload/issues
- *Source control:* GIT
- *Depends on:* closer-mop, mgl-pax-bootstrap, trivial-indent

- **[system] "autoload-doc"**
  - *Description:* Parts of the Autoload library that depend on `mgl-pax` are in this system to avoid the circular dependencies that would arise because `mgl-pax` depends on `autoload`. Note that `mgl-pax/navigate` and `mgl-pax/document` depend on this system, which renders most of this an implementation detail.
  - *Depends on:* autoload, dref, mgl-pax, named-readtables, pythonic-string-reader

2 Introduction

Autoload was factored out of PAX, so let's explore the motivation in that context. PAX is a large system with substantial dependencies, but what you actually need in deployment is tiny. The rest is for interactive use and documentation generation. By autoloading the optional parts, we can keep deployment size down and avoid annoyingly long compile times caused by code for unused features (and their transitive dependencies).

Users could load the relevant systems manually when they need them, but that would be quite disruptive and they would need to remember what system to load. Also, code whose dependencies may not be loaded needs to jump through hoops (e.g. `funcall` + `intern`) or rely on forward declarations. Autoload takes care of these issues: you can factor out code that isn't always necessary into some sub-asdf-system along with its dependencies. This removes the pressure to drop dependencies just to keep deployments lean and spares some users the long wait of compiling `ironclad`.

More abstractly, libraries often choose to limit dependencies, even if it means sacrificing features or duplicating code, to minimize

- compilation time,
- memory usage in deployment, and
- the risk of breakage through dependencies.

Autoload mitigates these issues by loading heavy dependencies on demand. The core idea is

```
(defmacro autoload (name asdf-system)
  `(defun ,name (&rest args)
     (asdf:load-system ,asdf-system)
     (apply ',name args)))
```

Suppose we have a library called `my-lib` that autoloads `my-lib/full`. In `my-lib`, we could use `autoload` as

```
(autoload foo "my-lib/full")
```

and have

```
(defun foo (x)
  "doc"
  (1+ x))
```

in `my-lib/full`.

However, manually keeping the loaddefs (e.g. the `autoload` form above) in sync with the definitions is fragile, so we introduce the `defun/auto` autodef to mark autoloaded functions in the `my-lib/full` system:

```
(defun/auto foo (x)
  "doc"
  (1+ x))
```

ASDF Integration

To generate loaddefs, we add a few lines to the system definitions:

```
(asdf:defsystem "my-lib"
  :defsystem-depends-on ("autoload")
  :class "autoload:autoload-system"
  :auto-depends-on ("my-lib/full")
  :auto-loaddefs "loaddefs.lisp"
  :components ((:file "loaddefs")
               ...))
```

```
(asdf:defsystem "my-lib/full"
  :defsystem-depends-on ("autoload")
  :class "autoload:autoload-system"
  :components (...))
```

Then, the loaddefs can be extracted:

```
(extract-loaddefs "my-lib")
=> ((autoload foo "my-lib/full" :arglist "(x)" :docstring "doc"))
```

This is implemented by loading the `:auto-depends-on` of `my-lib` and recording `defun/auto`s. `extract-loaddefs` is a low-level utility used by `record-loaddefs`, which writes its results to the system's `:auto-loaddefs`, `"loaddefs.lisp"` in the above example. So, all we need to do is call `record-loaddefs` to regenerate the loaddefs file:

```
(record-loaddefs "my-lib")
```

To prevent the loaddefs file from getting out of sync with the definitions, `asdf:test-system` calls `check-loaddefs` by default.

ASDF, and by extension Quicklisp, doesn't know about the declared `:auto-depends-on`, so `(ql:quickload "my-lib")` does not install the autoloaded dependencies. They can be installed manually with

```
(autodeps "my-lib" :installer #'ql:quickload)
```

If all the autoloaded dependencies are installed, one can eagerly load them to ensure that autoloading is not triggered later (e.g. in deployment):

```
(map nil #'asdf:load-system (autodeps "my-lib"))
```

Other Features

Autoloading is not only for Functions:

- Autoloading Classes at the time of their first instantiation is supported.
- Variables can be marked for early definition and have their initial values assigned if the initial value form provably doesn't have dependencies. If that's not the case, subject to platform support, the definition in the loaded system injects a global binding even in the presence of local bindings.
- Multiple Packages can have their final states and interdependencies reconstructed before loading their systems even if they were mutated operations like `import` and `export`.

3 Basics

- **[glossary-term] loaddef**

  A loaddef is a preliminary definition that serves as a stand-in until the fully-realized implementation is loaded. Accessing it may or may not load a system. See `loaddef-function-p`, `loaddef-class-p`, `loaddef-variable-p` and `loaddef-package-p`.

- **[glossary-term] autodef**

  An autodef (e.g. `(defun/auto name ...)`) performs the job of its plain counterpart (`defun`). In addition, it marks the definition (of `name` as a function) for Automatically Generating Loaddefs and, at the time of the first such autodef, it signals an `autoload-warning` if `name` has not been declared as a loaddef (has never been `loaddef-function-p`). See `defun/auto`, `defclass/auto`, `defvar/auto` and `defpackage/auto`.

3.1 Loading Systems

Function and class loaddefs trigger the loading of `asdf:systems`. Unlike normal ASDF dependencies (declared in `:depends-on`), autoload dependencies (which may be declared in `:auto-depends-on`) are allowed to be circular. The rules for loading are as follows.

1. It is an `autoload-error` if loading is triggered during compile time or during a `load` of either a source file or a compiled file. This is to prevent infinite autoload recursion.
2. It is an `autoload-error` if the system does not exist.
3. The system is loaded under `with-compilation-unit(0 1) :override t` and `with-standard-io-syntax` but with `*print-readably* nil`. Other non-portable measures may be taken to standardize the dynamic environment. Errors signalled during the load are not handled or resignalled by the Autoload library.

4. It is an `autoload-error` if the loaddef is not replaced by a normal definition or deleted by the loaded system, that is, when it remains a loaddef (e.g. in terms of `loaddef-function-p`).

3.2 Conditions

- **[condition]** **autoload-error** *error*

  Signalled for some failures during Loading Systems.

- **[condition]** **autoload-warning** *simple-warning*

  See `autoload`, autodef and `:auto-depends-on` for when this is signalled.

3.3 Functions

- **[macro]** **autoload** *name system-name &key (arglist nil) docstring*

  This is the loaddef for autodef `defun/auto`.

  Define a function stub with `name` and return `name`. The arguments are not evaluated. If `name` has an `fdefinition` and it is not `loaddef-function-p`, then this does nothing and returns `nil`.

  The stub first loads `system-name`, then it `apply`s the function `name` to the arguments originally provided to the stub.

  The stub is not defined at compile time, which matches the required semantics of `defun`. `name` is `declaim`ed with `ftype function` and `notinline`.

  - `arglist` will be installed as the stub's arglist if specified and it's supported on the platform (currently only SBCL). If `arglist` is a string, the effective value of `arglist` is read from it. If the read fails, an `autoload-warning` is signalled and processing continues as if `arglist` had not been provided.

    Arglists are for interactive purposes only. For example, they are shown by SLIME autodoc and returned by `dref:arglist`.

  - `docstring`, if non-`nil`, will be the stub's docstring. If `nil`, then a generic docstring that says what system it autoloads will be used.

  When `autoload` is macroexpanded during the compilation or loading of an `autoload-system`, it signals an `autoload-warning` if `system-name` is not among those declared in `:auto-depends-on`.

- **[function]** **loaddef-function-p** *name*

  See if an `autoload` for `name` was established, and since then it has not been redefined (e.g. with `defun/auto`, `defun`) or made `fmakunbound`.

- **[macro]** **defun/auto** *name lambda-list &body body*

  This is the autodef for the loaddef `autoload`.

Like `defun`, but also silence redefinition warnings. `name` may be of the form `(definer name)`. In that case, instead of `defun`, `definer` is used to establish the underlying function binding.

Loaddef: The corresponding loaddef is an `autoload` form. `extract-loaddefs` with `process-arglist t` installs `lambda-list` as the `arglist`. If `process-arglist` is `nil`, then `arglist` will not be passed to `autoload`.

- **[macro]** **defgeneric/auto** *name lambda-list &body options*

  A shorthand for `( defun/auto (defgeneric name) ...)`.

3.4 Classes

- **[macro]** **autoload-class** *class-name system-name &key docstring (metaclass 'standard-class)*

  This is the loaddef for autodef `defclass/auto`.

  Define a dummy class with `class-name` and arrange for `system-name` to be loaded when the class or any of its subclasses are instantiated. Returns the class object. The arguments are not evaluated. If `class-name` denotes a `class` and it is not `loaddef-class-p`, then it does nothing and returns `nil`.

  When `autoload-class` is macroexpanded during the compilation or loading of an `autoload-system`, it signals an `autoload-warning` if `system-name` is not among those declared in `:auto-depends-on`.

  - `docstring`, if non-`nil`, will be the stub's docstring. If `nil`, then a generic docstring that says what system it autoloads will be used.
  - `metaclass` is a symbol denoting `standard-class` or a subclass of it. Also, classes with this metaclass must be allowed to inherit from standard classes. In MOP terms, `closer-mop:validate-superclass` must return true when called with an instance of `metaclass` and an instance of `standard-class`.

  The dummy class is also defined at compile time to approximate the semantics of `defclass`. It has `metaclass` with a single superclass and no slots. These are visible through introspection (e.g. via `closer-mop:class-direct-superclasses`), which does not trigger autoloading.

  > *Note*: `initialize-instance :around` methods specialized on a subclass of `class-name` may run twice in the context of the `make-instance` that triggers autoloading.

- **[function]** **loaddef-class-p** *name*

  See if an `autoload-class` for `name` was established, and since then it has not been redefined (e.g. with `defclass/auto`, `defclass`) or deleted (with `(setf (find-class ...) nil)`). Subclasses do not inherit this property.

- **[macro]** **defclass/auto** *name direct-superclasses direct-slots &rest options*

  This is the autodef for the loaddef `autoload-class`.

Like `defclass`. `name` may be of the form `(definer name)`. In that case, instead of `defclass`, `definer` is used to establish the underlying class definition.

Loaddef: The corresponding loaddef is an `autoload-class` form. Note that the metaclass of the class `name` must already be defined when the loaddef is evaluated.

3.5 Variables

- **[function] loaddef-variable-p** *name*

  See if a loaddef was generated from a `defvar/auto` for `name`, but this autodef has not been evaluated.

- **[macro] defvar/auto** *var &optional (val nil) doc*

  This is an autodef with no public loaddef. See below.

  Unlike `defvar`, this works with the *global* binding on Lisps that support it (currently Allegro, CCL, ECL, SBCL). This is to handle the case when a system that uses `defvar` with a default value is autoloaded while that variable is locally bound:

```
;; Some base system only foreshadows *X*.
(declaim (special *x*))
(let ((*x* 1))
  ;; Imagine that the system that defines *X* is autoloaded here.
  (defvar/auto *x* 2)
  *x*)
=> 1
```

  Loaddef: The corresponding loaddef is not public and must be generated. The generated loaddef declaims the variable special and maybe sets its initial value and docstring. If the initial value form in `defvar/auto` is detected as a simple constant form, then it is evaluated and its value is assigned to the variable as in `defvar`. Simple constant forms are strings, numbers, characters, keywords, constants in the CL package, and `quote`d nested lists containing any of the previous or any symbol from the `cl` package and other packages for which loaddefs have been generated in the same `extract-loaddefs` call (see `defpackage/auto`)).

  In case the global binding of `var` has been set between the corresponding loaddef and its first autodef, `val` is evaluated for side effect.

3.6 Packages

- **[function] loaddef-package-p** *name*

  See if a loaddef was generated from a `defpackage/auto` for `name`, but this autodef has not been evaluated nor has the package been deleted.

- **[macro] defpackage/auto** *name &rest options*

  This is an autodef with no public loaddef. See below.

Unlike `defpackage`, if the package is already defined, `defpackage/auto` extends it additively. The additivity means that instead of replacing the package definition or signalling errors on redefinition, it expands into individual package-altering operations such as `shadow`, `use-package` and `export`. This allows the package state to be built incrementally, but it also means that the `(definer name)` syntax cannot be supported. `defpackage/auto` is idempotent.

In addition, `defpackage/auto` deviates from `defpackage` in the following ways.

- The default `:use` list is empty.
- `:size` is not supported.
- Implementation-specific extensions such as `:local-nicknames` are not supported. Use `add-package-local-nickname` after the `defpackage/auto`.

Loaddef: The corresponding loaddef is not public and must be generated. As in the expansion of `defpackage/auto` itself, the generated operations are additive.

- The generated loaddef reconstructs the package states as they exist after all `:auto-depends-on` systems are loaded. Thus, manual modifications after the `defpackage/auto` definition (e.g. by additional `export`s) are reflected in the loaddef.
- To handle circular dependencies, the loaddefs of all autodef packages and those passed in the `packages` argument to `extract-loaddefs` are generated in an interleaved manner. First, all packages are created, then their state is reconstructed in phases following `defpackage`.
- Any reference to non-existent packages (e.g. in `:use`) or symbols in non-existent packages (e.g. `:import-from`) is silently skipped when the loaddef is evaluated.

Instead of `defpackage/auto`, one may use, for example, `defpackage` or `uiop:define-package` and arrange for Automatically Generating Loaddefs for the package by listing it in `:packages` of `:auto-loaddefs`.

4 ASDF Integration

- **[class] autoload-system** *asdf/system:system*

Inheriting from this class in an `asdf:defsystem` form enables the features documented in the reader methods. Consider the following example.

```
(asdf:defsystem "my-system"
  :defsystem-depends-on ("autoload")
  :class "autoload:autoload-system"
  :auto-depends-on ("dyndep")
  :auto-loaddefs "loaddefs.lisp"
  :components ((:file "package")
               (:file "loaddefs")
               ...))
```

With the above,

- It is an error if an `autoload` refers to a system other than `dyndep`.
- ("`record-loaddefs "my-system")` will update `loaddefs.lisp`.
- `(asdf:test-system "my-system")` checks that `loaddefs.lisp` is up-to-date.

If the package definitions are also generated with `record-loaddefs` (e.g. because there is a `defpackage/auto` in `dyndep` or `:auto-loaddefs` specifies `:packages`), then we can do without the `package.lisp` file:

```
(asdf:defsystem "my-system"
  :defsystem-depends-on ("autoload")
  :class "autoload:autoload-system"
  :auto-depends-on ("dyndep")
  :auto-loaddefs ("loaddefs.lisp" :packages #:my-pkg)
  :components ((:file "loaddefs")
               ...))
```

- **[class]** **autoload-cl-source-file** *asdf/lisp-action:cl-source-file*

  This is the `:default-component-class` of `autoload-system`. ASDF Integration relies on source files belonging to this class. When combining autoload with another ASDF extension that has its own `asdf:cl-source-file` subclass, define a new class that inherits from both, and use that as `:default-component-class`.

- **[reader]** **system-auto-depends-on** *autoload-system (:auto-depends-on = nil)*

  This is the list of the names of systems that this system may autoload. The names are canonicalized with `asdf:coerce-name`. It is an `autoload-warning` if a loaddef refers to a system not listed here. This is also used by `extract-loaddefs` and affects the checks performed by Loading Systems.

- **[reader]** **system-auto-loaddefs** *autoload-system (:auto-loaddefs = nil)*

  When non-`nil`, this specifies arguments for Automatically Generating Loaddefs. It may be a single pathname designator or a list of the form

```
(loaddefs-file &key (process-arglist t) (process-docstring t)
                    packages (test t))
```

  - `loaddefs-file` designates the pathname where `record-loaddefs` writes the extracted loaddefs. The pathname is relative to `asdf:system-source-directory` of `system` and is `open`ed with `:if-exists :supersede`.
  - `process-arglist`, `process-docstring` and `packages` are passed on by `record-loaddefs` to `extract-loaddefs`.
  - If `test`, then `check-loaddefs` is run by `asdf:test-system`.

  Conditions signalled while ASDF is compiling or loading the file given have a `record-loaddefs` restart.

- **[function]** **autodeps** *system &key (cross-autoloaded t) installer*

Return the list of system names that may be autoloaded by `system` or any of its direct or indirect dependencies. This recursively visits systems in the dependency tree, traversing both normal (`:depends-on`) and autoloaded (`:auto-depends-on`) dependencies. It works even if `system` is not an `autoload-system`.

- `cross-autoloaded` controls whether systems only reachable from `system` via intermediate autoloaded dependencies are visited. Thus, if `cross-autoloaded` is `nil`, then the returned list is the first boundary of autoloaded systems.
- If `installer` is non-`nil`, it is called when an autoloaded system that is not installed (i.e. `asdf:find-system` fails) is visited. `installer` is passed a single argument, the name of the system to be installed. It may or may not install the system.

If an autoloaded system is not installed (i.e. `asdf:find-system` fails, even after `installer` had a chance), then its dependencies are unknown and cannot be traversed. Note that autoloaded systems that are not installed are still visited and included in the returned list.

The following example makes sure that all autoloaded dependencies (direct or indirect) of `my-system` are installed:

```
(autodeps "my-system" :installer #'ql:quickload)
```

4.1 Automatically Generating Loaddefs

- **[function]** **extract-loaddefs** *system &key (process-arglist t) (process-docstring t) packages*

  List the loaddef forms of the autodef definitions in `:auto-depends-on` of `system`.

  There is rarely a need to call this function directly, as `record-loaddefs` and `check-loaddefs` provide ASDF Integration.

  > *Note*: This is an expensive operation, as it loads or reloads the direct dependencies listed in `:auto-depends-on` one by one with `asdf:load-system :force t` to find the autodefs. As a side-effect, this can later cause spurious recompilation of systems that depend on the force-loaded systems.

  See the individual autodefs for descriptions of the generated loaddefs.

  - If `process-docstring`, then the docstrings extracted from autodef definitions will be associated with the definition.

  Note that if a definition is not made with an autodef, then `extract-loaddefs` will not detect it. For such functions, loaddefs must be written manually.

- **[function]** **write-loaddefs** *loaddefs stream*

  Write `loaddefs` to `stream` so they can be `load`ed or included in an `asdf:defsystem`.

- **[function]** **record-loaddefs** *system*

  `extract-loaddefs` from `system` and `write-loaddefs`. The arguments of these functions are taken from `system`'s `:auto-loaddefs`.

As `extract-loaddefs` loads the direct autoloaded dependencies, compiler warnings (e.g. about undefined specials and functions) may occur that go away once the generated loaddefs are in place. The easiest way to trigger this is to call `record-loaddefs` before these dependencies have been loaded. In this case, temporarily emptying the loaddefs file and fixing these warnings is recommended.

`record-loaddefs` may also be used as a condition handler, in which case it invokes the `record-loaddefs` restart.

- **[function] check-loaddefs** *system &key (errorp t)*

  In the `autoload-system` `system`, check that both recorded and manual autoload declarations are correct.

  - If `:auto-loaddefs` is specified, check that the file generated by `record-loaddefs` is up-to-date.
  - Check that all manual (non-generated) loaddefs in `system` are resolved (e.g. no longer `loaddef-function-p`) by loading `:auto-depends-on`.

  If `errorp`, then signal an error if a check fails or the loaddefs file cannot be read. If `:auto-loaddefs` is specified, then the `record-loaddefs` restart is provided.

  If `errorp` is `nil`, then instead of signalling an error, return `nil`.

  This function is called automatically by `asdf:test-op` on an `autoload-system` method if `:auto-loaddefs` has `:test t`.

- **[restart] record-loaddefs**

  Provided by `check-loaddefs` and also when the compilation of the loaddefs file declared in `:auto-loaddefs` fails. The function `record-loaddefs` can be used as a condition handler to invoke this restart.

5 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: `DEFTYPE`s, classes, conditions, structs
- *d*: documentation sections and glossary terms
- *l*: definitions of definition types
- *s*: ASDF systems
- *p*: packages
- *n*: named readtables
- *v*: special variables and constants
- *r*: restarts

- *?*: other

5.1 Function and Macro Index

5.2 Type Index

5.3 Misc Index

5.4 Concept Index