ASDF: Another System Definition Facility

This manual describes ASDF, a system definition facility for Common Lisp programs and libraries.

You can find the latest version of this manual at http://common-lisp.net/project/asdf/asdf.html.

Table of Contents

1 Introduction

ASDF is Another System Definition Facility: a tool for specifying how systems of Common Lisp software are comprised of components (sub-systems and files), and how to operate on these components in the right order so that they can be compiled, loaded, tested, etc.

ASDF presents three faces: one for users of Common Lisp software who want to reuse other people's code, one for writers of Common Lisp software who want to specify how to build their systems, one for implementers of Common Lisp extensions who want to extend the build system. See Section 3.4 [Loading a system], page 7, to learn how to use ASDF to load a system. See Chapter 5 [Defining systems with defsystem], page 11, to learn how to define a system of your own. See Chapter 6 [The object model of ASDF], page 19, for a description of the ASDF internals and how to extend ASDF.

Nota Bene: We have released ASDF 2.000 on May 31st 2010, and ASDF 3.0 on January 31st 2013. Releases of ASDF 2 and later have since then been included in all actively maintained CL implementations that used to bundle ASDF 1, plus some implementations that didn't use to, and has been made to work with all actively used CL implementations and a few more. See Chapter 12 ["What has changed between ASDF 1 and ASDF 2?"], page 55. Furthermore, it is possible to upgrade from ASDF 1 to ASDF 2 or ASDF 3 on the fly. For this reason, we have stopped supporting ASDF 1 and ASDF 2. If you are using ASDF 1 or ASDF 2 and are experiencing any kind of issues or limitations, we recommend you upgrade to ASDF 3 — and we explain how to do that. See Chapter 2 [Loading ASDF], page 2.

Also note that ASDF is not to be confused with ASDF-Install. ASDF-Install is not part of ASDF, but a separate piece of software. ASDF-Install is also unmaintained and obsolete. We recommend you use Quicklisp instead, which works great and is being actively maintained. If you want to download software from version control instead of tarballs, so you may more easily modify it, we recommend clbuild.

2 Loading ASDF

2.1 Loading a pre-installed ASDF

Most recent Lisp implementations include a copy of ASDF 2, and soon ASDF 3. You can usually load this copy using Common Lisp's `require` function:

```
(require "asdf")
```

As of the writing of this manual, the following implementations provide ASDF 2 this way: abcl allegro ccl clisp cmucl ecl lispworks mkcl sbcl xcl. The following implementation doesn't provide it yet but will in an upcoming release: scl. The following implementations are obsolete, not actively maintained, and most probably will never bundle it: cormanlisp gcl genera mcl.

If the implementation you are using doesn't provide ASDF 2 or ASDF 3, see see Chapter 2 [Loading an otherwise installed ASDF], page 2 below. If that implementation is still actively maintained, you may also send a bug report to your Lisp vendor and complain about their failing to provide ASDF.

NB: all implementations except clisp also accept `(require "ASDF")`, `(require 'asdf)` and `(require :asdf)`. For portability's sake, you probably want to use `(require "asdf")`.

2.2 Checking whether ASDF is loaded

To check whether ASDF is properly loaded in your current Lisp image, you can run this form:

```
(asdf:asdf-version)
```

If it returns a string, that is the version of ASDF that is currently installed.

If it raises an error, then either ASDF is not loaded, or you are using an old version of ASDF.

You can check whether an old version is loaded by checking if the ASDF package is present. The form below will allow you to programmatically determine whether a recent version is loaded, an old version is loaded, or none at all:

```
(when (find-package :asdf)
  (let ((ver (symbol-value (or (find-symbol (string :*asdf-version*) :asdf)
                               (find-symbol (string :*asdf-revision*) :asdf)))))
    (etypecase ver
      (string ver)
      (cons (with-output-to-string (s)
              (loop for (n . m) on ver do (princ n s) (when m (princ "." s)))))
      (null "1.0"))))
```

If it returns `nil` then ASDF is not installed. Otherwise it should return a string. If it returns `"1.0"`, then it can actually be any version before 1.77 or so, or some buggy variant of 1.x.

If you are experiencing problems with ASDF, please try upgrading to the latest released version, using the method below, before you contact us and raise an issue.

2.3 Upgrading ASDF

If your implementation provides ASDF 3 or later, you only need to `(require "asdf")`: ASDF will automatically look whether an updated version of itself is available amongst the regularly configured systems, before it compiles anything else. See see Chapter 3 [Configuring ASDF], page 5 below.

If your implementation does provide ASDF 2 or later, but not ASDF 3 or later, and you want to upgrade to a more recent version, you need to install and configure your ASDF as above, and additionally, you need to explicitly tell ASDF to load itself, right after you require your implementation's old ASDF 2:

```
(require "asdf")
(asdf:load-system :asdf)
```

If on the other hand, your implementation only provides an old ASDF, you will require a special configuration step and an old-style loading. Take special attention to not omit the trailing directory separator `/` at the end of your pathname:

```
(require "asdf")
(push #p"/path/to/new/asdf/" asdf:*central-registry*)
(asdf:oos 'asdf:load-op :asdf)
```

Note that ASDF 1 won't redirect its output files, or at least won't do it according to your usual ASDF 2 configuration. You therefore need write access on the directory where you install the new ASDF, and make sure you're not using it for multiple mutually incompatible implementations. At worst, you may have to have multiple copies of the new ASDF, e.g. one per implementation installation, to avoid clashes. Note that to our knowledge all implementations that provide ASDF provide ASDF 2 in their latest release, so you may want to upgrade your implementation rather than go through that hoop.

Finally, if you are using an unmaintained implementation that does not provide ASDF at all, see see Chapter 2 [Loading an otherwise installed ASDF], page 2 below.

Note that there are some limitations to upgrading ASDF:

- Previously loaded ASDF extension becomes invalid, and will need to be reloaded. This applies to e.g. CFFI-Grovel, or to hacks used by ironclad, etc. Since it isn't possible to automatically detect what extensions are present that need to be invalidated, ASDF will actually invalidate all previously loaded systems when it is loaded on top of a different ASDF version, starting with ASDF 2.014.8 (as far as releases go, 2.015); and it will automatically attempt this self-upgrade as its very first step starting with ASDF 3.
- For this an many other reasons, it important reason to load, configure and upgrade ASDF (if needed) as one of the very first things done by your build and startup scripts. Until all implementations provide ASDF 3 or later, it is safer if you upgrade ASDF and its extensions as a special step at the very beginning of whatever script you are running, before you start using ASDF to load anything else; even afterwards, it is still a good idea, to avoid having to load and reload code twice as it gets invalidated.
- Until all implementations provide ASDF 3 or later, it is unsafe to upgrade ASDF as part of loading a system that depends on a more recent version of ASDF, since the new one might shadow the old one while the old one is running, and the running old one will be confused when extensions are loaded into the new one. In the meantime, we

recommend that your systems should *not* specify `:depends-on (:asdf)`, or `:depends-on ((:version :asdf "2.010"))`, but instead that they check that a recent enough ASDF is installed, with such code as:

```
(unless (or #+asdf2 (asdf:version-satisfies
                     (asdf:asdf-version) *required-asdf-version*))
  (error "FOO requires ASDF ~A or later." *required-asdf-version*))
```

- Until all implementations provide ASDF 3 or later, it is unsafe for a system to transitively depend on ASDF and not directly depend on ASDF; if any of the system you use either depends-on asdf, system-depends-on asdf, or transitively does, you should also do as well.

2.4 Loading an otherwise installed ASDF

If your implementation doesn't include ASDF, if for some reason the upgrade somehow fails, does not or cannot apply to your case, you will have to install the file '`asdf.lisp`' somewhere and load it with:

```
(load "/path/to/your/installed/asdf.lisp")
```

The single file '`asdf.lisp`' is all you normally need to use ASDF.

You can extract this file from latest release tarball on the ASDF website. If you are daring and willing to report bugs, you can get the latest and greatest version of ASDF from its git repository. See Chapter 11 [Getting the latest version], page 54.

For maximum convenience you might want to have ASDF loaded whenever you start your Lisp implementation, for example by loading it from the startup script or dumping a custom core — check your Lisp implementation's manual for details.

3 Configuring ASDF

3.1 Configuring ASDF to find your systems

So it may compile and load your systems, ASDF must be configured to find the '`.asd`' files that contain system definitions.

Since ASDF 2, the preferred way to configure where ASDF finds your systems is the `source-registry` facility, fully described in its own chapter of this manual. See Chapter 7 [Controlling where ASDF searches for systems], page 31.

The default location for a user to install Common Lisp software is under '`~/.local/share/common-lisp/source/`'. If you install software there (it can be a symlink), you don't need further configuration. If you're installing software yourself at a location that isn't standard, you have to tell ASDF where you installed it. See below. If you're using some tool to install software (e.g. Quicklisp), the authors of that tool should already have configured ASDF.

The simplest way to add a path to your search path, say '`/home/luser/.asd-link-farm/`' is to create the directory '`~/.config/common-lisp/source-registry.conf.d/`' and there create a file with any name of your choice, and with the type '`conf`', for instance '`42-asd-link-farm.conf`' containing the line:

```
(:directory "/home/luser/.asd-link-farm/")
```

If you want all the subdirectories under '`/home/luser/lisp/`' to be recursively scanned for '`.asd`' files, instead use:

```
(:tree "/home/luser/lisp/")
```

Note that your Operating System distribution or your system administrator may already have configured system-managed libraries for you.

The required '`.conf`' extension allows you to have disabled files or editor backups (ending in '`~`'), and works portably (for instance, it is a pain to allow both empty and non-empty extension on CLISP). Excluded are files the name of which start with a '`.`' character. It is customary to start the filename with two digits that specify the order in which the directories will be scanned.

ASDF will automatically read your configuration the first time you try to find a system. You can reset the source-registry configuration with:

```
(asdf:clear-source-registry)
```

And you probably should do so before you dump your Lisp image, if the configuration may change between the machine where you save it at the time you save it and the machine you resume it at the time you resume it. Actually, you should use `(asdf:clear-configuration)` before you dump your Lisp image, which includes the above.

3.2 Configuring ASDF to find your systems — old style

The old way to configure ASDF to find your systems is by `push`ing directory pathnames onto the variable `asdf:*central-registry*`.

You must configure this variable between the time you load ASDF and the time you first try to use it. Loading and configuring ASDF presumably happen as part of some

initialization script that builds or starts your Common Lisp software system. (For instance, some SBCL users used to put it in their '`~/.sbclrc`'.)

The `asdf:*central-registry*` is empty by default in ASDF 2 or ASDF 3, but is still supported for compatibility with ASDF 1. When used, it takes precedence over the above source-registry[1].

For instance, if you wanted ASDF to find the '`.asd`' file '`/home/me/src/foo/foo.asd`' your initialization script could after it loads ASDF with `(require "asdf")` configure it with:

```
(push "/home/me/src/foo/" asdf:*central-registry*)
```

Note the trailing slash: when searching for a system, ASDF will evaluate each entry of the central registry and coerce the result to a pathname[2] at which point the presence of the trailing directory name separator is necessary to tell Lisp that you're discussing a directory rather than a file.

Typically, however, there are a lot of '`.asd`' files, and a common idiom was to have to put a bunch of *symbolic links* to '`.asd`' files in a common directory and push *that* directory (the "link farm") to the `asdf:*central-registry*` instead of pushing each of the many involved directories to the `asdf:*central-registry*`. ASDF knows how to follow such *symlinks* to the actual file location when resolving the paths of system components (on Windows, you can use Windows shortcuts instead of POSIX symlinks; if you try aliases under MacOS, we are curious to hear about your experience).

For example, if `#p"/home/me/cl/systems/"` (note the trailing slash) is a member of `*central-registry*`, you could set up the system *foo* for loading with asdf with the following commands at the shell:

```
$ cd /home/me/cl/systems/
$ ln -s ~/src/foo/foo.asd .
```

This old style for configuring ASDF is not recommended for new users, but it is supported for old users, and for users who want to programmatically control what directories are added to the ASDF search path.

3.3 Configuring where ASDF stores object files

ASDF lets you configure where object files will be stored. Sensible defaults are provided and you shouldn't normally have to worry about it.

This allows the same source code repository may be shared between several versions of several Common Lisp implementations, between several users using different compilation

[1] It is possible to further customize the system definition file search. That's considered advanced use, and covered later: search forward for `*system-definition-search-functions*`. See Chapter 5 [Defining systems with defsystem], page 11.

[2] ASDF will indeed call `eval` on each entry. It will also skip entries that evaluate to `nil`.

Strings and pathname objects are self-evaluating, in which case the `eval` step does nothing; but you may push arbitrary SEXP onto the central registry, that will be evaluated to compute e.g. things that depend on the value of shell variables or the identity of the user.

The variable `asdf:*central-registry*` is thus a list of "system directory designators". A *system directory designator* is a form which will be evaluated whenever a system is to be found, and must evaluate to a directory to look in. By "directory" here, we mean "designator for a pathname with a supplied DIRECTORY component".

options and without write privileges on shared source directories, etc. This also allows to keep source directories uncluttered by plenty of object files.

Starting with ASDF 2, the `asdf-output-translations` facility was added to ASDF itself, that controls where object files will be stored. This facility is fully described in a chapter of this manual, Chapter 8 [Controlling where ASDF saves compiled files], page 39.

The simplest way to add a translation to your search path, say from '`/foo/bar/baz/quux/`' to '`/where/i/want/my/fasls/`' is to create the directory '`~/.config/common-lisp/asdf-output-translations.conf.d/`' and there create a file with any name of your choice and the type '`conf`', for instance '`42-bazquux.conf`' containing the line:

```
("/foo/bar/baz/quux/" "/where/i/want/my/fasls/")
```

To disable output translations for source under a given directory, say '`/toto/tata/`' you can create a file '`40-disable-toto.conf`' with the line:

```
("/toto/tata/")
```

To wholly disable output translations for all directories, you can create a file '`00-disable.conf`' with the line:

```
(t t)
```

Note that your Operating System distribution or your system administrator may already have configured translations for you. In absence of any configuration, the default is to redirect everything under an implementation-dependent subdirectory of '`~/.cache/common-lisp/`'. See Chapter 7 [Controlling where ASDF searches for systems], page 31, for full details.

The required '`.conf`' extension allows you to have disabled files or editor backups (ending in '`~`'), and works portably (for instance, it is a pain to allow both empty and non-empty extension on CLISP). Excluded are files the name of which start with a '`.`' character. It is customary to start the filename with two digits that specify the order in which the directories will be scanned.

ASDF will automatically read your configuration the first time you try to find a system. You can reset the source-registry configuration with:

```
(asdf:clear-output-translations)
```

And you probably should do so before you dump your Lisp image, if the configuration may change between the machine where you save it at the time you save it and the machine you resume it at the time you resume it. (Once again, you should use `(asdf:clear-configuration)` before you dump your Lisp image, which includes the above.)

Finally note that before ASDF 2, other ASDF add-ons offered the same functionality, each in subtly different and incompatible ways: ASDF-Binary-Locations, cl-launch, common-lisp-controller. ASDF-Binary-Locations is now not needed anymore and should not be used. cl-launch 3.000 and common-lisp-controller 7.2 have been updated to just delegate this functionality to ASDF.

3.4 Resetting Configuration

When you dump and restore an image, or when you tweak your configuration, you may want to reset the ASDF configuration. For that you may use the following function:

`clear-configuration` [Function]
: undoes any ASDF configuration, regarding source-registry or output-translations.

If you use SBCL, CMUCL or SCL, you may use this snippet so that the ASDF configuration be cleared automatically as you dump an image:

```
#+(or cmu sbcl scl)
(pushnew 'clear-configuration
         #+(or cmu scl) ext:*before-save-initializations*
         #+sbcl sb-ext:*save-hooks*)
```

For compatibility with all Lisp implementations, however, you might want instead your build script to explicitly call `(asdf:clear-configuration)` at an appropriate moment before dumping.

4 Using ASDF

4.1 Loading a system

The system *foo* is loaded (and compiled, if necessary) by evaluating the following Lisp form:

```
(asdf:load-system :foo)
```

On some implementations (namely recent versions of ABCL, Allegro CL, Clozure CL, CMUCL, ECL, GNU CLISP, LispWorks, MKCL, SBCL and XCL), ASDF hooks into the `CL:REQUIRE` facility and you can just use:

```
(require :foo)
```

In older versions of ASDF, you needed to use `(asdf:oos 'asdf:load-op :foo)`. If your ASDF is too old to provide `asdf:load-system` though we recommend that you upgrade to ASDF 3. See Chapter 2 [Loading an otherwise installed ASDF], page 2.

Note the name of a system is specified as a string or a symbol, typically a keyword. If a symbol (including a keyword), its name is taken and lowercased. The name must be a suitable value for the `:name` initarg to `make-pathname` in whatever filesystem the system is to be found. The lower-casing-symbols behaviour is unconventional, but was selected after some consideration. Observations suggest that the type of systems we want to support either have lowercase as customary case (unix, mac, windows) or silently convert lowercase to uppercase (lpns), so this makes more sense than attempting to use `:case :common`, which is reported not to work on some implementations

4.2 Other Operations

ASDF provides three commands for the most common system operations: `load-system`, `compile-system` or `test-system`. It also provides `require-system`, a version of `load-system` that skips trying to update systems that are already loaded.

Because ASDF is an extensible system for defining *operations* on *components*, it also provides a generic function `operate` (which is usually abbreviated by `oos`). You'll use `oos` whenever you want to do something beyond compiling, loading and testing.

Output from ASDF and ASDF extensions are supposed to be sent to the CL stream `*standard-output*`, and so rebinding that stream around calls to `asdf:operate` should redirect all output from ASDF operations.

Reminder: before ASDF can operate on a system, however, it must be able to find and load that system's definition. See Chapter 3 [Configuring ASDF to find your systems], page 5.

For the advanced users, note that `require-system` calls `load-system` with keyword arguments `:force-not (already-loaded-systems)`. `already-loaded-systems` returns a list of the names of loaded systems. `load-system` applies `operate` with the operation from `*load-system-operation*`, which by default is `load-op`, the system, and any provided keyword arguments.

4.3 Summary

To use ASDF:

- Load ASDF itself into your Lisp image, either through `(require "asdf")` or else through `(load "/path/to/asdf.lisp")`.
- Make sure ASDF can find system definitions thanks to proper source-registry configuration.
- Load a system with `(asdf:load-system :my-system)` or use some other operation on some system of your choice.

4.4 Moving on

That's all you need to know to use ASDF to load systems written by others. The rest of this manual deals with writing system definitions for Common Lisp software you write yourself, including how to extend ASDF to define new operation and component types.

5 Defining systems with defsystem

This chapter describes how to use asdf to define systems and develop software.

5.1 The defsystem form

Systems can be constructed programmatically by instantiating components using `make-instance`. Most of the time, however, it is much more practical to use a static `defsystem` form. This section begins with an example of a system definition, then gives the full grammar of `defsystem`.

Let's look at a simple system. This is a complete file that would usually be saved as '`hello-lisp.asd`':

```
(in-package :asdf)

(defsystem "hello-lisp"
  :description "hello-lisp: a sample Lisp system."
  :version "0.2.1"
  :author "Joe User <joe@example.com>"
  :licence "Public Domain"
  :components ((:file "packages")
               (:file "macros" :depends-on ("packages"))
               (:file "hello" :depends-on ("macros"))))
```

Some notes about this example:

- The file starts with an `in-package` form to use package `asdf`. You could instead start your definition by using a qualified name `asdf:defsystem`.
- If in addition to simply using `defsystem`, you are going to define functions, create ASDF extension, globally bind symbols, etc., it is recommended that to avoid namespace pollution between systems, you should create your own package for that purpose, for instance replacing the above `(in-package :asdf)` with:

```
(defpackage :foo-system
  (:use :cl :asdf))

(in-package :foo-system)
```

- The `defsystem` form defines a system named `hello-lisp` that contains three source files: '`packages`', '`macros`' and '`hello`'.
- The file '`macros`' depends on '`packages`' (presumably because the package it's in is defined in '`packages`'), and the file '`hello`' depends on '`macros`' (and hence, transitively on '`packages`'). This means that ASDF will compile and load '`packages`' and '`macros`' before starting the compilation of file '`hello`'.
- The files are located in the same directory as the file with the system definition. ASDF resolves symbolic links (or Windows shortcuts) before loading the system definition file and stores its location in the resulting system[1]. This is a good thing because the user can move the system sources without having to edit the system definition.

[1] It is possible, though almost never necessary, to override this behaviour.

- Make sure you know how the `:version` numbers will be parsed! They are parsed as period-separated lists of integers. I.e., in the example, `0.2.1` is to be interpreted, roughly speaking, as `(0 2 1)`. In particular, version `0.2.1` is interpreted the same as `0.0002.1` and is strictly version-less-than version `0.20.1`, even though the two are the same when interpreted as decimal fractions. Instead of a string representing the version, the `:version` argument can be an expression that is resolved to such a string using the following trivial domain-specific language: in addition to being a literal string, it can be an expression of the form `(:read-file-form <pathname-or-string> :at <access-at-specifier>)`, which will be resolved by reading a form in the specified pathname (read as a subpathname of the current system if relative or a unix-namestring). You may use a `uiop:access-at` specifier with the (optional) `:at` keyword, by default the specifier is `0`, meaning the first form is returned.

5.2 A more involved example

Let's illustrate some more involved uses of `defsystem` via a slightly convoluted example:

```
(defsystem "foo"
  :version "1.0.0"
  :components ((:module "mod"
                        :components ((:file "bar")
                                     (:file"baz")
                                     (:file "quux"))
                        :perform (compile-op :after (op c)
                                             (do-something c))
                        :explain (compile-op :after (op c)
                                             (explain-something c)))
                       (:file "blah")))
```

The `:module` component named `"mod"` is a collection of three files, which will be located in a subdirectory of the main code directory named '`mod`' (this location can be overridden; see the discussion of the `:pathname` option in Section 5.3 [The defsystem grammar], page 13).

The method-form tokens provide a shorthand for defining methods on particular components. This part

```
:perform (compile-op :after (op c)
                     (do-something c))
:explain (compile-op :after (op c)
                     (explain-something c))
```

has the effect of

```
(defmethod perform :after ((op compile-op) (c (eql ...)))
           (do-something c))
(defmethod explain :after ((op compile-op) (c (eql ...)))
           (explain-something c))
```

where `...` is the component in question. In this case `...` would expand to something like

```
(find-component "foo" "mod")
```

For more details on the syntax of such forms, see Section 5.3 [The defsystem grammar], page 13. For more details on what these methods do, see Section 6.1 [Operations], page 20 in Chapter 6 [The object model of ASDF], page 19.

5.3 The defsystem grammar

```
system-definition := ( defsystem system-designator system-option* )

system-option := :defsystem-depends-on system-list
                 | :weakly-depends-on system-list
                 | :class class-name (see discussion below)
                 | module-option
                 | option

module-option := :components component-list
                 | :serial [ t | nil ]

option :=
        | :pathname pathname-specifier
        | :default-component-class class-name
        | :perform method-form
        | :explain method-form
        | :output-files method-form
        | :operation-done-p method-form
        | :if-feature feature-expression
        | :depends-on ( dependency-def* )
        | :in-order-to ( dependency+ )


system-list := ( simple-component-name* )

component-list := ( component-def* )

component-def  := ( component-type simple-component-name option* )

component-type := :module | :file | :static-file | other-component-type

other-component-type := symbol-by-name (see Section 5.3 [Component types],
page 13)

# This is used in :depends-on, as opposed to ``dependency,''
# which is used in :in-order-to
dependency-def := simple-component-name
                | (feature feature-name)
                | ( :version simple-component-name version-specifier)

# ``dependency'' is used in :in-order-to, as opposed to
```

```
# ‘‘dependency-def’’
dependency := (dependent-op requirement+)
requirement := (required-op required-component+)
             | (:feature feature-name)
dependent-op := operation-name
required-op := operation-name

simple-component-name := string
                       |  symbol

pathname-specifier := pathname | string | symbol

method-form := (operation-name qual lambda-list &rest
 body)
qual := method qualifier

component-dep-fail-option := :fail | :try-next | :ignore

feature-expression := keyword | (:and feature-expression*)
                       | (:or feature-expression*) | (:not feature-expression)
```

5.3.1 Component names

Component names (`simple-component-name`) may be either strings or symbols.

5.3.2 Component types

Component type names, even if expressed as keywords, will be looked up by name in the current package and in the asdf package, if not found in the current package. So a component type `my-component-type`, in the current package `my-system-asd` can be specified as `:my-component-type`, or `my-component-type`.

`system` and its subclasses are *not* allowed as component types for such children components.

5.3.3 System class names

A system class name will be looked up in the same way as a Component type (see above), except that only `system` and its subclasses are allowed. Typically, one will not need to specify a system class name, unless using a non-standard system class defined in some ASDF extension, typically loaded through `DEFSYSTEM-DEPENDS-ON`, see below. For such class names in the ASDF package, we recommend that the `:class` option be specified using a keyword symbol, such as

```
:class :MY-NEW-SYSTEM-SUBCLASS
```

This practice will ensure that package name conflicts are avoided. Otherwise, the symbol `MY-NEW-SYSTEM-SUBCLASS` will be read into the current package *before* it has been exported from the ASDF extension loaded by `:defsystem-depends-on`, causing a name conflict in the current package.

5.3.4 Defsystem depends on

The `:defsystem-depends-on` option to `defsystem` allows the programmer to specify another ASDF-defined system or set of systems that must be loaded *before* the system definition is processed. Typically this is used to load an ASDF extension that is used in the system definition.

5.3.5 Weakly depends on

We do *NOT* recommend you use this feature. If you are tempted to write a system *foo* that weakly-depends-on a system *bar*, we recommend that you should instead write system *foo* in a parametric way, and offer some special variable and/or some hook to specialize its behavior; then you should write a system *foo+bar* that does the hooking of things together.

The (deprecated) `:weakly-depends-on` option to `defsystem` allows the programmer to specify another ASDF-defined system or set of systems that ASDF should *try* to load, but need not load in order to be successful. Typically this is used if there are a number of systems that, if present, could provide additional functionality, but which are not necessary for basic function.

Currently, although it is specified to be an option only to `defsystem`, this option is accepted at any component, but it probably only makes sense at the `defsystem` level. Programmers are cautioned not to use this component option except at the `defsystem` level, as this anomalous behavior may be removed without warning.

Finally, you might look into the `asdf-system-connections` extension, that will let you define additional code to be loaded when two systems are simultaneously loaded. It may or may not be considered good style, but at least it can be used in a way that has deterministic behavior independent of load order, unlike `weakly-depends-on`.

5.3.6 Pathname specifiers

A pathname specifier (`pathname-specifier`) may be a pathname, a string or a symbol. When no pathname specifier is given for a component, which is the usual case, the component name itself is used.

If a string is given, which is the usual case, the string will be interpreted as a Unix-style pathname where `/` characters will be interpreted as directory separators. Usually, Unix-style relative pathnames are used (i.e. not starting with `/`, as opposed to absolute pathnames); they are relative to the path of the parent component. Finally, depending on the `component-type`, the pathname may be interpreted as either a file or a directory, and if it's a file, a file type may be added corresponding to the `component-type`, or else it will be extracted from the string itself (if applicable).

For instance, the `component-type :module` wants a directory pathname, and so a string `"foo/bar"` will be interpreted as the pathname '`#p"foo/bar/"`'. On the other hand, the `component-type :file` wants a file of type `lisp`, and so a string `"foo/bar"` will be interpreted as the pathname '`#p"foo/bar.lisp"`', and a string `"foo/bar.quux"` will be interpreted as the pathname '`#p"foo/bar.quux.lisp"`'. Finally, the `component-type :static-file` wants a file without specifying a type, and so a string `"foo/bar"` will be interpreted as the pathname '`#p"foo/bar"`', and a string `"foo/bar.quux"` will be interpreted as the pathname '`#p"foo/bar.quux"`'.

ASDF does not interpret the string `".."` to designate the parent directory. This string will be passed through to the underlying operating system for interpretation. We *believe* that this will work on all platforms where ASDF is deployed, but do not guarantee this behavior. A pathname object with a relative directory component of `:up` or `:back` is the only guaranteed way to specify a parent directory.

If a symbol is given, it will be translated into a string, and downcased in the process. The downcasing of symbols is unconventional, but was selected after some consideration. Observations suggest that the type of systems we want to support either have lowercase as customary case (Unix, Mac, windows) or silently convert lowercase to uppercase (lpns), so this makes more sense than attempting to use `:case :common` as argument to `make-pathname`, which is reported not to work on some implementations.

Pathname objects may be given to override the path for a component. Such objects are typically specified using reader macros such as `#p` or `#.(make-pathname ...)`. Note however, that `#p...` is a shorthand for `#.(parse-namestring ...)` and that the behavior of `parse-namestring` is completely non-portable, unless you are using Common Lisp `logical-pathname`s, which themselves involve other non-portable behavior (see Section 5.3 [Using logical pathnames], page 13, below). Pathnames made with `#.(make-pathname ...)` can usually be done more easily with the string syntax above. The only case that you really need a pathname object is to override the component-type default file type for a given component. Therefore, pathname objects should only rarely be used. Unhappily, ASDF 1 didn't properly support parsing component names as strings specifying paths with directories, and the cumbersome `#.(make-pathname ...)` syntax had to be used. An alternative to `#.` read-time evaluation is to use `(eval '(defsystem ... ,pathname ...))`.

Note that when specifying pathname objects, ASDF does not do any special interpretation of the pathname influenced by the component type, unlike the procedure for pathname-specifying strings. On the one hand, you have to be careful to provide a pathname that correctly fulfills whatever constraints are required from that component type (e.g. naming a directory or a file with appropriate type); on the other hand, you can circumvent the file type that would otherwise be forced upon you if you were specifying a string.

5.3.7 Version specifiers

Version specifiers are strings to be parsed as period-separated lists of integers. I.e., in the example, `"0.2.1"` is to be interpreted, roughly speaking, as `(0 2 1)`. In particular, version `"0.2.1"` is interpreted the same as `"0.0002.1"`, though the latter is not canonical and may lead to a warning being issued. Also, `"1.3"` and `"1.4"` are both strictly `uiop:version<` to `"1.30"`, quite unlike what would have happened had the version strings been interpreted as decimal fractions.

System definers are encouraged to use version identifiers of the form *x.y.z* for major version, minor version and patch level, where significant API incompatibilities are signaled by an increased major number.

See Section 6.2.1 [Common attributes of components], page 26.

5.3.8 Using logical pathnames

We do not generally recommend the use of logical pathnames, especially not so to newcomers to Common Lisp. However, we do support the use of logical pathnames by old timers, when such is their preference.

To use logical pathnames, you will have to provide a pathname object as a `:pathname` specifier to components that use it, using such syntax as `#p"LOGICAL-HOST:absolute;path;to;component.lisp"`.

You only have to specify such logical pathname for your system or some top-level component. Sub-components' relative pathnames, specified using the string syntax for names, will be properly merged with the pathnames of their parents. The specification of a logical pathname host however is *not* otherwise directly supported in the ASDF syntax for pathname specifiers as strings.

The `asdf-output-translation` layer will avoid trying to resolve and translate logical pathnames. The advantage of this is that you can define yourself what translations you want to use with the logical pathname facility. The disadvantage is that if you do not define such translations, any system that uses logical pathnames will behave differently under asdf-output-translations than other systems you use.

If you wish to use logical pathnames you will have to configure the translations yourself before they may be used. ASDF currently provides no specific support for defining logical pathname translations.

Note that the reasons we do not recommend logical pathnames are that (1) there is no portable way to set up logical pathnames before they are used, (2) logical pathnames are limited to only portably use a single character case, digits and hyphens. While you can solve the first issue on your own, describing how to do it on each of fifteen implementations supported by ASDF is more than we can document. As for the second issue, mind that the limitation is notably enforced on SBCL, and that you therefore can't portably violate the limitations but must instead define some encoding of your own and add individual mappings to name physical pathnames that do not fit the restrictions. This can notably be a problem when your Lisp files are part of a larger project in which it is common to name files or directories in a way that includes the version numbers of supported protocols, or in which files are shared with software written in different programming languages where conventions include the use of underscores, dots or CamelCase in pathnames.

5.3.9 Serial dependencies

If the `:serial t` option is specified for a module, ASDF will add dependencies for each child component, on all the children textually preceding it. This is done as if by `:depends-on`.

```
:serial t
:components ((:file "a") (:file "b") (:file "c"))
```

is equivalent to

```
:components ((:file "a")
             (:file "b" :depends-on ("a"))
             (:file "c" :depends-on ("a" "b")))
```

5.3.10 Source location

The `:pathname` option is optional in all cases for systems defined via `defsystem`, and in the usual case the user is recommended not to supply it.

Instead, ASDF follows a hairy set of rules that are designed so that

1. `find-system` will load a system from disk and have its pathname default to the right place.

2. This pathname information will not be overwritten with `*default-pathname-defaults*` (which could be somewhere else altogether) if the user loads up the '`.asd`' file into his editor and interactively re-evaluates that form.

If a system is being loaded for the first time, its top-level pathname will be set to:

- The host/device/directory parts of `*load-truename*`, if it is bound.
- `*default-pathname-defaults*`, otherwise.

If a system is being redefined, the top-level pathname will be

- changed, if explicitly supplied or obtained from `*load-truename*` (so that an updated source location is reflected in the system definition)
- changed if it had previously been set from `*default-pathname-defaults*`
- left as before, if it had previously been set from `*load-truename*` and `*load-truename*` is currently unbound (so that a developer can evaluate a `defsystem` form from within an editor without clobbering its source location)

5.3.11 if-feature option

This option allows you to specify a feature expression to be evaluated as if by `#+` to conditionally include a component in your build. If the expression is false, the component is dropped as well as any dependency pointing to it. As compared to using `#+` which is expanded at read-time, this allows you to have an object in your component hierarchy that can be used for manipulations beside building your project. This option was added in ASDF 3.

5.3.12 if-component-dep-fails option

This option was removed in ASDF 3. Its semantics was limited in purpose and dubious to explain, and its implementation was breaking a hole into the ASDF object model. Please use the `if-feature` option instead.

5.4 Other code in .asd files

Files containing `defsystem` forms are regular Lisp files that are executed by `load`. Consequently, you can put whatever Lisp code you like into these files. However, it is recommended to keep such forms to a minimal, and to instead define `defsystem` extensions that you use with `:defsystem-depends-on`.

If however, you might insist on including code in the `.asd` file itself, e.g., to examine and adjust the compile-time environment, possibly adding appropriate features to `*features*`. If so, here are some conventions we recommend you follow, so that users can control certain details of execution of the Lisp in '`.asd`' files:

- Any informative output (other than warnings and errors, which are the condition system's to dispose of) should be sent to the standard CL stream `*standard-output*`, so that users can easily control the disposition of output from ASDF operations.

6 The object model of ASDF

ASDF is designed in an object-oriented way from the ground up. Both a system's structure and the operations that can be performed on systems follow a extensible protocol.

This allows the addition of behaviours: for example, `cffi` adds support of special FFI description files to interface with C libraries and of wrapper files to embed C code in Lisp; `abcl-jar` supports creating Java JAR archives in ABCL; and `poiu` supports for compiling code in parallel using background processes.

This chapter deals with `component`s and `operation`s.

A `component` represents an individual source file or a group of source files, and the things that get transformed into. A `system` is a component at the top level of the component hierarchy. A `source-file` is a component representing a single source-file and the successive output files into which it is transformed. A `module` is an intermediate component itself grouping several other components, themselves source-files or further modules.

An `Operation` represents a transformation that can be performed on a component, turning them from source files to intermediate results to final outputs.

A pair of an `operation` and a `component` is called an `action`. An `action` represents a particular build step to be `perform`ed, after all its dependencies have been fulfilled. In the ASDF model, actions depend on other actions. The term *action* itself was used by Kent Pitman in his old article, but was only used by ASDF hackers starting with the ASDF 2; but the concept is ubiquitous since the very beginning of ASDF 1, though previously implicit.

Then, there are many *functions* available to users, extenders and implementers of ASDF to use, define or implement the activities that are part of building your software. Though they manipulate `action`s, most of these functions do not take as an argument a reified pair (a `cons` cell) of an operation and a component; instead, they usually take two separate arguments, which allows to take advantage of the power CLOS-style multiple dispatch for fun and profit.

There are many *hooks* in which to add functionality, by customizing the behavior of existing *functions*.

Last but not least is the notion of *dependency* between two actions. The structure of dependencies between actions is a directed *dependency graph*. ASDF is invoked by being told to *operate* with some *operation* on some toplevel *system*; it will then *traverse* the graph and build a *plan* that follows its structure. To be successfully buildable, this graph of actions but be acyclic. If, as a user, extender or implementer of ASDF, you fail to keep the dependency graph without cycles, ASDF will fail loudly as it eventually finds one. To clearly distinguish the direction of dependencies, ASDF 3 uses the words *requiring* and *required* as applied to an action depending on the other: the requiring action `depends-on` the completion of all required actions before it may itself be `perform`ed.

Using the `defsystem` syntax, users may easily express direct dependencies along the graph of the object hierarchy: between a component and its parent, its children, and its siblings. By defining custom CLOS methods, you can express more elaborate dependencies as you wish. Most common operations, such as `load-op`, `compile-op` or `load-source-op` are automatically propagate "downward" the component hierarchy and are "covariant"

with it: to act the operation on the parent module, you must first act it on all the children components, with the action on the parent being parent of the action on each child. Other operations, such as `prepare-op` and `prepare-source-op` (introduced in ASDF 3) are automatically propagated "upward" the component hierarchy and are "contravariant" with it: to perform the operation of preparing for compilation of a child component, you must perform the operation of preparing for compilation of its parent component, and so on, ensuring that all the parent's dependencies are (compiled and) loaded before the child component may be compiled and loaded. Yet other operations, such as `test-op` or `load-fasl-op` remain at the system level, and are not propagated along the hierarchy, but instead do something global on the system.

6.1 Operations

An *operation* object of the appropriate type is instantiated whenever the user wants to do something with a system like

- compile all its files
- load the files into a running lisp environment
- copy its source files somewhere else

Operations can be invoked directly, or examined to see what their effects would be without performing them. There are a bunch of methods specialised on operation and component type that actually do the grunt work.

The operation object contains whatever state is relevant for this purpose (perhaps a list of visited nodes, for example) but primarily is a nice thing to specialise operation methods on and easier than having them all be `EQL` methods.

Operations are invoked on systems via `operate`.

`operate` *operation system &rest*`initargs` *&key*`force force-not verbose` *&allow-other-keys* [Generic function]

`oos` *operation system &rest*`initargs` *&key&allow-other-keys* [Generic function]

`operate` invokes *operation* on *system*. `oos` is a synonym for `operate`.

operation is a symbol that is passed, along with the supplied *initargs*, to `make-instance` to create the operation object. *system* is a system designator.

The *initargs* are passed to the `make-instance` call when creating the operation object. Note that dependencies may cause the operation to invoke other operations on the system or its components: the new operations will be created with the same *initargs* as the original one.

If *force* is `:all`, then all systems are forced to be recompiled even if not modified since last compilation. If *force* is `t`, then only the system being loaded is forced to be recompiled even if not modified since last compilation, but other systems are not affected. If *force* is a list, then it specifies a list of systems that are forced to be recompiled even if not modified since last compilation. If *force-not* is `:all`, then all systems are forced not to be recompiled even if modified since last compilation. If *force-not* is `t`, then only the system being loaded is forced not to be recompiled even if modified since last compilation, but other systems are not affected. If *force-not* is a list, then it specifies a list of systems that are forced not to be recompiled even if

modified since last compilation. *force* takes precedences over *force-not*; both of them apply to systems that are dependencies and were already compiled.

To see what `operate` would do, you can use:

```
(asdf:traverse operation-class system-name)
```

6.1.1 Predefined operations of ASDF

All the operations described in this section are in the `asdf` package. They are invoked via the `operate` generic function.

```
(asdf:operate 'asdf:operation-name :system-name {operation-options ...})
```

`compile-op` [Operation]

This operation compiles the specified component. A `cl-source-file` will be `compile-file`'d. All the children and dependencies of a system or module will be recursively compiled by `compile-op`.

`compile-op` depends on `prepare-op` which itself depends on a `load-op` of all of a component's dependencies, as well as of its parent's dependencies. When `operate` is called on `compile-op`, all these dependencies will be loaded as well as compiled; yet, some parts of the system main remain unloaded, because nothing depends on them. Use `load-op` to load a system.

`load-op` [Operation]

This operation loads the compiled code for a specified component. A `cl-source-file` will have its compiled fasl `load`ed, which fasl is the output of `compile-op` that `load-op` depends on. All the children and dependencies of a system or module will be recursively loaded by `load-op`.

`load-op` depends on `prepare-op` which itself depends on a `load-op` of all of a component's dependencies, as well as of its parent's dependencies.

`prepare-op` [Operation]

This operation ensures that the dependencies of a component and its recursive parents are loaded (as per `load-op`), as a prerequisite before `compile-op` and `load-op` operations may be performed on a given component.

`load-source-op, prepare-source-op` [Operation]

`load-source-op` will load the source for the files in a module rather than they compiled fasl output. It has a `prepare-source-op` analog to `prepare-op`, that ensures the dependencies are themselves loaded via `load-source-op`.

There is no provision in ASDF for ensuring that some components are always loaded as source, while others are always compiled. While this idea often comes up in discussions, it actually doesn't play well with either the linking model of ECL or with various bundle operations (see below), and is eventually not workable; also the dependency model of ASDF would have to be modified incompatibly to allow for such trick. If your code doesn't compile cleanly, fix it. If compilation makes it slow, use `declaim` or `eval-when` to adjust your compiler settings, or eschew compilation by `eval`uating a quoted source form at load-time.

`test-op` [Operation]

This operation will perform some tests on the module. The default method will do nothing. The default dependency is to require `load-op` to be performed on the module first. The default `operation-done-p` is that the operation is *never* done — we assume that if you invoke the `test-op`, you want to test the system, even if you have already done so.

The results of this operation are not defined by ASDF. It has proven difficult to define how the test operation should signal its results to the user in a way that is compatible with all of the various test libraries and test techniques in use in the community.

People typically define `test-op` methods like thus:

```
(defmethod perform ((o asdf:test-op)
                    (s (eql (asdf:find-system :my-system))))
  (asdf:load-system :my-system-test)
  (funcall (read-from-string "my-system-test:test-suite")))
```

Using `load-system` in the perform method rather than an `:in-order-to` dependency, is sometimes necessary for backward compatibility with ASDF 2 and older, to avoid circular dependencies that could arise because of the way these old versions propagate dependencies.

If you don't care for compatibility with ASDF 2, you could use the following options in your `defsystem` form:

```
:in-order-to ((test-op (load-op :my-system-test)))
:perform (test-op (o c) (symbol-call :my-system-test :test-suite))
```

`fasl-op`, `monolithic-fasl-op`, `load-fasl-op`, `binary-op`, `monolithic-binary-op`, `lib-op`, `monolithic-lib-op`, `dll-op`, `monolithic-dll-op`, `program-op` [Operation]

These are "bundle" operations, that can create a single-file "bundle" for all the contents of each system in an application, or for the entire application.

`fasl-op` will create a single fasl file for each of the systems needed, grouping all its many fasls in one, so you can deliver each system as a single fasl. `monolithic-fasl-op` will create a single fasl file for target system and all its dependencies, so you can deliver your entire application as a single fasl. `load-fasl-op` will load the output of `fasl-op` (though if it the output is not up-to-date, it will load the intermediate fasls indeed as part of building it); this matters a lot on ECL, where the dynamic linking involved in loading tens of individual fasls can be noticeably more expensive than loading a single one.

Once you have created a fasl with `fasl-op`, you can use `precompiled-system` to deliver it in a way that is compatible with clients having dependencies on your system, whether it is distributed as source or as a single binary; the '`.asd`' file to be delivered with the fasl will look like this:

```
(defsystem :mysystem :class :precompiled-system
    :fasl (some expression that will evaluate to a pathname))
```

Or you can use `binary-op` to let ASDF create such a system for you as well as the `fasl-op` output, or `monolithic-binary-op`. This allows you to deliver code for your systems or applications as a single file. Of course, if you want to test the

result in the current image, *before* you try to use any newly created '`.asd`' files, you should not forget to `(asdf:clear-configuration)` or at least `(asdf:clear-source-registry)`, so it re-populates the source-registry from the filesystem.

The `program-op` operation will create an executable program from the specified system and its dependencies. You can use UIOP for its pre-image-dump hooks, its post-image-restore hooks, and its access to command-line arguments. And you can specify an entry point `my-app:main` by specifying in your `defsystem` the option `:entry-point "my-app:main"`. Depending on your implementation, running `(asdf:operate 'asdf:program-op :my-app)` may quit the current Lisp image upon completion. See the example in '`test/hello-world-example.asd`' and '`test/hello.lisp`', as built and tested by '`test/test-program.script`' and '`test/make-hello-world.lisp`'.

There is also `lib-op` for building a linkable '`.a`' file (Windows: '`.lib`') from all linkable object dependencies (FFI files, and on ECL, Lisp files too), and its monolithic equivalent `monolithic-lib-op`. And there is also `dll-op` (respectively its monolithic equivalent `monolithic-lib-op`) for building a linkable '`.so`' file (Windows: '`.dll`', MacOS X: '`.dynlib`') to create a single dynamic library for all the extra FFI code to be linked into each of your systems (respectively your entire application).

All these "bundle" operations are available since ASDF 3 on all actively supported Lisp implementations, but may be unavailable on unmaintained legacy implementations. This functionality was previously available for select implementations, as part of a separate system `asdf-bundle`, itself descended from the ECL-only `asdf-ecl`.

The pathname of the output of bundle operations is subject to output-translation as usual, unless the operation is equal to the `:build-operation` argument to `defsystem`. This behavior is not very satisfactory and may change in the future. Maybe you have suggestions on how to better configure it?

`concatenate-source-op`, `monolithic-concatenate-source-op`, `load-concatenated-source-op`, `compile-concatenated-source-op`, `load-compiled-concatenated-source-op`, `monolithic-load-concatenated-source-op`, `monolithic-compile-concatenated-source-op`, `monolithic-load-compiled-concatenated-source-op` [Operation]

These operation, as their respective names indicate, consist in concatenating all `cl-source-file` source files in a system (or in a system and all its dependencies, if monolithic), in the order defined by dependencies, then loading the result, or compiling then loading the result.

These operations are useful to deliver a system or application as a single source file, and for testing that said file loads properly, or compiles then loads properly.

ASDF itself is notably delivered as a single source file this way using `monolithic-concatenate-source-op`, transcluding a prelude and the `uiop` library before the `asdf/defsystem` system itself.

6.1.2 Creating new operations

ASDF was designed to be extensible in an object-oriented fashion. To teach ASDF new tricks, a programmer can implement the behaviour he wants by creating a subclass of `operation`.

ASDF's pre-defined operations are in no way "privileged", but it is requested that developers never use the `asdf` package for operations they develop themselves. The rationale for this rule is that we don't want to establish a "global asdf operation name registry", but also want to avoid name clashes.

Your operation *must* usually provide methods for one or more of the following generic functions:

- `perform` Unless your operation, like `prepare-op`, is for dependency propagation only, the most important function for which to define a method is usually `perform`, which will be called to perform the operation on a specified component, after all dependencies have been performed.

  The `perform` method must call `output-files` (see below) to find out where to put its files, because the user is allowed to override the method or tweak the output-translation mechanism. Perform should only use the primary value returned by `output-files`. If one and only one output file is expected, it can call `output-file` that checks that this is the case and returns the first and only list element.

- `output-files` If your perform method has any output, you must define a method for this function. for ASDF to determine where the outputs of performing operation lie.

  Your method may return two values, a list of pathnames, and a boolean. If the boolean is `nil` (or you fail to return multiple values), then enclosing `:around` methods may translate these pathnames, e.g. to ensure object files are somehow stored in some implementation-dependent cache. If the boolean is `t` then the pathnames are marked not be translated by the enclosing `:around` method.

- `component-depends-on` If the action of performing the operation on a component has dependencies, you must define a method on `component-depends-on`.

  Your method will take as specialized arguments an operation and a component which together identify an action, and return a list of entries describing actions that this action depends on. The format of entries is described below.

  It is *strongly* advised that you should always append the results of `(call-next-method)` to the results of your method, or "interesting" failures will likely occur, unless you're a true specialist of ASDF internals. It is unhappily too late to compatibly use the `append` method combination, but conceptually that's the protocol that is being manually implemented.

  Each entry returned by `component-depends-on` is itself a list.

  The first element of an entry is an operation designator: either an operation object designating itself, or a symbol that names an operation class (that ASDF will instantiate using `make-operation`). For instance, `load-op`, `compile-op` and `prepare-op` are common such names, denoting the respective operations.

  The rest of each entry is a list of component designators: either a component object designating itself, or an identifier to be used with `find-component`. `find-component` will be called with the current component's parent as parent, and the identifier as second argument. The identifier is typically a string, a symbol (to be downcased as per `coerce-name`), or a list of strings or symbols. In particular, the empty list `nil` denotes the parent itself.

An operation *may* provide methods for the following generic functions:

- `input-files` A method for this function is often not needed, since ASDF has a pretty clever default `input-files` mechanism. You only need create a method if there are multiple ultimate input files, and/or the bottom one doesn't depend on the `component-pathname` of the component.
- `operation-done-p` You only need to define a method on that function if you can detect conditions that invalidate previous runs of the operation, even though no filesystem timestamp has changed, in which case you return `nil` (the default is `t`).

  For instance, the method for `test-op` always returns `nil`, so that tests are always run afresh. Of course, the `test-op` for your system could depend on a deterministically repeatable `test-report-op`, and just read the results from the report files, in which case you could have this method return `t`.

Operations that print output should send that output to the standard CL stream `*standard-output*`, as the Lisp compiler and loader do.

6.2 Components

A *component* represents a source file or (recursively) a collection of components. A *system* is (roughly speaking) a top-level component that can be found via `find-system`.

A *system designator* is a system itself, or a string or symbol that behaves just like any other component name (including with regard to the case conversion rules for component names).

A *component designator*, relative to a base component, is either a component itself, or a string or symbol, or a list of designators.

`find-system` *system-designator &optional*(*error-p t*) [Function]

Given a system designator, `find-system` finds and returns a system. If no system is found, an error of type `missing-component` is thrown, or `nil` is returned if `error-p` is false.

To find and update systems, `find-system` funcalls each element in the `*system-definition-search-functions*` list, expecting a pathname to be returned, or a system object, from which a pathname may be extracted, and that will be registered. The resulting pathname (if any) is loaded if one of the following conditions is true:

- there is no system of that name in memory
- the pathname is different from that which was previously loaded
- the file's `last-modified` time exceeds the `last-modified` time of the system in memory

When system definitions are loaded from '`.asd`' files, a new scratch package is created for them to load into, so that different systems do not overwrite each others operations. The user may also wish to (and is recommended to) include `defpackage` and `in-package` forms in his system definition files, however, so that they can be loaded manually if need be.

The default value of `*system-definition-search-functions*` is a list of two functions. The first function looks in each of the directories given by evaluating members of `*central-registry*` for a file whose name is the name of the system and whose type is '`asd`'. The first such file is returned, whether or not it turns out to actually

define the appropriate system. The second function does something similar, for the directories specified in the `source-registry`. Hence, it is strongly advised to define a system *foo* in the corresponding file *foo.asd*.

`find-component` *base path* [Function]

Given a *base* component (or designator for such), and a *path*, find the component designated by the *path* starting from the *base*.

If *path* is a component object, it designates itself, independently from the base.

If *path* is a string, or symbol denoting a string via `coerce-name`, then *base* is resolved to a component object, which must be a system or module, and the designated component is the child named by the *path*.

If *path* is a `cons` cell, `find-component` with the base and the `car` of the *path*, and the resulting object is used as the base for a tail call to `find-component` with the `car` of the *path*.

If *base* is a component object, it designates itself.

If *base* is null, then *path* is used as the base, with `nil` as the path.

If *base* is a string, or symbol denoting a string via `coerce-name`, it designates a system as per `find-system`.

If *base* is a `cons` cell, it designates the component found by `find-component` with its `car` as base and `cdr` as path.

6.2.1 Common attributes of components

All components, regardless of type, have the following attributes. All attributes except `name` are optional.

6.2.1.1 Name

A component name is a string or a symbol. If a symbol, its name is taken and lowercased.

Unless overridden by a `:pathname` attribute, the name will be interpreted as a pathname specifier according to a Unix-style syntax. See Section 5.3 [Pathname specifiers], page 13.

6.2.1.2 Version identifier

This optional attribute specifies a version for the current component. The version should typically be a string of integers separated by dots, for example '`1.0.11`'. For more information on version specifiers, see Section 5.3 [The defsystem grammar], page 13.

A version may then be queried by the generic function `version-satisfies`, to see if `:version` dependencies are satisfied, and when specifying dependencies, a constraint of minimal version to satisfy can be specified using e.g. `(:version "mydepname" "1.0.11")`.

Note that in the wild, we typically see version numbering only on components of type `system`. Presumably it is much less useful within a given system, wherein the library author is responsible to keep the various files in synch.

6.2.1.3 Required features

Traditionally defsystem users have used `#+` reader conditionals to include or exclude specific per-implementation files. This means that any single implementation cannot read the entire

system, which becomes a problem if it doesn't wish to compile it, but instead for example to create an archive file containing all the sources, as it will omit to process the system-dependent sources for other systems.

Each component in an asdf system may therefore specify using `:if-feature` a feature expression using the same syntax as `#+` does, such that any reference to the component will be ignored during compilation, loading and/or linking if the expression evaluates to false. Since the expression is read by the normal reader, you must explicitly prefix your symbols with `:` so they be read as keywords; this is as contrasted with the `#+` syntax that implicitly reads symbols in the keyword package by default.

For instance, `:if-feature (:and :x86 (:or :sbcl :cmu :scl))` specifies that the given component is only to be compiled and loaded when the implementation is SBCL, CMUCL or Scieneer CL on an x86 machine. You can not write it as `:if-feature (and x86 (or sbcl cmu scl))` since the symbols would presumably fail to be read as keywords.

6.2.1.4 Dependencies

This attribute specifies dependencies of the component on its siblings. It is optional but often necessary.

There is an excitingly complicated relationship between the initarg and the method that you use to ask about dependencies

Dependencies are between (operation component) pairs. In your initargs for the component, you can say

```
:in-order-to ((compile-op (load-op "a" "b") (compile-op "c"))
              (load-op (load-op "foo")))
```

This means the following things:

- before performing compile-op on this component, we must perform load-op on *a* and *b*, and compile-op on *c*,
- before performing `load-op`, we have to load *foo*

The syntax is approximately

```
(this-op @{(other-op required-components)@}+)

simple-component-name := string
                       |  symbol

required-components := simple-component-name
                     | (required-components required-components)

component-name := simple-component-name
                | (:version simple-component-name minimum-version-object)
```

Side note:

This is on a par with what ACL defsystem does. mk-defsystem is less general: it has an implied dependency

```
for all source file x, (load x) depends on (compile x)
```

and using a `:depends-on` argument to say that *b* depends on *a actually* means that

`(compile b) depends on (load a)`

This is insufficient for e.g. the McCLIM system, which requires that all the files are loaded before any of them can be compiled]

End side note

In ASDF, the dependency information for a given component and operation can be queried using `(component-depends-on operation component)`, which returns a list

```
((load-op "a") (load-op "b") (compile-op "c") ...)
```

`component-depends-on` can be subclassed for more specific component/operation types: these need to `(call-next-method)` and append the answer to their dependency, unless they have a good reason for completely overriding the default dependencies.

If it weren't for CLISP, we'd be using `LIST` method combination to do this transparently. But, we need to support CLISP. If you have the time for some CLISP hacking, I'm sure they'd welcome your fixes.

A minimal version can be specified for a component you depend on (typically another system), by specifying `(:version "other-system" "1.2.3")` instead of simply `"other-system"` as the dependency. See the discussion of the semantics of `:version` in the defsystem grammar.

6.2.1.5 pathname

This attribute is optional and if absent (which is the usual case), the component name will be used.

See Section 5.3 [Pathname specifiers], page 13, for an explanation of how this attribute is interpreted.

Note that the `defsystem` macro (used to create a "top-level" system) does additional processing to set the filesystem location of the top component in that system. This is detailed elsewhere. See Chapter 5 [Defining systems with defsystem], page 11.

6.2.1.6 properties

This attribute is optional.

Packaging systems often require information about files or systems in addition to that specified by ASDF's pre-defined component attributes. Programs that create vendor packages out of ASDF systems therefore have to create "placeholder" information to satisfy these systems. Sometimes the creator of an ASDF system may know the additional information and wish to provide it directly.

`(component-property component property-name)` and associated `setf` method will allow the programmatic update of this information. Property names are compared as if by `EQL`, so use symbols or keywords or something.

6.2.2 Pre-defined subclasses of component

`source-file` [Component]

A source file is any file that the system does not know how to generate from other components of the system.

Note that this is not necessarily the same thing as "a file containing data that is typically fed to a compiler". If a file is generated by some pre-processor stage (e.g.

a '`.h`' file from '`.h.in`' by autoconf) then it is not, by this definition, a source file. Conversely, we might have a graphic file that cannot be automatically regenerated, or a proprietary shared library that we received as a binary: these do count as source files for our purposes.

Subclasses of source-file exist for various languages. *FIXME: describe these.*

`module` [Component]

A module is a collection of sub-components.

A module component has the following extra initargs:

- `:components` the components contained in this module
- `:default-component-class` All children components which don't specify their class explicitly are inferred to be of this type.
- `:if-component-dep-fails` This attribute was removed in ASDF 3. Do not use it. Use `:if-feature` instead.
- `:serial` When this attribute is set, each subcomponent of this component is assumed to depend on all subcomponents before it in the list given to `:components`, i.e. all of them are loaded before a compile or load operation is performed on it.

The default operation knows how to traverse a module, so most operations will not need to provide methods specialised on modules.

`module` may be subclassed to represent components such as foreign-language linked libraries or archive files.

`system` [Component]

`system` is a subclass of `module`.

A system is a module with a few extra attributes for documentation purposes; these are given elsewhere. See Section 5.3 [The defsystem grammar], page 13.

Users can create new classes for their systems: the default `defsystem` macro takes a `:class` keyword argument.

6.2.3 Creating new component types

New component types are defined by subclassing one of the existing component classes and specializing methods on the new component class.

FIXME: this should perhaps be explained more throughly, not only by example ...

As an example, suppose we have some implementation-dependent functionality that we want to isolate in one subdirectory per Lisp implementation our system supports. We create a subclass of `cl-source-file`:

```
(defclass unportable-cl-source-file (cl-source-file)
  ())
```

Function `asdf:implementation-type` (exported since 2.014.14) gives us the name of the subdirectory. All that's left is to define how to calculate the pathname of an `unportable-cl-source-file`.

```
(defmethod component-pathname ((component unportable-cl-source-file))
  (merge-pathnames*
   (parse-unix-namestring (format nil "~(~A~)/" (asdf:implementation-type)))
```

```
    (call-next-method)))
```

The new component type is used in a `defsystem` form in this way:

```
(defsystem :foo
    :components
    ((:file "packages")
     ...
     (:unportable-cl-source-file "threads"
      :depends-on ("packages" ...))
     ...
    )
```

6.3 Functions

version version-spec [version-satisfies]

Does *version* satisfy the *version-spec*. A generic function. ASDF provides built-in methods for *version* being a `component` or `string`. *version-spec* should be a string. If it's a component, its version is extracted as a string before further processing.

A version string satisfies the version-spec if after parsing, the former is no older than the latter. Therefore `"1.9.1"`, `"1.9.2"` and `"1.10"` all satisfy `"1.9.1"`, but `"1.8.4"` or `"1.9"` do not. For more information about how `version-satisfies` parses and interprets version strings and specifications, see Section 5.3 [The defsystem grammar], page 13 (version specifiers) and Section 6.2.1 [Common attributes of components], page 26.

Note that in versions of ASDF prior to 3.0.1, including the entire ASDF 1 and ASDF 2 series, `version-satisfies` would also require that the version and the version-spec have the same major version number (the first integer in the list); if the major version differed, the version would be considered as not matching the spec. But that feature was not documented, therefore presumably not relied upon, whereas it was a nuisance to several users. Starting with ASDF 3.0.1, `version-satisfies` does not treat the major version number specially, and returns T simply if the first argument designates a version that isn't older than the one specified as a second argument. If needs be, the `(:version ...)` syntax for specifying dependencies could be in the future extended to specify an exclusive upper bound for compatible versions as well as an inclusive lower bound.

7 Controlling where ASDF searches for systems

7.1 Configurations

Configurations specify paths where to find system files.

1. The search registry may use some hardcoded wrapping registry specification. This allows some implementations (notably SBCL) to specify where to find some special implementation-provided systems that need to precisely match the version of the implementation itself.
2. An application may explicitly initialize the source-registry configuration using the configuration API (see Chapter 7 [Configuration API], page 31, below) in which case this takes precedence. It may itself compute this configuration from the command-line, from a script, from its own configuration file, etc.
3. The source registry will be configured from the environment variable `CL_SOURCE_REGISTRY` if it exists.
4. The source registry will be configured from user configuration file '`$XDG_CONFIG_DIRS/common-lisp/source-registry.conf`' (which defaults to '`~/.config/common-lisp/source-registry.conf`') if it exists.
5. The source registry will be configured from user configuration directory '`$XDG_CONFIG_DIRS/common-lisp/source-registry.conf.d/`' (which defaults to '`~/.config/common-lisp/source-registry.conf.d/`') if it exists.
6. The source registry will be configured from system configuration file '`/etc/common-lisp/source-registry.conf`' if it exists/
7. The source registry will be configured from system configuration directory '`/etc/common-lisp/source-registry.conf.d/`' if it exists.
8. The source registry will be configured from a default configuration. This configuration may allow for implementation-specific systems to be found, for systems to be found the current directory (at the time that the configuration is initialized) as well as `:directory` entries for '`$XDG_DATA_DIRS/common-lisp/systems/`' and `:tree` entries for '`$XDG_DATA_DIRS/common-lisp/source/`'. For instance, SBCL will include directories for its contribs when it can find them; it will look for them where SBCL was installed, or at the location specified by the `SBCL_HOME` environment variable.

Each of these configurations is specified as an s-expression in a trivial domain-specific language (defined below). Additionally, a more shell-friendly syntax is available for the environment variable (defined yet below).

Each of these configurations is only used if the previous configuration explicitly or implicitly specifies that it includes its inherited configuration.

Additionally, some implementation-specific directories may be automatically prepended to whatever directories are specified in configuration files, no matter if the last one inherits or not.

7.2 Truenames and other dangers

One great innovation of the original ASDF was its ability to leverage `CL:TRUENAME` to locate where your source code was and where to build it, allowing for symlink farms as a simple

but effective configuration mechanism that is easy to control programmatically. ASDF 3 still supports this configuration style, and it is enabled by default; however we recommend you instead use our source-registry configuration mechanism described below, because it is easier to setup in a portable way across users and implementations.

Addtionally, some people dislike truename, either because it is very slow on their system, or because they are using content-addressed storage where the truename of a file is related to a digest of its individual contents, and not to other files in the same intended project. For these people, ASDF 3 allows to eschew the `TRUENAME` mechanism, by setting the variable *asdf:*resolve-symlinks** to `nil`.

PS: Yes, if you haven't read Vernor Vinge's short but great classic "True Names... and Other Dangers" then you're in for a treat.

7.3 XDG base directory

Note that we purport to respect the XDG base directory specification as to where configuration files are located, where data files are located, where output file caches are located. Mentions of XDG variables refer to that document.

`http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html`

This specification allows the user to specify some environment variables to customize how applications behave to his preferences.

On Windows platforms, when not using Cygwin, instead of the XDG base directory specification, we try to use folder configuration from the registry regarding `Common AppData` and similar directories. Since support for querying the Windows registry is not possible to do in reasonable amounts of portable Common Lisp code, ASDF 3 relies on the environment variables that Windows usually exports.

7.4 Backward Compatibility

For backward compatibility as well as to provide a practical backdoor for hackers, ASDF will first search for `.asd` files in the directories specified in `asdf:*central-registry*` before it searches in the source registry above.

See Chapter 3 [Configuring ASDF to find your systems — old style], page 5.

By default, `asdf:*central-registry*` will be empty.

This old mechanism will therefore not affect you if you don't use it, but will take precedence over the new mechanism if you do use it.

7.5 Configuration DSL

Here is the grammar of the s-expression (SEXP) DSL for source-registry configuration:

```
;; A configuration is a single SEXP starting with keyword :source-registry
;; followed by a list of directives.
CONFIGURATION := (:source-registry DIRECTIVE ...)

;; A directive is one of the following:
DIRECTIVE :=
    ;; INHERITANCE DIRECTIVE:
```

```
    ;; Your configuration expression MUST contain
    ;; exactly one of either of these:
    :inherit-configuration | ; splices inherited configuration (often specified last)
    :ignore-inherited-configuration | ; drop inherited configuration (specified anywhe

    ;; forward compatibility directive (since ASDF 2.011.4), useful when
    ;; you want to use new configuration features but have to bootstrap a
    ;; the newer required ASDF from an older release that doesn't sport said features:
    :ignore-invalid-entries | ; drops subsequent invalid entries instead of erroring o

    ;; add a single directory to be scanned (no recursion)
    (:directory DIRECTORY-PATHNAME-DESIGNATOR) |

    ;; add a directory hierarchy, recursing but excluding specified patterns
    (:tree DIRECTORY-PATHNAME-DESIGNATOR) |

    ;; override the defaults for exclusion patterns
    (:exclude EXCLUSION-PATTERN ...) |
    ;; augment the defaults for exclusion patterns
    (:also-exclude EXCLUSION-PATTERN ...) |
    ;; Note that the scope of a an exclude pattern specification is
    ;; the rest of the current configuration expression or file.

    ;; splice the parsed contents of another config file
    (:include REGULAR-FILE-PATHNAME-DESIGNATOR) |

    ;; This directive specifies that some default must be spliced.
    :default-registry

REGULAR-FILE-PATHNAME-DESIGNATOR := PATHNAME-DESIGNATOR ;; interpreted as a file
DIRECTORY-PATHNAME-DESIGNATOR := PATHNAME-DESIGNATOR ;; interpreted as a directory nam

PATHNAME-DESIGNATOR :=
    NIL | ;; Special: skip this entry.
    ABSOLUTE-COMPONENT-DESIGNATOR ;; see pathname DSL

EXCLUSION-PATTERN := a string without wildcards, that will be matched exactly
against the name of a any subdirectory in the directory component
        of a path. e.g. "_darcs" will match `#p"/foo/bar/_darcs/src/bar.asd"'
```

Pathnames are designated using another DSL, shared with the output-translations configuration DSL below. The DSL is resolved by the function `asdf::resolve-location`, to be documented and exported at some point in the future.

```
ABSOLUTE-COMPONENT-DESIGNATOR :=
    (ABSOLUTE-COMPONENT-DESIGNATOR RELATIVE-COMPONENT-DESIGNATOR ...) |
    STRING | ;; namestring (better be absolute or bust, directory assumed where applic
             ;; In output-translations, directory is assumed and **/*.*.* added if it'
```

```
                ;; On MCL, a MacOSX-style POSIX namestring (for MacOS9 style, use #p"..."
                ;; Note that none of the above applies to strings used in *central-regist
                ;; which doesn't use this DSL: they are processed as normal namestrings.
                ;; however, you can compute what you put in the *central-registry*
                ;; based on the results of say (asdf::resolve-location "/Users/fare/cl/cl
        PATHNAME | ;; pathname (better be an absolute path, or bust)
                   ;; In output-translations, unless followed by relative components,
                   ;; it better have appropriate wildcards, as in **/*.*.*
        :HOME | ;; designates the user-homedir-pathname ~/
        :USER-CACHE | ;; designates the default location for the user cache
        :HERE | ;; designates the location of the configuration file
                ;; (or *default-pathname-defaults*, if invoked interactively)
        :ROOT ;; magic, for output-translations source only: paths that are relative
              ;; to the root of the source host and device
        ;; Not valid anymore: :SYSTEM-CACHE (was a security hazard)

    RELATIVE-COMPONENT-DESIGNATOR :=
        (RELATIVE-COMPONENT-DESIGNATOR RELATIVE-COMPONENT-DESIGNATOR ...) |
        STRING | ;; relative directory pathname as interpreted by parse-unix-namestring.
                 ;; In output translations, if last component, **/*.*.* is added
        PATHNAME | ;; pathname; unless last component, directory is assumed.
        :IMPLEMENTATION | ;; directory based on implementation, e.g. sbcl-1.0.45-linux-x64
        :IMPLEMENTATION-TYPE | ;; a directory based on lisp-implementation-type only, e.g.
        :DEFAULT-DIRECTORY | ;; a relativized version of the default directory
        :*/ | ;; any direct subdirectory (since ASDF 2.011.4)
        :**/ | ;; any recursively inferior subdirectory (since ASDF 2.011.4)
        :*.*.* | ;; any file (since ASDF 2.011.4)
        ;; Not supported (anymore): :UID and :USERNAME
```

For instance, as a simple case, my '`~/.config/common-lisp/source-registry.conf`', which is the default place ASDF looks for this configuration, once contained:

```
(:source-registry
  (:tree (:home "cl")) ;; will expand to e.g. "/home/joeluser/cl/"
  :inherit-configuration)
```

7.6 Configuration Directories

Configuration directories consist in files each containing a list of directives without any enclosing `(:source-registry ...)` form. The files will be sorted by namestring as if by `string<` and the lists of directives of these files with be concatenated in order. An implicit `:inherit-configuration` will be included at the *end* of the list.

This allows for packaging software that has file granularity (e.g. Debian's `dpkg` or some future version of `clbuild`) to easily include configuration information about distributed software.

The convention is that, for sorting purposes, the names of files in such a directory begin with two digits that determine the order in which these entries will be read. Also, the type

of these files is conventionally `"conf"` and as a limitation to some implementations (e.g. GNU clisp), the type cannot be `nil`.

Directories may be included by specifying a directory pathname or namestring in an `:include` directive, e.g.:

```
(:include "/foo/bar/")
```

Hence, to achieve the same effect as my example '`~/.config/common-lisp/source-registry.conf`' above, I could simply create a file '`~/.config/common-lisp/source-registry.conf.d/33-home-fare-cl.co`' alone in its directory with the following contents:

```
(:tree "/home/fare/cl/")
```

7.6.1 The :here directive

The `:here` directive is an absolute pathname designator that refers to the directory containing the configuration file currently being processed.

The `:here` directive is intended to simplify the delivery of complex CL systems, and for easy configuration of projects shared through revision control systems, in accordance with our design principle that each participant should be able to provide all and only the information available to him or her.

Consider a person X who has set up the source code repository for a complex project with a master directory '`dir/`'. Ordinarily, one might simply have the user add a directive that would look something like this:

```
(:tree  "path/to/dir")
```

But what if X knows that there are very large subtrees under dir that are filled with, e.g., Java source code, image files for icons, etc.? All of the asdf system definitions are contained in the subdirectories '`dir/src/lisp/`' and '`dir/extlib/lisp/`', and these are the only directories that should be searched.

In this case, X can put into '`dir/`' a file '`asdf.conf`' that contains the following:

```
(:source-registry
   (:tree (:here "src/lisp/"))
   (:tree (:here "extlib/lisp"))
   (:directory (:here "outlier/")))
```

Then when someone else (call her Y) checks out a copy of this repository, she need only add

```
(:include "/path/to/my/checkout/directory/asdf.conf")
```

to one of her previously-existing asdf source location configuration files, or invoke `initialize-source-registry` with a configuration form containing that s-expression. ASDF will find the .conf file that X has provided, and then set up source locations within the working directory according to X's (relative) instructions.

7.7 Shell-friendly syntax for configuration

When considering environment variable `CL_SOURCE_REGISTRY` ASDF will skip to next configuration if it's an empty string. It will `READ` the string as a SEXP in the DSL if it begins with a paren `(` and it will be interpreted much like `TEXINPUTS` list of paths, where

* paths are separated by a `:` (colon) on Unix platforms (including cygwin), by a `;` (semicolon) on other platforms (mainly, Windows).

* each entry is a directory to add to the search path.

* if the entry ends with a double slash `//` then it instead indicates a tree in the subdirectories of which to recurse.

* if the entry is the empty string (which may only appear once), then it indicates that the inherited configuration should be spliced there.

7.8 Search Algorithm

In case that isn't clear, the semantics of the configuration is that when searching for a system of a given name, directives are processed in order.

When looking in a directory, if the system is found, the search succeeds, otherwise it continues.

When looking in a tree, if one system is found, the search succeeds. If multiple systems are found, the consequences are unspecified: the search may succeed with any of the found systems, or an error may be raised. ASDF currently returns the first system found, XCVB currently raised an error. If none is found, the search continues.

Exclude statements specify patterns of subdirectories the systems from which to ignore. Typically you don't want to use copies of files kept by such version control systems as Darcs. Exclude statements are not propagated to further included or inherited configuration files or expressions; instead the defaults are reset around every configuration statement to the default defaults from `asdf::*default-source-registry-exclusions*`.

Include statements cause the search to recurse with the path specifications from the file specified.

An inherit-configuration statement cause the search to recurse with the path specifications from the next configuration (see Chapter 7 [Configurations], page 31 above).

7.9 Caching Results

The implementation is allowed to either eagerly compute the information from the configurations and file system, or to lazily re-compute it every time, or to cache any part of it as it goes. To explicitly flush any information cached by the system, use the API below.

7.10 Configuration API

The specified functions are exported from your build system's package. Thus for ASDF the corresponding functions are in package ASDF, and for XCVB the corresponding functions are in package XCVB.

`initialize-source-registry` *&optionalPARAMETER* [Function]

will read the configuration and initialize all internal variables. You may extend or override configuration from the environment and configuration files with the given *PARAMETER*, which can be `nil` (no configuration override), or a SEXP (in the SEXP DSL), a string (as in the string DSL), a pathname (of a file or directory with configuration), or a symbol (fbound to function that when called returns one of the above).

`clear-source-registry` [Function]
: undoes any source registry configuration and clears any cache for the search algorithm. You might want to call this function (or better, `clear-configuration`) before you dump an image that would be resumed with a different configuration, and return an empty configuration. Note that this does not include clearing information about systems defined in the current image, only about where to look for systems not yet defined.

`ensure-source-registry` *&optionalPARAMETER* [Function]
: checks whether a source registry has been initialized. If not, initialize it with the given *PARAMETER*.

Every time you use ASDF's `find-system`, or anything that uses it (such as `operate`, `load-system`, etc.), `ensure-source-registry` is called with parameter `nil`, which the first time around causes your configuration to be read. If you change a configuration file, you need to explicitly `initialize-source-registry` again, or maybe simply to `clear-source-registry` (or `clear-configuration`) which will cause the initialization to happen next time around.

7.11 Status

This mechanism is vastly successful, and we have declared that `asdf:*central-registry*` is not recommended anymore, though we will continue to support it. All hooks into implementation-specific search mechanisms have been integrated in the `wrapping-source-registry` that everyone uses implicitly.

7.12 Rejected ideas

Alternatives I considered and rejected included:

1. Keep `asdf:*central-registry*` as the master with its current semantics, and somehow the configuration parser expands the new configuration language into a expanded series of directories of subdirectories to lookup, pre-recursing through specified hierarchies. This is kludgy, and leaves little space of future cleanups and extensions.
2. Keep `asdf:*central-registry*` remains the master but extend its semantics in completely new ways, so that new kinds of entries may be implemented as a recursive search, etc. This seems somewhat backwards.
3. Completely remove `asdf:*central-registry*` and break backwards compatibility. Hopefully this will happen in a few years after everyone migrate to a better ASDF and/or to XCVB, but it would be very bad to do it now.
4. Replace `asdf:*central-registry*` by a symbol-macro with appropriate magic when you dereference it or setf it. Only the new variable with new semantics is handled by the new search procedure. Complex and still introduces subtle semantic issues.

I've been suggested the below features, but have rejected them, for the sake of keeping ASDF no more complex than strictly necessary.

- More syntactic sugar: synonyms for the configuration directives, such as `(:add-directory X)` for `(:directory X)`, or `(:add-directory-hierarchy X)` or `(:add-directory X :recurse t)` for `(:tree X)`.

- The possibility to register individual files instead of directories.
- Integrate Xach Beane's tilde expander into the parser, or something similar that is shell-friendly or shell-compatible. I'd rather keep ASDF minimal. But maybe this precisely keeps it minimal by removing the need for evaluated entries that ASDF has? i.e. uses of `USER-HOMEDIR-PATHNAME` and `$SBCL_HOME` Hopefully, these are already superseded by the `:default-registry`
- Using the shell-unfriendly syntax `/**` instead of `//` to specify recursion down a filesystem tree in the environment variable. It isn't that Lisp friendly either.

7.13 TODO

- Add examples

7.14 Credits for the source-registry

Thanks a lot to Stelian Ionescu for the initial idea.

Thanks to Rommel Martinez for the initial implementation attempt.

All bad design ideas and implementation bugs are to mine, not theirs. But so are good design ideas and elegant implementation tricks.

— Francois-Rene Rideau `fare@tunes.org`, Mon, 22 Feb 2010 00:07:33 -0500

8 Controlling where ASDF saves compiled files

Each Common Lisp implementation has its own format for compiled files (fasls for short, short for "fast loading"). If you use multiple implementations (or multiple versions of the same implementation), you'll soon find your source directories littered with various '`fasl`'s, '`dfsl`'s, '`cfsl`'s and so on. Worse yet, some implementations use the same file extension while changing formats from version to version (or platform to platform) which means that you'll have to recompile binaries as you switch from one implementation to the next.

Since ASDF 2, ASDF includes the `asdf-output-translations` facility to mitigate the problem.

8.1 Configurations

Configurations specify mappings from input locations to output locations. Once again we rely on the XDG base directory specification for configuration. See Chapter 7 [XDG base directory], page 31.

1. Some hardcoded wrapping output translations configuration may be used. This allows special output translations (or usually, invariant directories) to be specified corresponding to the similar special entries in the source registry.
2. An application may explicitly initialize the output-translations configuration using the Configuration API in which case this takes precedence. (see Chapter 8 [Configuration API], page 39.) It may itself compute this configuration from the command-line, from a script, from its own configuration file, etc.
3. The source registry will be configured from the environment variable `ASDF_OUTPUT_TRANSLATIONS` if it exists.
4. The source registry will be configured from user configuration file '`$XDG_CONFIG_DIRS/common-lisp/asdf-output-translations.conf`' (which defaults to '`~/.config/common-lisp/asdf-output-translations.conf`') if it exists.
5. The source registry will be configured from user configuration directory '`$XDG_CONFIG_DIRS/common-lisp/asdf-output-translations.conf.d/`' (which defaults to '`~/.config/common-lisp/asdf-output-translations.conf.d/`') if it exists.
6. The source registry will be configured from system configuration file '`/etc/common-lisp/asdf-output-translations.conf`' if it exists.
7. The source registry will be configured from system configuration directory '`/etc/common-lisp/asdf-output-translations.conf.d/`' if it exists.

Each of these configurations is specified as a SEXP in a trival domain-specific language (defined below). Additionally, a more shell-friendly syntax is available for the environment variable (defined yet below).

Each of these configurations is only used if the previous configuration explicitly or implicitly specifies that it includes its inherited configuration.

Note that by default, a per-user cache is used for output files. This allows the seamless use of shared installations of software between several users, and takes files out of the way of the developers when they browse source code, at the expense of taking a small toll when developers have to clean up output files and find they need to get familiar with output-translations first.

8.2 Backward Compatibility

We purposefully do NOT provide backward compatibility with earlier versions of `ASDF-Binary-Locations` (8 Sept 2009), `common-lisp-controller` (7.0) or `cl-launch` (2.35), each of which had similar general capabilities. The previous APIs of these programs were not designed for configuration by the end-user in an easy way with configuration files. Recent versions of same packages use the new `asdf-output-translations` API as defined below: `common-lisp-controller` (7.2) and `cl-launch` (3.000). `ASDF-Binary-Locations` is fully superseded and not to be used anymore.

This incompatibility shouldn't inconvenience many people. Indeed, few people use and customize these packages; these few people are experts who can trivially adapt to the new configuration. Most people are not experts, could not properly configure these features (except inasmuch as the default configuration of `common-lisp-controller` and/or `cl-launch` might have been doing the right thing for some users), and yet will experience software that "just works", as configured by the system distributor, or by default.

Nevertheless, if you are a fan of `ASDF-Binary-Locations`, we provide a limited emulation mode:

`enable-asdf-binary-locations-compatibility` [Function]
&keycentralize-lisp-binaries default-toplevel-directory include-per-user-information map-all-source-files source-to-target-mappings

This function will initialize the new `asdf-output-translations` facility in a way that emulates the behavior of the old `ASDF-Binary-Locations` facility. Where you would previously set global variables **centralize-lisp-binaries**, **default-toplevel-directory**, **include-per-user-information**, **map-all-source-files** or **source-to-target-mappings** you will now have to pass the same values as keyword arguments to this function. Note however that as an extension the `:source-to-target-mappings` keyword argument will accept any valid pathname designator for `asdf-output-translations` instead of just strings and pathnames.

If you insist, you can also keep using the old `ASDF-Binary-Locations` (the one available as an extension to load of top of ASDF, not the one built into a few old versions of ASDF), but first you must disable `asdf-output-translations` with `(asdf:disable-output-translations)`, or you might experience "interesting" issues.

Also, note that output translation is enabled by default. To disable it, use `(asdf:disable-output-translations)`.

8.3 Configuration DSL

Here is the grammar of the SEXP DSL for `asdf-output-translations` configuration:

```
;; A configuration is single SEXP starting with keyword :source-registry
;; followed by a list of directives.
CONFIGURATION := (:output-translations DIRECTIVE ...)

;; A directive is one of the following:
DIRECTIVE :=
    ;; INHERITANCE DIRECTIVE:
    ;; Your configuration expression MUST contain
```

```
    ;; exactly one of either of these:
    :inherit-configuration | ; splices inherited configuration (often specified last)
    :ignore-inherited-configuration | ; drop inherited configuration (specified anywhere)

    ;; forward compatibility directive (since ASDF 2.011.4), useful when
    ;; you want to use new configuration features but have to bootstrap a
    ;; the newer required ASDF from an older release that doesn't sport said features:
    :ignore-invalid-entries | ; drops subsequent invalid entries instead of erroring out

    ;; include a configuration file or directory
    (:include PATHNAME-DESIGNATOR) |

    ;; enable global cache in ~/.common-lisp/cache/sbcl-1.0.45-linux-amd64/ or something.
    :enable-user-cache |
    ;; Disable global cache. Map / to /
    :disable-cache |

    ;; add a single directory to be scanned (no recursion)
    (DIRECTORY-DESIGNATOR DIRECTORY-DESIGNATOR)

    ;; use a function to return the translation of a directory designator
    (DIRECTORY-DESIGNATOR (:function TRANSLATION-FUNCTION))

DIRECTORY-DESIGNATOR :=
    NIL | ;; As source: skip this entry. As destination: same as source
    T | ;; as source matches anything, as destination leaves pathname unmapped.
    ABSOLUTE-COMPONENT-DESIGNATOR ;; same as in the source-registry language

TRANSLATION-FUNCTION :=
    SYMBOL | ;; symbol of a function that takes two arguments,
             ;; the pathname to be translated and the matching DIRECTORY-DESIGNATOR
    LAMBDA   ;; A form which evalutates to a function taking two arguments consisting of
             ;; the pathname to be translated and the matching DIRECTORY-DESIGNATOR
```

Relative components better be either relative or subdirectories of the path before them, or bust.

The last component, if not a pathname, is notionally completed by '`/**/*.*`'. You can specify more fine-grained patterns by using a pathname object as the last component e.g. '`#p"some/path/**/foo*/bar-*.fasl"`'

You may use `#+features` to customize the configuration file.

The second designator of a mapping may be `nil`, indicating that files are not mapped to anything but themselves (same as if the second designator was the same as the first).

When the first designator is `t`, the mapping always matches. When the first designator starts with `:root`, the mapping matches any host and device. In either of these cases, if the second designator isn't `t` and doesn't start with `:root`, then strings indicating the host and

pathname are somehow copied in the beginning of the directory component of the source pathname before it is translated.

When the second designator is `t`, the mapping is the identity. When the second designator starts with `:root`, the mapping preserves the host and device of the original pathname. Notably, this allows you to map files to a subdirectory of the whichever directory the file is in. Though the syntax is not quite as easy to use as we'd like, you can have an (source destination) mapping entry such as follows in your configuration file, or you may use `enable-asdf-binary-locations-compatibility` with `:centralize-lisp-binaries nil` which will do the same thing internally for you:

```
#.(let ((wild-subdir (make-pathname :directory '(:relative :wild-inferiors)))
        (wild-file (make-pathname :name :wild :version :wild :type :wild)))
    `((:root ,wild-subdir ,wild-file) ;; Or using the implicit wildcard, just :root
      (:root ,wild-subdir :implementation ,wild-file)))
```

Starting with ASDF 2.011.4, you can use the simpler: `` `(:root (:root :**/ :implementation :*.*.*)) ``

`:include` statements cause the search to recurse with the path specifications from the file specified.

If the `translate-pathname` mechanism cannot achieve a desired translation, the user may provide a function which provides the required algorithim. Such a translation function is specified by supplying a list as the second `directory-designator` the first element of which is the keyword `:function`, and the second element of which is either a symbol which designates a function or a lambda expression. The function designated by the second argument must take two arguments, the first being the pathname of the source file, the second being the wildcard that was matched. The result of the function invocation should be the translated pathname.

An `:inherit-configuration` statement cause the search to recurse with the path specifications from the next configuration. See Chapter 8 [Configurations], page 39, above.

- `:enable-user-cache` is the same as `(t :user-cache)`.
- `:disable-cache` is the same as `(t t)`.
- `:user-cache` uses the contents of variable `asdf::*user-cache*` which by default is the same as using `(:home ".cache" "common-lisp" :implementation)`.
- `:system-cache` uses the contents of variable `asdf::*system-cache*` which by default is the same as using `("/var/cache/common-lisp" :uid :implementation-type)` (on Unix and cygwin), or something semi-sensible on Windows.

8.4 Configuration Directories

Configuration directories consist in files each contains a list of directives without any enclosing `(:output-translations ...)` form. The files will be sorted by namestring as if by `string<` and the lists of directives of these files with be concatenated in order. An implicit `:inherit-configuration` will be included at the *end* of the list.

This allows for packaging software that has file granularity (e.g. Debian's `dpkg` or some future version of `clbuild`) to easily include configuration information about software being distributed.

The convention is that, for sorting purposes, the names of files in such a directory begin with two digits that determine the order in which these entries will be read. Also, the type of these files is conventionally `"conf"` and as a limitation of some implementations, the type cannot be `nil`.

Directories may be included by specifying a directory pathname or namestring in an `:include` directive, e.g.:

```
(:include "/foo/bar/")
```

8.5 Shell-friendly syntax for configuration

When considering environment variable `ASDF_OUTPUT_TRANSLATIONS` ASDF will skip to next configuration if it's an empty string. It will `READ` the string as an SEXP in the DSL if it begins with a paren `(` and it will be interpreted as a list of directories. Directories should come by pairs, indicating a mapping directive. Entries are separated by a `:` (colon) on Unix platforms (including cygwin), by a `;` (semicolon) on other platforms (mainly, Windows).

The magic empty entry, if it comes in what would otherwise be the first entry in a pair, indicates the splicing of inherited configuration. If it comes as the second entry in a pair, it indicates that the directory specified first is to be left untranslated (which has the same effect as if the directory had been repeated).

8.6 Semantics of Output Translations

From the specified configuration, a list of mappings is extracted in a straightforward way: mappings are collected in order, recursing through included or inherited configuration as specified. To this list is prepended some implementation-specific mappings, and is appended a global default.

The list is then compiled to a mapping table as follows: for each entry, in order, resolve the first designated directory into an actual directory pathname for source locations. If no mapping was specified yet for that location, resolve the second designated directory to an output location directory add a mapping to the table mapping the source location to the output location, and add another mapping from the output location to itself (unless a mapping already exists for the output location).

Based on the table, a mapping function is defined, mapping source pathnames to output pathnames: given a source pathname, locate the longest matching prefix in the source column of the mapping table. Replace that prefix by the corresponding output column in the same row of the table, and return the result. If no match is found, return the source pathname. (A global default mapping the filesystem root to itself may ensure that there will always be a match, with same fall-through semantics).

8.7 Caching Results

The implementation is allowed to either eagerly compute the information from the configurations and file system, or to lazily re-compute it every time, or to cache any part of it as it goes. To explicitly flush any information cached by the system, use the API below.

8.8 Output location API

The specified functions are exported from package ASDF.

`initialize-output-translations` *&optionalPARAMETER* [Function]
: will read the configuration and initialize all internal variables. You may extend or override configuration from the environment and configuration files with the given *PARAMETER*, which can be `nil` (no configuration override), or a SEXP (in the SEXP DSL), a string (as in the string DSL), a pathname (of a file or directory with configuration), or a symbol (fbound to function that when called returns one of the above).

`disable-output-translations` [Function]
: will initialize output translations in a way that maps every pathname to itself, effectively disabling the output translation facility.

`clear-output-translations` [Function]
: undoes any output translation configuration and clears any cache for the mapping algorithm. You might want to call this function (or better, `clear-configuration`) before you dump an image that would be resumed with a different configuration, and return an empty configuration. Note that this does not include clearing information about systems defined in the current image, only about where to look for systems not yet defined.

`ensure-output-translations` *&optionalPARAMETER* [Function]
: checks whether output translations have been initialized. If not, initialize them with the given *PARAMETER*. This function will be called before any attempt to operate on a system.

`apply-output-translations` *PATHNAME* [Function]
: Applies the configured output location translations to *PATHNAME* (calls `ensure-output-translations` for the translations).

Every time you use ASDF's `output-files`, or anything that uses it (that may compile, such as `operate`, `perform`, etc.), `ensure-output-translations` is called with parameter `nil`, which the first time around causes your configuration to be read. If you change a configuration file, you need to explicitly `initialize-output-translations` again, or maybe `clear-output-translations` (or `clear-configuration`), which will cause the initialization to happen next time around.

8.9 Credits for output translations

Thanks a lot to Bjorn Lindberg and Gary King for `ASDF-Binary-Locations`, and to Peter van Eynde for `Common Lisp Controller`.

All bad design ideas and implementation bugs are to mine, not theirs. But so are good design ideas and elegant implementation tricks.

— Francois-Rene Rideau `fare@tunes.org`

9 Error handling

9.1 ASDF errors

If ASDF detects an incorrect system definition, it will signal a generalised instance of `SYSTEM-DEFINITION-ERROR`.

Operations may go wrong (for example when source files contain errors). These are signalled using generalised instances of `OPERATION-ERROR`.

9.2 Compilation error and warning handling

ASDF checks for warnings and errors when a file is compiled. The variables **compile-file-warnings-behaviour** and **compile-file-errors-behavior** control the handling of any such events. The valid values for these variables are `:error`, `:warn`, and `:ignore`.

10 Miscellaneous additional functionality

ASDF includes several additional features that are generally useful for system definition and development.

10.1 Controlling file compilation

When declaring a component (system, module, file), you can specify a keyword argument `:around-compile function`. If left unspecified (and therefore unbound), the value will be inherited from the parent component if any, or with a default of `nil` if no value is specified in any transitive parent.

The argument must be a either `nil`, a fbound symbol, a lambda-expression (e.g. `(lambda (thunk) ...(funcall thunk ...) ...)`) a function object (e.g. using `#.#'` but that's discouraged because it prevents the introspection done by e.g. asdf-dependency-grovel), or a string that when `read` yields a symbol or a lambda-expression. `nil` means the normal compile-file function will be called. A non-nil value designates a function of one argument that will be called with a function that will invoke `compile-file*` with various arguments; the around-compile hook may supply additional keyword arguments to pass to that call to `compile-file*`.

One notable argument that is heeded by `compile-file*` is `:compile-check`, a function called when the compilation was otherwise a success, with the same arguments as `compile-file`; the function shall return true if the compilation and its resulting compiled file respected all system-specific invariants, and false (`nil`) if it broke any of those invariants; it may issue warnings or errors before it returns `nil`. (NB: The ability to pass such extra flags is only available starting with ASDF 2.22.3.) This feature is notably exercised by asdf-finalizers.

By using a string, you may reference a function, symbol and/or package that will only be created later during the build, but isn't yet present at the time the defsystem form is evaluated. However, if your entire system is using such a hook, you may have to explicitly override the hook with `nil` for all the modules and files that are compiled before the hook is defined.

Using this hook, you may achieve such effects as: locally renaming packages, binding **readtables** and other syntax-controlling variables, handling warnings and other conditions, proclaiming consistent optimization settings, saving code coverage information, maintaining meta-data about compilation timings, setting gensym counters and PRNG seeds and other sources of non-determinism, overriding the source-location and/or timestamping systems, checking that some compile-time side-effects were properly balanced, etc.

Note that there is no around-load hook. This is on purpose. Some implementations such as ECL, GCL or MKCL link object files, which allows for no such hook. Other implementations allow for concatenating FASL files, which doesn't allow for such a hook either. We aim to discourage something that's not portable, and has some dubious impact on performance and semantics even when it is possible. Things you might want to do with an around-load hook are better done around-compile, though it may at times require some creativity (see e.g. the `package-renaming` system).

10.2 Controlling source file character encoding

Starting with ASDF 2.21, components accept a `:encoding` option so authors may specify which character encoding should be used to read and evaluate their source code. When left unspecified, the encoding is inherited from the parent module or system; if no encoding is specified at any point, the default `:autodetect` is assumed. By default, only `:default`, `:utf-8` and `:autodetect` are accepted. `:autodetect`, the default, calls `*encoding-detection-hook*` which by default always returns `*default-encoding*` which itself defaults to `:default`.

In other words, there now are plenty of extension hooks, but by default ASDF follows the backwards compatible behavior of using whichever `:default` encoding your implementation uses, which itself may or may not vary based on environment variables and other locale settings. In practice this means that only source code that only uses ASCII is guaranteed to be read the same on all implementations independently from any user setting.

Additionally, for backward-compatibility with older versions of ASDF and/or with implementations that do not support unicode and its many encodings, you may want to use the reader conditionals `#+asdf-unicode #+asdf-unicode` to protect any `:encoding` *`encoding`* statement as `:asdf-unicode` will be present in `*features*` only if you're using a recent ASDF on an implementation that supports unicode. We recommend that you avoid using unprotected `:encoding` specifications until after ASDF 2.21 or later becomes widespread, hopefully by the end of 2012.

While it offers plenty of hooks for extension, and one such extension is being developed (see below), ASDF itself only recognizes one encoding beside `:default`, and that is `:utf-8`, which is the *de facto* standard, already used by the vast majority of libraries that use more than ASCII. On implementations that do not support unicode, the feature `:asdf-unicode` is absent, and the `:default` external-format is used to read even source files declared as `:utf-8`. On these implementations, non-ASCII characters intended to be read as one CL character may thus end up being read as multiple CL characters. In most cases, this shouldn't affect the software's semantics: comments will be skipped just the same, strings with be read and printed with slightly different lengths, symbol names will be accordingly longer, but none of it should matter. But a few systems that actually depend on unicode characters may fail to work properly, or may work in a subtly different way. See for instance `lambda-reader`.

We invite you to embrace UTF-8 as the encoding for non-ASCII characters starting today, even without any explicit specification in your `.asd` files. Indeed, on some implementations and configurations, UTF-8 is already the `:default`, and loading your code may cause errors if it is encoded in anything but UTF-8. Therefore, even with the legacy behavior, non-UTF-8 is guaranteed to break for some users, whereas UTF-8 is pretty much guaranteed not to break anywhere (provided you do *not* use a BOM), although it might be read incorrectly on some implementations. In the future, we intend to make `:utf-8` the default value of `*default-encoding*`, to be enforced everywhere, so at least the code is guaranteed to be read correctly everywhere it can be.

If you need non-standard character encodings for your source code, use the extension system `asdf-encodings`, by specifying `:defsystem-depends-on (:asdf-encodings)` in your `defsystem`. This extension system will register support for more encodings using the `*encoding-external-format-hook*` facility, so you can explicitly specify `:encoding`

`:latin1` in your `.asd` file. Using the `*encoding-detection-hook*` it will also eventually implement some autodetection of a file's encoding from an emacs-style `-*- mode: lisp ; coding: latin1 -*-` declaration, or otherwise based on an analysis of octet patterns in the file. At this point, asdf-encoding only supports the encodings that are supported as part of your implementation. Since the list varies depending on implementations, we once again recommend you use `:utf-8` everywhere, which is the most portable (next is `:latin1`).

If you're not using a version of Quicklisp that has it, you may get the source for `asdf-encodings` using git: *`git clone git://common-lisp.net/projects/asdf/asdf-encodings.git`* or *`git clone ssh://common-lisp.net/project/asdf/git/asdf-encodings.git`*. You can also browse the repository on http://common-lisp.net/gitweb?p=projects/asdf/

In the future, we intend to change the default `*default-encoding*` to `:utf-8`, which is already the de facto standard for most libraries that use non-ASCII characters: utf-8 works everywhere and was backhandedly enforced by a lot of people using SBCL and utf-8 and sending reports to authors so they make their packages compatible. A survey showed only about a handful few libraries are incompatible with non-UTF-8, and then, only in comments, and we believe that authors will adopt UTF-8 when prompted. See the April 2012 discussion on the asdf-devel mailing-list. For backwards compatibility with users who insist on a non-UTF-8 encoding, but cannot immediately transition to using `asdf-encodings` (maybe because it isn't ready), it will still be possible to use the `:encoding :default` option in your `defsystem` form to restore the behavior of ASDF 2.20 and earlier. This shouldn't be required in libraries, because user pressure as mentioned above will already have pushed library authors towards using UTF-8; but authors of end-user programs might care.

When you use `asdf-encodings`, any further loaded `.asd` file will use the autodetection algorithm to determine its encoding; yet if you depend on this detection happening, you may want to explicitly load `asdf-encodings` early in your build, for by the time you can use `:defsystem-depends-on`, it is already too late to load it. In practice, this means that the `*default-encoding*` is usually used for `.asd` files. Currently, this defaults to `:default` for backwards compatibility, and that means that you shouldn't rely on non-ASCII characters in a .asd file. Since component (path)names are the only real data in these files, and non-ASCII characters are not very portable for file names, this isn't too much of an issue. We still encourage you to use either plain ASCII or UTF-8 in `.asd` files, as we intend to make `:utf-8` the default encoding in the future. This might matter, for instance, in meta-data about author's names.

10.3 Miscellaneous Functions

These functions are exported by ASDF for your convenience.

`system-relative-pathname` *system name &keytype* [Function]

It's often handy to locate a file relative to some system. The `system-relative-pathname` function meets this need.

It takes two mandatory arguments *system* and *name* and a keyword argument *type*: *system* is name of a system, whereas *name* and optionally *type* specify a relative pathname, interpreted like a component pathname specifier by `coerce-pathname`. See Section 5.3 [Pathname specifiers], page 13.

It returns a pathname built from the location of the system's source directory and the relative pathname. For example:

```
> (asdf:system-relative-pathname 'cl-ppcre "regex.data")
#P"/repository/other/cl-ppcre/regex.data"
```

`system-source-directory` *system-designator* [Function]

ASDF does not provide a turnkey solution for locating data (or other miscellaneous) files that are distributed together with the source code of a system. Programmers can use `system-source-directory` to find such files. Returns a pathname object. The *system-designator* may be a string, symbol, or ASDF system object.

`clear-system` *system-designator* [Function]

It is sometimes useful to force recompilation of a previously loaded system. In these cases, it may be useful to `(asdf:clear-system :foo)` to remove the system from the table of currently loaded systems; the next time the system `foo` or one that depends on it is re-loaded, `foo` will then be loaded again. Alternatively, you could touch `foo.asd` or remove the corresponding fasls from the output file cache. (It was once conceived that one should provide a list of systems the recompilation of which to force as the `:force` keyword argument to `load-system`; but this has never worked, and though the feature was fixed in ASDF 2.000, it remains `cerror`'ed out as nobody ever used it.)

Note that this does not and cannot by itself undo the previous loading of the system. Common Lisp has no provision for such an operation, and its reliance on irreversible side-effects to global datastructures makes such a thing impossible in the general case. If the software being re-loaded is not conceived with hot upgrade in mind, this re-loading may cause many errors, warnings or subtle silent problems, as packages, generic function signatures, structures, types, macros, constants, etc. are being redefined incompatibly. It is up to the user to make sure that reloading is possible and has the desired effect. In some cases, extreme measures such as recursively deleting packages, unregistering symbols, defining methods on `update-instance-for-redefined-class` and much more are necessary for reloading to happen smoothly. ASDF itself goes through notable pains to make such a hot upgrade possible with respect to its own code, and what it does is ridiculously complex; look at the beginning of '`asdf.lisp`' to see what it does.

`register-preloaded-system` *name &restkeys* [Function]

A system with name *name*, created by `make-instance` with extra keys *keys* (e.g. `:version`), is registered as *preloaded*. That is, its code has already been loaded into the current image, and if at some point some other system `:depends-on` it yet no source code is found, it is considered as already provided, and ASDF will not raise a `missing-component` error.

This function is particularly useful if you distribute your code as fasls with either `fasl-op` or `monolithic-fasl-op`, and want to register systems so that dependencies will work uniformly whether you're using your software from source or from fasl.

`run-shell-command` *control-string &restargs* [Function]

This function is obsolete and present only for the sake of backwards-compatibility: "If it's not backwards, it's not compatible". We *strongly* discourage its use. Its current

behavior is only well-defined on Unix platforms (which include MacOS X and cygwin). On Windows, anything goes. The following documentation is only for the purpose of your migrating away from it in a way that preserves semantics.

Instead we recommend the use `run-program`, described in the next section, and available as part of ASDF since ASDF 3.

`run-shell-command` takes as arguments a format `control-string` and arguments to be passed to `format` after this control-string to produce a string. This string is a command that will be evaluated with a POSIX shell if possible; yet, on Windows, some implementations will use CMD.EXE, while others (like SBCL) will make an attempt at invoking a POSIX shell (and fail if it is not present).

10.4 Some Utility Functions

The below functions are not exported by ASDF itself, but by UIOP, available since ASDF 3. Some of them have precursors in ASDF 2, but we recommend you rely on ASDF 3 for active developments. UIOP provides many, many more utility functions, and we recommend you read its README and sources for more information.

`parse-unix-namestring` *name &keytype defaults dot-dot ensure-directory &allow-other-keys* [Function]

Coerce NAME into a PATHNAME using standard Unix syntax.

Unix syntax is used whether or not the underlying system is Unix; on such non-Unix systems it is only usable but for relative pathnames; but especially to manipulate relative pathnames portably, it is of crucial to possess a portable pathname syntax independent of the underlying OS. This is what `parse-unix-namestring` provides, and why we use it in ASDF.

When given a `pathname` object, just return it untouched. When given `nil`, just return `nil`. When given a non-null `symbol`, first downcase its name and treat it as a string. When given a `string`, portably decompose it into a pathname as below.

`#\/` separates directory components.

The last `#\/`-separated substring is interpreted as follows: 1- If *type* is `:directory` or *ensure-directory* is true, the string is made the last directory component, and its `name` and `type` are `nil`. if the string is empty, it's the empty pathname with all slots `nil`. 2- If *type* is `nil`, the substring is a file-namestring, and its `name` and `type` are separated by `split-name-type`. 3- If *type* is a string, it is the given `type`, and the whole string is the `name`.

Directory components with an empty name the name `.` are removed. Any directory named `..` is read as *dot-dot*, which must be one of `:back` or `:up` and defaults to `:back`.

`host`, `device` and `version` components are taken from *defaults*, which itself defaults to `*nil-pathname*`, also used if *defaults* is `nil`. No host or device can be specified in the string itself, which makes it unsuitable for absolute pathnames outside Unix.

For relative pathnames, these components (and hence the defaults) won't matter if you use `merge-pathnames*` but will matter if you use `merge-pathnames`, which is an important reason to always use `merge-pathnames*`.

Arbitrary keys are accepted, and the parse result is passed to `ensure-pathname` with those keys, removing *type*, *defaults* and *dot-dot*. When you're manipulating pathnames that are supposed to make sense portably even though the OS may not be Unixish, we recommend you use `:want-relative t` to throw an error if the pathname is absolute

`merge-pathnames*` *specified &optionaldefaults* [Function]

This function is a replacement for `merge-pathnames` that uses the host and device from the *defaults* rather than the *specified* pathname when the latter is a relative pathname. This allows ASDF and its users to create and use relative pathnames without having to know beforehand what are the host and device of the absolute pathnames they are relative to.

`subpathname` *pathname subpath &keytype* [Function]

This function takes a *pathname* and a *subpath* and a *type*. If *subpath* is already a `pathname` object (not namestring), and is an absolute pathname at that, it is returned unchanged; otherwise, *subpath* is turned into a relative pathname with given *type* as per `parse-unix-namestring` with `:want-relative t :type` *type*, then it is merged with the `pathname-directory-pathname` of *pathname*, as per `merge-pathnames*`.

We strongly encourage the use of this function for portably resolving relative pathnames in your code base.

`subpathname*` *pathname subpath &keytype* [Function]

This function returns `nil` if the base *pathname* is `nil`, otherwise acts like `subpathname`.

`run-program` *command &keyignore-error-status force-shell input output error-output* [Function]

if-input-does-not-exist if-output-exists if-error-output-exists element-type external-format &allow-other-keys

`run-program` takes a *command* argument that is either a list of a program name or path and its arguments, or a string to be executed by a shell. It spawns the command, waits for it to return, verifies that it exited cleanly (unless told not too below), and optionally captures and processes its output. It accepts many keyword arguments to configure its behavior.

`run-program` returns three values: the first for the output, the second for the error-output, and the third for the return value. (Beware that before ASDF 3.0.2.11, it didn't handle input or error-output, and returned only one value, the one for the output if any handler was specified, or else the exit code; please upgrade ASDF, or at least UIOP, to rely on the new enhanced behavior.)

output is its most important argument; it specifies how the output is captured and processed. If it is `nil`, then the output is redirected to the null device, that will discard it. If it is `:interactive`, then it is inherited from the current process (beware: this may be different from your **standard-output**, and under SLIME will be on your `*inferior-lisp*` buffer). If it is `t`, output goes to your current **standard-output** stream. Otherwise, *output* should be a value that is a suitable first argument to `slurp-input-stream` (see below), or a list of such a value and keyword arguments.

In this case, `run-program` will create a temporary stream for the program output; the program output, in that stream, will be processed by a call to `slurp-input-stream`, using *output* as the first argument (or if it's a list the first element of *output* and the rest as keywords). The primary value resulting from that call (or `nil` if no call was needed) will be the first value returned by `run-program`. E.g., using `:output :string` will have it return the entire output stream as a string. And using `:output '(:string :stripped t)` will have it return the same string stripped of any ending newline.

error-output is similar to *output*, except that the resulting value is returned as the second value of `run-program`. `t` designates the **error-output**. Also `:output` means redirecting the error output to the output stream, in which case `nil` is returned.

input is similar to *output*, except that `vomit-output-stream` is used, no value is returned, and `t` designates the **standard-input**.

`element-type` and `external-format` are passed on to your Lisp implementation, when applicable, for creation of the output stream.

One and only one of the stream slurping or vomiting may or may not happen in parallel in parallel with the subprocess, depending on options and implementation, and with priority being given to output processing. Other streams are completely produced or consumed before or after the subprocess is spawned, using temporary files.

`force-shell` forces evaluation of the command through a shell, even if it was passed as a list rather than a string. If a shell is used, it is '`/bin/sh`' on Unix or '`CMD.EXE`' on Windows, except on implementations that (erroneously, IMNSHO) insist on consulting `$SHELL` like clisp.

`ignore-error-status` causes `run-program` to not raise an error if the spawned program exits in error. Following POSIX convention, an error is anything but a normal exit with status code zero. By default, an error of type `subprocess-error` is raised in this case.

`run-program` works on all platforms supported by ASDF, except Genera. See the source code for more documentation.

`slurp-input-stream` *processor input-stream &key* [Function]

It's a generic function of two arguments, a target object and an input stream, and accepting keyword arguments. Predefined methods based on the target object are as follow:

If the object is a function, the function is called with the stream as argument.

If the object is a cons, its first element is applied to its rest appended by a list of the input stream.

If the object is an output stream, the contents of the input stream are copied to it. If the *linewise* keyword argument is provided, copying happens line by line, and an optional *prefix* is printed before each line. Otherwise, copying happen based on a buffer of size *buffer-size*, using the specified *element-type*.

If the object is `'string` or `:string`, the content is captured into a string. Accepted keywords include the *element-type* and a flag *stripped*, which when true causes any single line ending to be removed as per `uiop:stripln`.

If the object is `:lines`, the content is captured as a list of strings, one per line, without line ending. If the *count* keyword argument is provided, it is a maximum count of lines to be read.

If the object is `:line`, the content is capture as with `:lines` above, and then its sub-object is extracted with the *at* argument, which defaults to `0`, extracting the first line. A number will extract the corresponding line. See the documentation for `uiop:access-at`.

If the object is `:forms`, the content is captured as a list of S-expressions, as read by the Lisp reader. If the *count* argument is provided, it is a maximum count of lines to be read. We recommend you control the syntax with such macro as `uiop:with-safe-io-syntax`.

If the object is `:form`, the content is capture as with `:forms` above, and then its sub-object is extracted with the *at* argument, which defaults to `0`, extracting the first form. A number will extract the corresponding form. See the documentation for `uiop:access-at`. We recommend you control the syntax with such macro as `uiop:with-safe-io-syntax`.

11 Getting the latest version

Decide which version you want. The `master` branch is where development happens; its `HEAD` is usually OK, including the latest fixes and portability tweaks, but an occasional regression may happen despite our (limited) test suite.

The `release` branch is what cautious people should be using; it has usually been tested more, and releases are cut at a point where there isn't any known unresolved issue.

You may get the ASDF source repository using git: *`git clone git://common-lisp.net/projects/asdf/asdf.git`*

You will find the above referenced tags in this repository. You can also browse the repository on http://common-lisp.net/gitweb?p=projects/asdf/asdf.git.

Discussion of ASDF development is conducted on the mailing list *`asdf-devel@common-lisp.net`*. http://common-lisp.net/cgi-bin/mailman/listinfo/asdf-devel

12 FAQ

12.1 "Where do I report a bug?"

ASDF bugs are tracked on launchpad: `https://launchpad.net/asdf`.

If you're unsure about whether something is a bug, or for general discussion, use the asdf-devel mailing list

12.2 "What has changed between ASDF 1 and ASDF 2?"

12.2.1 What are ASDF 1 and ASDF 2?

On May 31st 2010, we have released ASDF 2. ASDF 2 refers to release 2.000 and later. (Releases between 1.656 and 1.728 were development releases for ASDF 2.) ASDF 1 to any release earlier than 1.369 or so. If your ASDF doesn't sport a version, it's an old ASDF 1.

ASDF 2 and its release candidates push `:asdf2` onto `*features*` so that if you are writing ASDF-dependent code you may check for this feature to see if the new API is present. *All* versions of ASDF should have the `:asdf` feature.

Additionally, all versions of ASDF 2 define a function `(asdf:asdf-version)` you may use to query the version; and the source code of recent versions of ASDF 2 features the version number prominently on the second line of its source code.

If you are experiencing problems or limitations of any sort with ASDF 1, we recommend that you should upgrade to ASDF 2, or whatever is the latest release.

12.2.2 ASDF can portably name files in subdirectories

Common Lisp namestrings are not portable, except maybe for logical pathnamestrings, that themselves have various limitations and require a lot of setup that is itself ultimately non-portable.

In ASDF 1, the only portable ways to refer to pathnames inside systems and components were very awkward, using `#.(make-pathname ...)` and `#.(merge-pathnames ...)`. Even the above were themselves were inadequate in the general case due to host and device issues, unless horribly complex patterns were used. Plenty of simple cases that looked portable actually weren't, leading to much confusion and greavance.

ASDF 2 implements its own portable syntax for strings as pathname specifiers. Naming files within a system definition becomes easy and portable again. See Chapter 10 [Miscellaneous additional functionality], page 46, `merge-pathnames*`, `coerce-pathname`.

On the other hand, there are places where systems used to accept namestrings where you must now use an explicit pathname object: `(defsystem ... :pathname "LOGICAL-HOST:PATH;TO;SYSTEM;" ...)` must now be written with the `#p` syntax: `(defsystem ... :pathname #p"LOGICAL-HOST:PATH;TO;SYSTEM;" ...)`

See Section 5.3 [Pathname specifiers], page 13.

12.2.3 Output translations

A popular feature added to ASDF was output pathname translation: `asdf-binary-locations`, `common-lisp-controller`, `cl-launch` and other hacks were all implementing it in ways both mutually incompatible and difficult to configure.

Output pathname translation is essential to share source directories of portable systems across multiple implementations or variants thereof, or source directories of shared installations of systems across multiple users, or combinations of the above.

In ASDF 2, a standard mechanism is provided for that, `asdf-output-translations`, with sensible defaults, adequate configuration languages, a coherent set of configuration files and hooks, and support for non-Unix platforms.

See Chapter 8 [Controlling where ASDF saves compiled files], page 39.

12.2.4 Source Registry Configuration

Configuring ASDF used to require special magic to be applied just at the right moment, between the moment ASDF is loaded and the moment it is used, in a way that is specific to the user, the implementation he is using and the application he is building.

This made for awkward configuration files and startup scripts that could not be shared between users, managed by administrators or packaged by distributions.

ASDF 2 provides a well-documented way to configure ASDF, with sensible defaults, adequate configuration languages, and a coherent set of configuration files and hooks.

We believe it's a vast improvement because it decouples application distribution from library distribution. The application writer can avoid thinking where the libraries are, and the library distributor (dpkg, clbuild, advanced user, etc.) can configure them once and for every application. Yet settings can be easily overridden where needed, so whoever needs control has exactly as much as required.

At the same time, ASDF 2 remains compatible with the old magic you may have in your build scripts (using `*central-registry*` and `*system-definition-search-functions*`) to tailor the ASDF configuration to your build automation needs, and also allows for new magic, simpler and more powerful magic.

See Chapter 7 [Controlling where ASDF searches for systems], page 31.

12.2.5 Usual operations are made easier to the user

In ASDF 1, you had to use the awkward syntax `(asdf:oos 'asdf:load-op :foo)` to load a system, and similarly for `compile-op`, `test-op`.

In ASDF 2, you can use shortcuts for the usual operations: `(asdf:load-system :foo)`, and similarly for `compile-system`, `test-system`.

12.2.6 Many bugs have been fixed

The following issues and many others have been fixed:

- The infamous TRAVERSE function has been revamped completely between ASDF 1 and ASDF 2, with many bugs squashed. In particular, dependencies were not correctly propagated across modules but now are. It has been completely rewritten many times over between ASDF 2.000 and ASDF 3, with fundamental issues in the original model being fixed. Timestamps were not propagated at all, and now are. The internal model of how actions depend on each other is now both consistent and complete. The :version and the :force (system1 .. systemN) feature have been fixed.
- Performance has been notably improved for large systems (say with thousands of components) by using hash-tables instead of linear search, and linear-time list accumulation instead of quadratic-time recursive appends.

- Many features used to not be portable, especially where pathnames were involved. Windows support was notably quirky because of such non-portability.
- The internal test suite used to massively fail on many implementations. While still incomplete, it now fully passes on all implementations supported by the test suite, except for GCL (due to GCL bugs).
- Support was lacking for some implementations. ABCL and GCL were notably wholly broken. ECL extensions were not integrated with ASDF release.
- The documentation was grossly out of date.

12.2.7 ASDF itself is versioned

Between new features, old bugs fixed, and new bugs introduced, there were various releases of ASDF in the wild, and no simple way to check which release had which feature set. People using or writing systems had to either make worst-case assumptions as to what features were available and worked, or take great pains to have the correct version of ASDF installed.

With ASDF 2, we provide a new stable set of working features that everyone can rely on from now on. Use `#+asdf2` to detect presence of ASDF 2, `(asdf:version-satisfies (asdf:asdf-version) "2.345.67")` to check the availability of a version no earlier than required.

12.2.8 ASDF can be upgraded

When an old version of ASDF was loaded, it was very hard to upgrade ASDF in your current image without breaking everything. Instead you had to exit the Lisp process and somehow arrange to start a new one from a simpler image. Something that can't be done from within Lisp, making automation of it difficult, which compounded with difficulty in configuration, made the task quite hard. Yet as we saw before, the task would have been required to not have to live with the worst case or non-portable subset of ASDF features.

With ASDF 2, it is easy to upgrade from ASDF 2 to later versions from within Lisp, and not too hard to upgrade from ASDF 1 to ASDF 2 from within Lisp. We support hot upgrade of ASDF and any breakage is a bug that we will do our best to fix. There are still limitations on upgrade, though, most notably the fact that after you upgrade ASDF, you must also reload or upgrade all ASDF extensions.

12.2.9 Decoupled release cycle

When vendors were releasing their Lisp implementations with ASDF, they had to basically never change version because neither upgrade nor downgrade was possible without breaking something for someone, and no obvious upgrade path was visible and recommendable.

With ASDF 2, upgrade is possible, easy and can be recommended. This means that vendors can safely ship a recent version of ASDF, confident that if a user isn't fully satisfied, he can easily upgrade ASDF and deal with a supported recent version of it. This means that release cycles will be causally decoupled, the practical consequence of which will mean faster convergence towards the latest version for everyone.

12.2.10 Pitfalls of the transition to ASDF 2

The main pitfalls in upgrading to ASDF 2 seem to be related to the output translation mechanism.

- Output translations is enabled by default. This may surprise some users, most of them in pleasant way (we hope), a few of them in an unpleasant way. It is trivial to disable output translations. See Chapter 12 ["How can I wholly disable the compiler output cache?"], page 55.
- Some systems in the large have been known not to play well with output translations. They were relatively easy to fix. Once again, it is also easy to disable output translations, or to override its configuration.
- The new ASDF output translations are incompatible with ASDF-Binary-Locations. They replace A-B-L, and there is compatibility mode to emulate your previous A-B-L configuration. See `enable-asdf-binary-locations-compatibility` in see Chapter 8 [Backward Compatibility], page 39. But thou shalt not load ABL on top of ASDF 2.

Other issues include the following:

- ASDF pathname designators are now specified in places where they were unspecified, and a few small adjustments have to be made to some non-portable defsystems. Notably, in the `:pathname` argument to a `defsystem` and its components, a logical pathname (or implementation-dependent hierarchical pathname) must now be specified with `#p` syntax where the namestring might have previously sufficed; moreover when evaluation is desired `#.` must be used, where it wasn't necessary in the toplevel `:pathname` argument (but necessary in other `:pathname` arguments).
- There is a slight performance bug, notably on SBCL, when initially searching for '`asd`' files, the implicit `(directory "/configured/path/**/*.asd")` for every configured path `(:tree "/configured/path/")` in your `source-registry` configuration can cause a slight pause. Try to `(time (asdf:initialize-source-registry))` to see how bad it is or isn't on your system. If you insist on not having this pause, you can avoid the pause by overriding the default source-registry configuration and not use any deep `:tree` entry but only `:directory` entries or shallow `:tree` entries. Or you can fix your implementation to not be quite that slow when recursing through directories. *Update*: This performance bug fixed the hard way in 2.010.
- On Windows, only LispWorks supports proper default configuration pathnames based on the Windows registry. Other implementations make do with environment variables, that you may have to define yourself if you're using an older version of Windows. Windows support is somewhat less tested than Unix support. Please help report and fix bugs. *Update*: As of ASDF 2.21, all implementations should now use the same proper default configuration pathnames and they should actually work, though they haven't all been tested.
- The mechanism by which one customizes a system so that Lisp files may use a different extension from the default '`.lisp`' has changed. Previously, the pathname for a component was lazily computed when operating on a system, and you would `(defmethod source-file-type ((component cl-source-file) (system (eql (find-system 'foo)))) (declare (ignorable component system)) "lis")`. Now, the pathname for a component is eagerly computed when defining the system, and instead you will `(defclass cl-source-file.lis (cl-source-file) ((type :initform "lis")))` and use `:default-component-class cl-source-file.lis` as argument to `defsystem`, as detailed in a see Chapter 12 [FAQ], page 55 below.

12.3 Issues with installing the proper version of ASDF

12.3.1 "My Common Lisp implementation comes with an outdated version of ASDF. What to do?"

We recommend you upgrade ASDF. See Chapter 2 [Upgrading ASDF], page 2.

If this does not work, it is a bug, and you should report it. See Chapter 12 [Where do I report a bug], page 55. In the meantime, you can load '`asdf.lisp`' directly. See Chapter 2 [Loading ASDF], page 2.

12.3.2 "I'm a Common Lisp implementation vendor. When and how should I upgrade ASDF?"

Since ASDF 2, it should always be a good time to upgrade to a recent version of ASDF. You may consult with the maintainer for which specific version they recommend, but the latest `release` should be correct. We trust you to thoroughly test it with your implementation before you release it. If there are any issues with the current release, it's a bug that you should report upstream and that we will fix ASAP.

As to how to include ASDF, we recommend the following:

- If ASDF isn't loaded yet, then `(require "asdf")` should load the version of ASDF that is bundled with your system. If possible so should `(require "ASDF")`. You may have it load some other version configured by the user, if you allow such configuration.
- If your system provides a mechanism to hook into `CL:REQUIRE`, then it would be nice to add ASDF to this hook the same way that ABCL, CCL, CLISP, CMUCL, ECL, SBCL and SCL do it. Please send us appropriate code to this end.
- You may, like SBCL, have ASDF be implicitly used to require systems that are bundled with your Lisp distribution. If you do have a few magic systems that come with your implementation in a precompiled way such that one should only use the binary version that goes with your distribution, like SBCL does, then you should add them in the beginning of `wrapping-source-registry`.
- If you have magic systems as above, like SBCL does, then we explicitly ask you to *NOT* distribute '`asdf.asd`' as part of those magic systems. You should still include the file '`asdf.lisp`' in your source distribution and precompile it in your binary distribution, but '`asdf.asd`' if included at all, should be secluded from the magic systems, in a separate file hierarchy. Alternatively, you may provide the system after renaming it and its '`.asd`' file to e.g. `asdf-ecl` and '`asdf-ecl.asd`', or `sb-asdf` and '`sb-asdf.asd`'. Indeed, if you made '`asdf.asd`' a magic system, then users would no longer be able to upgrade ASDF using ASDF itself to some version of their preference that they maintain independently from your Lisp distribution.
- If you do not have any such magic systems, or have other non-magic systems that you want to bundle with your implementation, then you may add them to the `wrapping-source-registry`, and you are welcome to include '`asdf.asd`' amongst them. Non-magic systems should be at the back of the `wrapping-source-registry` while magic systems are at the front.
- Please send us upstream any patches you make to ASDF itself, so we can merge them back in for the benefit of your users when they upgrade to the upstream version.

12.4 Issues with configuring ASDF

12.4.1 "How can I customize where fasl files are stored?"

See Chapter 8 [Controlling where ASDF saves compiled files], page 39.

Note that in the past there was an add-on to ASDF called `ASDF-binary-locations`, developed by Gary King. That add-on has been merged into ASDF proper, then superseded by the `asdf-output-translations` facility.

Note that use of `asdf-output-translations` can interfere with one aspect of your systems — if your system uses `*load-truename*` to find files (e.g., if you have some data files stored with your program), then the relocation that this ASDF customization performs is likely to interfere. Use `asdf:system-relative-pathname` to locate a file in the source directory of some system, and use `asdf:apply-output-translations` to locate a file whose pathname has been translated by the facility.

12.4.2 "How can I wholly disable the compiler output cache?"

To permanently disable the compiler output cache for all future runs of ASDF, you can:

```
mkdir -p ~/.config/common-lisp/asdf-output-translations.conf.d/
echo ':disable-cache' > ~/.config/common-lisp/asdf-output-translations.conf.d/99-disab
```

This assumes that you didn't otherwise configure the ASDF files (if you did, edit them again), and don't somehow override the configuration at runtime with a shell variable (see below) or some other runtime command (e.g. some call to `asdf:initialize-output-translations`).

To disable the compiler output cache in Lisp processes run by your current shell, try (assuming `bash` or `zsh`) (on Unix and cygwin only):

```
export ASDF_OUTPUT_TRANSLATIONS=/:
```

To disable the compiler output cache just in the current Lisp process, use (after loading ASDF but before using it):

```
(asdf:disable-output-translations)
```

12.5 Issues with using and extending ASDF to define systems

12.5.1 "How can I cater for unit-testing in my system?"

ASDF provides a predefined test operation, `test-op`. See Section 6.1.1 [Predefined operations of ASDF], page 21. The test operation, however, is largely left to the system definer to specify. `test-op` has been a topic of considerable discussion on the asdf-devel mailing list, and on the launchpad bug-tracker.

Here are some guidelines:

- For a given system, *foo*, you will want to define a corresponding test system, such as *foo-test*. The reason that you will want this separate system is that ASDF does not out of the box supply components that are conditionally loaded. So if you want to have source files (with the test definitions) that will not be loaded except when testing, they should be put elsewhere.

- The *foo-test* system can be defined in an asd file of its own or together with *foo*. An aesthetic preference against cluttering up the filesystem with extra asd files should be balanced against the question of whether one might want to directly load *foo-test*. Typically one would not want to do this except in early stages of debugging.
- Record that testing is implemented by *foo-test*. For example:

```
(defsystem foo
   :in-order-to ((test-op (test-op foo-test)))
   ....)

(defsystem foo-test
   :depends-on (foo my-test-library ...)
   ....)
```

This procedure will allow you to support users who do not wish to install your test framework.

One oddity of ASDF is that `operate` (see Section 6.1 [Operations], page 20) does not return a value. So in current versions of ASDF there is no reliable programmatic means of determining whether or not a set of tests has passed, or which tests have failed. The user must simply read the console output. This limitation has been the subject of much discussion.

12.5.2 "How can I cater for documentation generation in my system?"

The ASDF developers are currently working to add a `doc-op` to the set of predefined ASDF operations. See Section 6.1.1 [Predefined operations of ASDF], page 21. See also https://bugs.launchpad.net/asdf/+bug/479470.

12.5.3 "How can I maintain non-Lisp (e.g. C) source files?"

See `cffi`'s `cffi-grovel`.

12.5.4 "I want to put my module's files at the top level. How do I do this?"

By default, the files contained in an asdf module go in a subdirectory with the same name as the module. However, this can be overridden by adding a `:pathname ""` argument to the module description. For example, here is how it could be done in the spatial-trees ASDF system definition for ASDF 2:

```
(asdf:defsystem :spatial-trees
  :components
  ((:module base
            :pathname ""
            :components
            ((:file "package")
             (:file "basedefs" :depends-on ("package"))
             (:file "rectangles" :depends-on ("package"))))
   (:module tree-impls
            :depends-on (base)
```

```
          :pathname ""
          :components
          ((:file "r-trees")
           (:file "greene-trees" :depends-on ("r-trees"))
           (:file "rstar-trees" :depends-on ("r-trees"))
           (:file "rplus-trees" :depends-on ("r-trees"))
           (:file "x-trees" :depends-on ("r-trees" "rstar-trees"))))
  (:module viz
          :depends-on (base)
          :pathname ""
          :components
          ((:static-file "spatial-tree-viz.lisp")))
  (:module tests
          :depends-on (base)
          :pathname ""
          :components
          ((:static-file "spatial-tree-test.lisp")))
  (:static-file "LICENCE")
  (:static-file "TODO")))
```

All of the files in the `tree-impls` module are at the top level, instead of in a '`tree-impls/`' subdirectory.

Note that the argument to `:pathname` can be either a pathname object or a string. A pathname object can be constructed with the '`#p"foo/bar/"`' syntax, but this is discouraged because the results of parsing a namestring are not portable. A pathname can only be portably constructed with such syntax as `#.(make-pathname :directory '(:relative "foo" "bar"))`, and similarly the current directory can only be portably specified as `#.(make-pathname :directory '(:relative))`. However, as of ASDF 2, you can portably use a string to denote a pathname. The string will be parsed as a `/`-separated path from the current directory, such that the empty string `""` denotes the current directory, and `"foo/bar"` (no trailing `/` required in the case of modules) portably denotes the same subdirectory as above. When files are specified, the last `/`-separated component is interpreted either as the name component of a pathname (if the component class specifies a pathname type), or as a name component plus optional dot-separated type component (if the component class doesn't specifies a pathname type).

12.5.5 How do I create a system definition where all the source files have a .cl extension?

Starting with ASDF 2.014.14, you may just pass the builtin class `cl-source-file.cl` as the `:default-component-class` argument to `defsystem`:

```
(defsystem my-cl-system
  :default-component-class cl-source-file.cl
  ...)
```

Another builtin class `cl-source-file.lsp` is offered for files ending in '`.lsp`'.

If you want to use a different extension for which ASDF doesn't provide builtin support, or want to support versions of ASDF earlier than 2.014.14 (but later than 2.000), you can define a class as follows:

```
;; Prologue: make sure we're using a sane package.
(defpackage :my-asdf-extension
   (:use :asdf :common-lisp)
   (:export #:cl-source-file.lis))
(in-package :my-asdf-extension)

(defclass cl-source-file.lis (cl-source-file)
  ((type :initform "lis")))
```

Then you can use it as follows:

```
(defsystem my-cl-system
  :default-component-class my-asdf-extension:cl-source-file.lis
  ...)
```

Of course, if you're in the same package, e.g. in the same file, you won't need to use the package qualifier before `cl-source-file.lis`. Actually, if all you're doing is defining this class and using it in the same file without other fancy definitions, you might skip package complications:

```
(in-package :asdf)
(defclass cl-source-file.lis (cl-source-file)
   ((type :initform "lis")))
(defsystem my-cl-system
  :default-component-class cl-source-file.lis
  ...)
```

It is possible to achieve the same effect in a way that supports both ASDF 1 and ASDF 2, but really, friends don't let friends use ASDF 1. Please upgrade to ASDF 3. In short, though: do same as above, but *before* you use the class in a `defsystem`, you also define the following method:

```
(defmethod source-file-type ((f cl-source-file.lis) (s system))
  (declare (ignorable f s))
  "lis")
```

13 TODO list

Here is an old list of things to do, in addition to the bugs that are now tracked on launchpad: https://launchpad.net/asdf.

13.1 Outstanding spec questions, things to add

** packaging systems

*** manual page component?

** style guide for .asd files

You should either use keywords or be careful with the package that you evaluate defsystem forms in. Otherwise `(defsystem partition ...)` being read in the `cl-user` package will intern a `cl-user:partition` symbol, which will then collide with the `partition:partition` symbol.

Actually there's a hairier packages problem to think about too. `in-order-to` is not a keyword: if you read `defsystem` forms in a package that doesn't use ASDF, odd things might happen.

** extending defsystem with new options

You might not want to write a whole parser, but just to add options to the existing syntax. Reinstate `parse-option` or something akin.

** Diagnostics

A "dry run" of an operation can be made with the following form:

```
(let ((asdf::*verbose-out* *standard-output*))
  (loop :for (op . comp) :in
    (asdf::traverse (make-instance '<operation-name> :force t)
                    (asdf:find-system <system-name>))
    :do (asdf:explain op comp)))
```

This uses unexported symbols. What would be a nice interface for this functionality?

13.2 Missing bits in implementation

** reuse the same scratch package whenever a system is reloaded from disk

Have a package ASDF-USER instead of all these temporary packages?

** proclamations probably aren't

** A revert function

Other possible interface: have a "revert" function akin to `make clean`.

```
(asdf:revert 'asdf:compile-op 'araneida)
```

would delete any files produced by `(compile-system :araneida)`. Of course, it wouldn't be able to do much about stuff in the image itself.

How would this work?

`traverse`

There's a difference between a module's dependencies (peers) and its components (children). Perhaps there's a similar difference in operations? For example, `(load "use") depends-on (load "macros")` is a peer, whereas `(load "use") depends-on (compile "use")` is more of a "subservient" relationship.

14 Inspiration

14.1 mk-defsystem (defsystem-3.x)

We aim to solve basically the same problems as `mk-defsystem` does. However, our architecture for extensibility better exploits CL language features (and is documented), and we intend to be portable rather than just widely-ported. No slight on the `mk-defsystem` authors and maintainers is intended here; that implementation has the unenviable task of supporting pre-ANSI implementations, which is no longer necessary.

The surface defsystem syntax of asdf is more-or-less compatible with `mk-defsystem`, except that we do not support the `source-foo` and `binary-foo` prefixes for separating source and binary files, and we advise the removal of all options to specify pathnames.

The `mk-defsystem` code for topologically sorting a module's dependency list was very useful.

14.2 defsystem-4 proposal

Marco and Peter's proposal for defsystem 4 served as the driver for many of the features in here. Notable differences are:

- We don't specify output files or output file extensions as part of the system.

  If you want to find out what files an operation would create, ask the operation.
- We don't deal with CL packages

  If you want to compile in a particular package, use an `in-package` form in that file (ilisp / SLIME will like you more if you do this anyway)
- There is no proposal here that `defsystem` does version control.

  A system has a given version which can be used to check dependencies, but that's all.

The defsystem 4 proposal tends to look more at the external features, whereas this one centres on a protocol for system introspection.

14.3 kmp's "The Description of Large Systems", MIT AI Memo 801

Available in updated-for-CL form on the web at http://nhplace.com/kent/Papers/Large-Systems.html

In our implementation we borrow kmp's overall `PROCESS-OPTIONS` and concept to deal with creating component trees from `defsystem` surface syntax. [this is not true right now, though it used to be and probably will be again soon]

Concept Index

Function and Class Index

Variable Index

*

A