ASDF: Another System Definition Facility

Manual for Version 3.1.5

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

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

ASDF Copyright © 2001-2015 Daniel Barlow and contributors.

This manual Copyright © 2001-2015 Daniel Barlow and contributors.

This manual revised © 2009-2015 Robert P. Goldman and Francois-Rene Rideau.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Table of Contents

1 Introduction 1

2 Quick start summary 2

3 Loading ASDF 3
- 3.1 Loading a pre-installed ASDF 3
- 3.2 Checking whether ASDF is loaded 3
- 3.3 Upgrading ASDF 4
- 3.4 Replacing your implementation's ASDF 4
- 3.5 Loading ASDF from source 4

4 Configuring ASDF 5
- 4.1 Configuring ASDF to find your systems 5
- 4.2 Configuring ASDF to find your systems — old style 6
- 4.3 Configuring where ASDF stores object files 7
- 4.4 Resetting the ASDF configuration 7

5 Using ASDF 9
- 5.1 Loading a system 9
- 5.2 Convenience Functions 9
- 5.3 Moving on 10

6 Defining systems with defsystem 11
- 6.1 The defsystem form 11
- 6.2 A more involved example 12
- 6.3 The defsystem grammar 14
  - 6.3.1 Component names 16
  - 6.3.2 Component types 16
  - 6.3.3 System class names 16
  - 6.3.4 Defsystem depends on 16
  - 6.3.5 Weakly depends on 17
  - 6.3.6 Pathname specifiers 17
  - 6.3.7 Version specifiers 18
  - 6.3.8 Require 18
  - 6.3.9 Using logical pathnames 19
  - 6.3.10 Serial dependencies 19
  - 6.3.11 Source location (`:pathname`) 20
  - 6.3.12 if-feature option 20
  - 6.3.13 if-component-dep-fails option 20
  - 6.3.14 feature requirement 21
- 6.4 Other code in .asd files 21
- 6.5 The package-inferred-system extension 21

7 The Object model of ASDF 23

7.1 Operations 23
7.1.1 Predefined operations of ASDF 24
7.1.2 Creating new operations 27
7.2 Components 29
7.2.1 Common attributes of components 32
7.2.1.1 Name 32
7.2.1.2 Version identifier 32
7.2.1.3 Required features 32
7.2.1.4 Dependencies 33
7.2.1.5 pathname 34
7.2.1.6 properties 34
7.2.2 Pre-defined subclasses of component 34
7.2.3 Creating new component types 35
7.3 Dependencies 36
7.4 Functions 36

8 Controlling where ASDF searches for systems 38

8.1 Configurations 38
8.2 Truenames and other dangers 39
8.3 XDG base directory 39
8.4 Backward Compatibility 39
8.5 Configuration DSL 40
8.6 Configuration Directories 42
8.6.1 The :here directive 43
8.7 Shell-friendly syntax for configuration 43
8.8 Search Algorithm 44
8.9 Caching Results 44
8.10 Configuration API 45
8.11 Introspection 46
8.11.1 *source-registry-parameter* variable 46
8.11.2 Information about system dependencies 46
8.12 Status 46
8.13 Rejected ideas 46
8.14 TODO 47
8.15 Credits for the source-registry 47

9 Controlling where ASDF saves compiled files ... 48

9.1 Configurations ... 48
9.2 Backward Compatibility ... 49
9.3 Configuration DSL ... 49
9.4 Configuration Directories ... 52
9.5 Shell-friendly syntax for configuration ... 52
9.6 Semantics of Output Translations ... 52
9.7 Caching Results ... 53
9.8 Output location API ... 53
9.9 Credits for output translations ... 54

10 Error handling ... 55

10.1 ASDF errors ... 55
10.2 Compilation error and warning handling ... 55

11 Miscellaneous additional functionality ... 56

11.1 Controlling file compilation ... 56
11.2 Controlling source file character encoding ... 57
11.3 Miscellaneous Functions ... 58
11.4 Some Utility Functions ... 60

12 Getting the latest version ... 64

13 FAQ ... 65

13.1 "Where do I report a bug?" ... 65
13.2 Mailing list ... 65
13.3 "What has changed between ASDF 1, ASDF 2, and ASDF 3?" ... 65
13.3.1 What are ASDF 1, ASDF 2, and ASDF 3? ... 65
13.3.2 How do I detect the ASDF version? ... 66
13.3.3 ASDF can portably name files in subdirectories ... 66
13.3.4 Output translations ... 67
13.3.5 Source Registry Configuration ... 67
13.3.6 Usual operations are made easier to the user ... 67
13.3.7 Many bugs have been fixed ... 68
13.3.8 ASDF itself is versioned ... 68
13.3.9 ASDF can be upgraded ... 68
13.3.10 Decoupled release cycle ... 69
13.3.11 Pitfalls of the transition to ASDF 2 ... 69
13.3.12 Pitfalls of the upgrade to ASDF 3 ... 70
13.3.13 What happened to the bundle operations? ... 71
13.4 Issues with installing the proper version of ASDF ... 72
13.4.1 "My Common Lisp implementation comes with an outdated version of ASDF. What to do?" ... 72

13.4.2 "I'm a Common Lisp implementation vendor. When and how should I upgrade ASDF?" ... 72
13.5 Issues with configuring ASDF ... 73
13.5.1 "How can I customize where fasl files are stored?" ... 73
13.5.2 "How can I wholly disable the compiler output cache?" ... 73
13.6 Issues with using and extending ASDF to define systems ... 74
13.6.1 "How can I cater for unit-testing in my system?" ... 74
13.6.2 "How can I cater for documentation generation in my system?" ... 74
13.6.3 "How can I maintain non-Lisp (e.g. C) source files?" ... 74
13.6.4 "I want to put my module's files at the top level. How do I do this?" ... 74
13.6.5 How do I create a system definition where all the source files have a .cl extension? ... 75
13.6.6 How do I mark a source file to be loaded only and not compiled? ... 76
13.6.7 How do I work with readtables? ... 76
13.6.7.1 How should my system use a readtable exported by another system? ... 77
13.6.7.2 How should my library make a readtable available to other systems? ... 77
13.7 ASDF development FAQs ... 77
13.7.1 How do run the tests interactively in a REPL? ... 77

Ongoing Work ... 79

Bibliography ... 80

Concept Index ... 81

Function and Class Index ... 82

Variable Index ... 84

1 Introduction

ASDF is Another System Definition Facility: a tool for specifying how systems of Common Lisp software are made up 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. If you are new to ASDF, see Chapter 2 [the quick start guide], page 2.

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, and one for implementers of Common Lisp extensions who want to extend the build system. For more specifics, see Chapter 5 [Using ASDF], page 9, to learn how to use ASDF to load a system. See Chapter 6 [Defining systems with defsystem], page 11, to learn how to define a system of your own. See Chapter 7 [The object model of ASDF], page 23, for a description of the ASDF internals and how to extend ASDF.

Note that ASDF is *not* a tool for library and system *installation*; it plays a role like `make` or `ant`, not like a package manager. In particular, ASDF should not to be confused with Quicklisp or ASDF-Install, that attempt to find and download ASDF systems for you. Despite what the name might suggest, 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 (`http://www.quicklisp.org/`) instead, a Common Lisp package manager which works well 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 (`http://common-lisp.net/project/clbuild/`). We recommend `~/common-lisp/` as a place into which to install Common Lisp software; starting with ASDF 3.1.2, it is included in the default source-registry configuration.

Finally, note that this manual is incomplete. All the bases are covered, but many advanced topics are only barely alluded to, and there is not much in terms of examples. The source code remains the ultimate source of information, free software systems in Quicklisp remain the best source of examples, and the mailing-list the best place to ask for help.

2 Quick start summary

- To load an ASDF system:
  - Load ASDF itself into your Lisp image, using `(require "asdf")`. Check that you have a recent version using `(asdf:asdf-version)`. For more details, or if any of the above fails, see Chapter 3 [Loading ASDF], page 3.
  - Make sure software is installed where ASDF can find it. The simplest way is to put all your Lisp code in subdirectories of `~/common-lisp/` (starting with ASDF 3.1.2), or `~/.local/share/common-lisp/source/` (for ASDF 2 and later, or if you want to keep source in a hidden directory). For more details, see Section 4.1 [Configuring ASDF to find your systems], page 5.
  - Load your system with `(asdf:load-system "`*`my-system`*`")`. See Chapter 5 [Using ASDF], page 9.
- To make your own ASDF system:
  - As above, load and configure ASDF.
  - Make a new directory for your system, *`my-system/`*, again in a location where ASDF can find it. All else being equal, the easiest location is probably `~/common-lisp/my-system/`. See Section 4.1 [Configuring ASDF to find your systems], page 5.
  - Create an ASDF system definition listing the dependencies of your system, its components, and their interdependencies, and put it in *`my-system`*`.asd`. This file must have the same name as your system, all lowercase. See Chapter 6 [Defining systems with defsystem], page 11.
  - Use `(asdf:load-system "`*`my-system`*`")` to make sure it's all working properly. See Chapter 5 [Using ASDF], page 9.

3 Loading ASDF

3.1 Loading a pre-installed ASDF

The recommended way to load ASDF is via:

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

All actively maintained Lisp implementations now include a copy of ASDF 3 that you can load this way using Common Lisp's `require` function.[1]

If the implementation you are using doesn't provide a recent ASDF 3, we recommend you upgrade it. If for some reason you would rather not upgrade it, we recommend you replace your implementation's ASDF. See Section 3.4 [Replacing your implementation's ASDF], page 4. If all else fails, see see Section 3.5 [Loading ASDF from source], page 4 below.

If you use an actively maintained implementation that fails to provide an up-to-date enough stable release of ASDF, you may also send a bug report to your Lisp vendor and complain about it — or you may fix the issue yourself if it's free software.

As of the writing of this manual, the following implementations provide ASDF 3 this way: ABCL, Allegro CL, CLASP, Clozure CL, CMUCL, ECL, GNU CLISP, LispWorks, MKCL, SBCL. The following implementations only provide ASDF 2: MOCL, XCL. The following implementations don't provide ASDF: Corman CL, GCL, Genera, MCL, SCL. The latter implementations are not actively maintained (except maybe GCL); if some of them are ever released again, they probably will include ASDF 3.

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. SLIME notably sports a `slime-asdf` contrib that makes life easier with ASDF.

3.2 Checking whether ASDF is loaded

To check that ASDF is properly loaded, you can run this form:

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

If it returns a string, that is the version of ASDF that is currently installed. If that version is suitably recent (say, 3.1.2 or later), then you can skip directly to next chapter: See Chapter 4 [Configuring ASDF], page 5.

If it raises an error, then either ASDF is not loaded, or you are using a very old version of ASDF, and need to install ASDF 3.

For more precision in detecting versions old and new, see Section 13.3.2 [How do I detect the ASDF version?], page 66.

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.

---

[1] NB: all implementations except GNU CLISP also accept `(require "ASDF")`, `(require 'asdf)` and `(require :asdf)`. For portability's sake, you should use `(require "asdf")`.

3.3 Upgrading ASDF

If your implementation already provides ASDF 3 or later (and it should), but you want a more recent ASDF version than your implementation provides, then you just need to ensure the more recent ASDF is installed in a configured path, like any other system. We recommend you download an official tarball or checkout a release from git into `~/common-lisp/asdf/`. (see Section 4.1 [Configuring ASDF to find your systems], page 5).

Once the source code for ASDF is installed, you don't need any extra step to load it beyond the usual `(require "asdf")`: ASDF 3 will automatically look whether an updated version of itself is available amongst the regularly configured systems, before it compiles anything else.

If your implementation fails to provide ASDF 3 or later, see Section 3.4 [Replacing your implementation's ASDF], page 4.

3.4 Replacing your implementation's ASDF

All maintained implementations now provide ASDF 3 in their latest release. If your doesn't, we recommend you upgrade it.

Now, if you insist on using an old implementation that didn't provide ASDF or provided an old version, we recommend installing a recent ASDF, as explained below, into your implementation's installation directory. Thus your modified implementation will now provide ASDF 3. This requires proper write permissions and may necessitate execution as a system administrator.

The ASDF source repository contains a tool to help you upgrade your implementation's ASDF. You can invoke it from the shell command-line as `tools/asdf-tools install-asdf lispworks` (where you can replace `lispworks` by the name of the relevant implementation), or you can `(load "tools/install-asdf.lisp")` from your Lisp REPL.

This script works on Allegro CL, Clozure CL, CMU CL, ECL, GCL, GNU CLISP, LispWorks, MKCL, SBCL, SCL, XCL. It doesn't work on ABCL, Corman CL, Genera, MCL, MOCL. Happily, ABCL is usually pretty up to date and shouldn't need that script. GCL requires a very recent version, and hasn't been tested much. Corman CL, Genera, MCL are obsolete anyway. MOCL is incomplete.

3.5 Loading ASDF from source

If you write build scripts that must remain portable to old machines with old implementations that you cannot ensure have been upgraded or modified to provide a recent ASDF, you may 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 12 [Getting the latest version], page 64.

For scripts that try to use ASDF simply via `require` at first, and make heroic attempts to load it the hard way if at first they don't succeed, see `tools/load-asdf.lisp` distributed with the ASDF source repository, or the code of `cl-launch`.

4 Configuring ASDF

For standard use cases, ASDF should work pretty much out of the box. We recommend you skim the sections on configuring ASDF to find your systems and choose the method of installing Lisp software that works best for you. Then skip directly to See Chapter 5 [Using ASDF], page 9. That will probably be enough. You are unlikely to have to worry about the way ASDF stores object files, and resetting the ASDF configuration is usually only needed in corner cases.

4.1 Configuring ASDF to find your systems

In order to compile and load your systems, ASDF must be configured to find the `.asd` files that contain system definitions.

There are a number of different techniques for setting yourself up with ASDF, starting from easiest to the most complex:

- Put all of your systems in one of the standard locations, subdirectories of
  - `~/common-lisp/` or
  - `~/.local/share/common-lisp/source/`.

  If you install software there, you don't need further configuration.[1] You can then skip to the next section. See Section 5.1 [Loading a system], page 9.
- If you're using some tool to install software (e.g. Quicklisp), the authors of that tool should already have configured ASDF.
- If you have more specific desires about how to lay out your software on disk, 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 8 [Controlling where ASDF searches for systems], page 38. Here is a quick recipe for getting started.

  First create the directory `~/.config/common-lisp/source-registry.conf.d/`[2]; there create a file with any name of your choice but with the type `conf`[3], for instance `50-luser-lisp.conf`; in this file, add the following line to tell ASDF to recursively scan all the subdirectories under `/home/luser/lisp/` for `.asd` files: `(:tree "/home/luser/lisp/")`

  That's enough. You may replace `/home/luser/lisp/` by wherever you want to install your source code. You don't actually need to specify anything if you use the default `~/common-lisp/` as above and your implementation provides ASDF 3.1.2 or later. If

---

[1] `~/common-lisp/` is only included in the default configuration starting with ASDF 3.1.2 or later. If your implementation provides an earlier variant of ASDF, you may need to explicitly configure it to use this path, as further explained.

[2] For Windows users, and starting with ASDF 3.1.5, start from your `%LOCALAPPDATA%`, which is usually `~/AppData/Local/` (but you can ask in a `CMD.EXE` terminal `echo %LOCALAPPDATA%` to make sure) and underneath create a subpath `config/common-lisp/source-registry.conf.d/`.

[3] By requiring the `.conf` extension, and ignoring other files, ASDF allows you to have disabled files, editor backups, etc. in the same directory with your active configuration files.

ASDF will also ignore files whose names start with a `.` character.

It is customary to start the filename with two digits, to control the sorting of the `conf` files in the source registry directory, and thus the order in which the directories will be scanned.

your implementation provides an earlier variant of ASDF 3, you might want to specify `(:tree (:home "common-lisp/"))` for bootstrap purposes, then install a recent source tree of ASDF under `~/common-lisp/asdf/`.

If you prefer to use a "link farm", which is faster to use but costlier to manage than a recursive traversal, say at `/home/luser/.asd-link-farm/`, then you may instead (or additionally) create a file `42-asd-link-farm.conf`, containing the line: `(:directory "/home/luser/.asd-link-farm/")`

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

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

- In earlier versions of ASDF, the system source registry was configured using a global variable, `asdf:*central-registry*`. For more details about this, see the following section, Section 4.2 [Configuring ASDF to find your systems — old style], page 6. Unless you need to understand this, skip directly to Section 4.3 [Configuring where ASDF stores object files], page 7.

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

4.2 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.[4]

For example, let's say you want ASDF to find the `.asd` file `/home/me/src/foo/foo.asd`. In your lisp initialization file, you could have the following:

```
(require "asdf")
(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.[5] The trailing directory name

[4] 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 6 [Defining systems with defsystem], page 11.

[5] ASDF will indeed call `eval` on each entry. It will 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 s-expressions onto the central registry. These s-expressions may be evaluated to compute context-dependent entries, 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

separator is necessary to tell Lisp that you're discussing a directory rather than a file. If you leave it out, ASDF is likely to look in `/home/me/src/` instead of `/home/me/src/foo/` as you intended, and fail to find your system definition.

Typically there are a lot of `.asd` files, and a common idiom was to put *symbolic links* to all of one's `.asd` files in a common directory and push *that* directory (the "link farm") onto `asdf:*central-registry*`, instead of pushing each individual system directory.

ASDF knows to follow *symlinks* to the actual location of the systems.[6]

For example, if `#p"/home/me/cl/systems/"` is an element of `*central-registry*`, you could set up the system *foo* as follows:

```
$ 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 a simple way to programmatically control what directories are added to the ASDF search path.

4.3 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 to be shared between several versions of several Common Lisp implementations, between several users using different compilation options, with users who lack write privileges on shared source directories, etc. This also keeps source directories from being cluttered with object/fasl files.

Starting with ASDF 2, the `asdf-output-translations` facility was added to ASDF itself. This facility controls where object files will be stored. This facility is fully described in a chapter of this manual, Chapter 9 [Controlling where ASDF saves compiled files], page 48.

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 delegate object file placement to ASDF.

4.4 Resetting the ASDF 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.

---

evaluate to a directory to look in (or `NIL`). By "directory", we mean "designator for a pathname with a non-empty DIRECTORY component".

[6] 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.

This function is pushed onto the `uiop:*image-dump-hook*` by default, which means that if you save an image using `uiop:dump-image`, or via `asdf:image-op` and `asdf:program-op`, it will be automatically called to clear your configuration. If for some reason you prefer to call your implementation's underlying functionality, be sure to call `clear-configuration` manually, or push it into your implementation's equivalent of `uiop:*image-dump-hook*`, e.g. `sb-ext:*save-hooks*` on SBCL, or `ext:*before-save-initializations*` on CMUCL and SCL, etc.

5 Using ASDF

5.1 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 (see Section 5.2 [Convenience Functions], page 9), ASDF hooks into the `cl:require` facility and you can just use:

```
(require :foo)
```

Note that the canonical name of a system is a string, conventionally lowercase. A system name can also be specified as a symbol (including a keyword), in which case its `symbol-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. The type of systems we want to support either have lowercase as customary case (Unix, Mac, Windows) or silently convert lowercase to uppercase (lpns).

5.2 Convenience Functions

ASDF provides three commands for the most common system operations: `load-system`, `compile-system`, and `test-system`. It also provides `require-system`, a version of `load-system` that skips trying to update systems that are already loaded. And it provides `make`, a function that uses whichever operation was specified by the author of the target system, which by default behaves like `load-system`.

Because ASDF is an extensible system for defining *operations* on *components*, it also provides a generic function `operate`, so you may arbitrarily operate on your systems beyond the default operations. (At the interactive REPL, users often use its shorter alias `oos`, which stands for operate-on-system, a name inherited from `mk-defsystem`.) You'll use `operate` whenever you want to do something beyond compiling, loading and testing.

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

`load-system` applies `operate` with the operation from `*load-system-operation*` the system, and any provided keyword arguments. `*load-system-operation*` by default is `load-op`; it would be `load-bundle-op` by default on ECL, if only an implementation bug were fixed.

`require-system` skips any update to systems that have already been loaded, in the spirit of `cl:require`. It does it by calling `load-system` with a keyword option excluding already loaded systems.[1]. On actively maintained free software implementations (namely recent versions of ABCL, Clozure CL, CMUCL, ECL, GNU CLISP, MKCL and SBCL), once ASDF itself is loaded, `cl:require` too can load ASDF systems, by falling back on `require-system` for module names not recognized by the implementation.

[1] For the curious, the option is `:force-not (already-loaded-systems)`.

Note that `cl:require` and `require-system` are appropriate to load code that is not being modified during the current programming session. This notably includes the implementation-provided extension modules that `cl:require` can load. This also includes any number of ASDF systems that the user isn't either developing or debugging, for which a previously installed version is deemed to be satisfactory. `require-system` and through it `cl:require` can load these systems without any problem. But for code that you are actively developing, debugging, or otherwise modifying, you should use `load-system`, so ASDF will pick on your modifications and transitively re-build the modified files and everything that depends on them.

Finally, starting with ASDF 3.1, a function `make` is also available, that does "The Right Thing" with your system. The default behaviour is to load the system as if by `load-system`; but instead of this default, system authors can specify the intended use of their system by specifying their desired operation with a `:build-operation` argument in the system definition.

5.3 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.

6 Defining systems with defsystem

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

6.1 The 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 should be saved as `hello-lisp.asd` (in order that ASDF can find it when ordered to operate on the system named `"hello-lisp"`).

```
;; Usual Lisp comments are allowed here

(defsystem "hello-lisp"
  :description "hello-lisp: a sample Lisp system."
  :version "0.0.1"
  :author "Joe User <joe@example.com>"
  :licence "Public Domain"
  :depends-on ("optima.ppcre" "command-line-arguments")
  :components ((:file "packages")
               (:file "macros" :depends-on ("packages"))
               (:file "hello" :depends-on ("macros"))))
```

Some notes about this example:

- The `defsystem` form defines a system named `hello-lisp` that contains three source files: `packages.lisp`, `macros.lisp` and `hello.lisp`.
- The `.lisp` suffix is implicit for Lisp source files. The source files are located in the same directory as the `.asd` file with the system definition.
- 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` then `macros` before starting the compilation of file `hello`.
- This example system has external dependencies on two other systems, `optima.ppcre` (that provides a friendly interface to matching regular expressions), and `command-line-arguments` (that provides a way to parse arguments passed from the shell command line). To use this system, ASDF must be configured to find installed copies of these systems; it will load them before it tries to compile and load `hello-lisp`.
- This system also defines a bunch of metadata. While it is optional to define these fields (and other fields like `:bug-tracker`, `:mailto`, `:long-name`, `:long-description`, `:source-control`), it is strongly recommended to define the fields `:description`, `:version`, `:author`, and `:licence`, especially if you intend your software to be eventually included in Quicklisp.
- Make sure you know how the `:version` numbers will be parsed! Only period-separated non-negative integers are accepted at present. See below Version specifiers in Section 6.3 [The defsystem grammar], page 14.

- This file contains a single `defsystem` declaration. More complex system definition files are possible with arbitrary Lisp code, but we recommend that you keep it simple if you can. This will make your system definitions more robust and more future-proof.

This is all you need to know to define simple systems. The next example is much more involved, to give you a glimpse of how you can do more complex things. However, since it's ultimately arbitrary Lisp code, there is no bottom to the rabbit hole.

6.2 A more involved example

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

```
(in-package :asdf-user)

(defsystem "foo"
  :version (:read-file-form "variables" :at (3 2))
  :components
  ((:file "package")
   (:file "variables" :depends-on ("package"))
   (:module "mod"
     :depends-on ("package")
     :serial t
     :components ((:file "utils")
                  (:file "reader")
                  (:file "cooker")
                  (:static-file "data.raw"))
     :output-files (compile-op (o c) (list "data.cooked"))
     :perform (compile-op :after (o c)
        (cook-data
         :in (component-pathname (find-component c "data.raw"))
         :out (first (output-files o c)))))
   (:file "foo" :depends-on ("mod"))))

(defmethod action-description
    ((o compile-op) (c (eql (find-component "foo" "mod"))))
  "cooking data")
```

Here are some notes about this example:

- The main thing this file does is define a system `foo`. It also contains other Lisp forms, which we'll examine below.
- Besides Lisp source files, this system contains a `:module` component named `"mod"`, which is a collection of three Lisp source files `utils.lisp`, `reader.lisp`, `cooker.lisp` and `data.raw`
- Note that the `:static-file` does not have an implicit file type, unlike the Lisp source files.
- This files will be located in a subdirectory of the main code directory named `mod/` (this location could have been overridden to be in the same directory, or in a different subdirectory; see the discussion of the `:pathname` option in Section 6.3 [The defsystem grammar], page 14).

- The `:serial t` says that each sub-component of `mod` depends on the previous components, so that `cooker.lisp` depends-on `utils.lisp`, which depends-on `reader.lisp`. Also `data.raw` depends on all of them, but that doesn't matter since it's a static file; on the other hand, if it appeared first, then all the Lisp files would be recompiled when the data is modified, which is probably not what is desired in this case.
- The method-form tokens provide a shorthand for defining methods on particular components. This part

```
:output-files (compile-op (o c) (list "data.cooked"))
:perform (compile-op :after (o c)
   (cook-data
    :in (component-pathname (find-component c "data.raw"))
    :out (first (output-files o c))))
```

  has the effect of

```
(defmethod output-files ((o compile-op) (c (eql ...)))
  (list "data.cooked"))
(defmethod perform :after ((o compile-op) (c (eql ...)))
  (cook-data
   :in (component-pathname (find-component c "data.raw"))
   :out (first (output-files o 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 6.3 [The defsystem grammar], page 14. For more details on what these methods do, see Section 7.1 [Operations], page 23 in Chapter 7 [The object model of ASDF], page 23.
- There is an additional `defmethod` with a similar effect, because ASDF (as of ASDF 3.1.5) fails to accept inline-methods as above for `action-description`, instead only supporting the deprecated `explain` interface.
- In this case, these methods describe how this module defines code that it then uses to cook some data.
- Importantly, ASDF is told about the input and output files used by the data cooker, and to make sure everyone agrees, the cooking function explicitly uses ASDF to access pathnames to the input and output data.
- The file starts with a form `(in-package :asdf-user)`, but it is actually redundant, not necessary and not recommended. But yet more complex cases (also not recommended) may usefully use an `in-package` form.
- Indeed, ASDF does not load `.asd` files simply with `cl:load`, and neither should you. You should let ASDF find and load them when you operate on systems. If you somehow *must* load a `.asd` file, use the same function `asdf:load-asd` that ASDF uses. Among other things, it already binds the `*package*` to `asdf-user`. Recent versions of SLIME (2013-02 and later) know to do that when you *C-c C-k* when you use the `slime-asdf` contrib.
- You shouldn't use an `in-package` form if you're keeping things simple. You should only use `in-package` (and before it, a `defpackage`) when you're going to define new

classes, functions, variables, macros, etc., in the `.asd` file, and want to thereby avoid name clashes. Manuals for old versions of ASDF recommended use of such an idiom in `.asd` files, but as of ASDF 3, we recommend that you don't do that anymore, and instead define any ASDF extensions in their own system, on which you can then declare a dependency using `:defsystem-depends-on`. See Section 6.3 [The defsystem grammar], page 14.

- More generally, you can always rely on symbols from packages `asdf`, `common-lisp` and `uiop` being available in `.asd` files — most importantly including `defsystem`. It is therefore redundant and in bad taste to use a package-prefixed `asdf:defsystem` symbol in a `.asd` file. Just use `(defsystem ...)`. Only package-prefix it when somehow dynamically generating system definitions from a package that doesn't already use the ASDF package.
- `asdf-user` is actually only available starting since ASDF 3, but then again, ASDF 1 and 2 did crazy things with packages that ASDF 3 has stopped doing[1], and since all implementations provide ASDF 3, you shouldn't care about compatibility with ASDF 2. We do not support ASDF 2 anymore, and we recommend that neither should you.
- Starting with ASDF 3.1, `asdf-user` uses `uiop`, whereas in earlier variants of ASDF 3 it only used `uiop/package`. We recommend you either prefix use of UIOP functions with the package prefix `uiop:`, or make sure your system `:depends-on ((:version "asdf" "3.1.2"))` or has a `#-asdf3.1 (error "MY-SYSTEM requires ASDF 3.1.2")`.
- Finally, we elided most metadata, but showed how you can have ASDF automatically extract the system's version from a source file. In this case, the 3rd subform of the 4th form (note that Lisp uses 0-based indexing, English uses 1-based indexing). Presumably, the 4th form looks like `(defparameter *foo-version* "5.6.7")`.

6.3 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)
                 | system-option
                 | module-option
                 | option

# These are only available since ASDF 3 (actually its alpha release 2.27)
system-option := :homepage string
                 | :bug-tracker string
                 | :mailto string
```

[1] ASDF 1 and 2 (up until 2.26) used to dynamically create and delete temporary packages `asdfN`, one for each `.asd` file, in a misguided attempt to thereby reduce name clashes; but it failed at that goal and only made things more complex. ASDF 3 just uses a shared package `asdf-user` instead, and relies on the usual Common Lisp conventions to avoid clashes. As far as package oddities go, you may just notice that the `asdf-user` package also uses `uiop/common-lisp`, a variant of the `common-lisp` package that papers over deficiencies in more obscure Common Lisp implementations; but unless you care about Corman Lisp, GCL, Genera or MCL, you shouldn't be concerned.

```
                | :long-name string
                | :source-control source-control
                | :version version-specifier

source-control := (keyword string)

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 6.3 [Component types], page 14)

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

# ``dependency'' is used in :in-order-to, as opposed to
# ``dependency-def''
dependency := (dependent-op requirement+)
requirement := (required-op required-component+)
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)
```

6.3.1 Component names

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

6.3.2 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.

6.3.3 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.

6.3.4 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.

6.3.5 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 behaviour; 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 behaviour may be removed without warning.

6.3.6 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 interprets the string `".."` as the pathname directory component word `:back`, which when merged, goes back one level in the directory hierarchy.

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

behaviour of `parse-namestring` is completely non-portable, unless you are using Common Lisp `logical-pathname`s, which themselves involve other non-portable behaviour (see Section 6.3 [Using logical pathnames], page 14, 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 used not to 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.

6.3.7 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.

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; subforms can also be specified, with e.g. `(1 2 2)` specifying "the third subform (index 2) of the third subform (index 2) of the second form (index 1)" in the file (mind the off-by-one error in the English language).

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 7.2.1 [Common attributes of components], page 32.

6.3.8 Require

Use the implementation's own `require` to load the *module-name*.

It is good taste to use `:if-feature` *`:implementation-name`* rather than `#+`*`implementation-name`* to only depend on the specified module on the specific implementation that provides it. See [if-feature-option], page 20.

6.3.9 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.

6.3.10 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")))
```

6.3.11 Source location (`:pathname`)

The `:pathname` option is optional in all cases for systems defined via `defsystem`, and generally is unnecessary. In the simple case, source files will be found in the same directory as the system or, in the case of modules, in a subdirectory with the same name as the module.

More specifically, 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)

6.3.12 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, and that is accessible to outside code that wishes to reason about system structure.

Programmers should be careful to consider **when** the `:if-feature` is evaluated. Recall that ASDF first computes a build plan, and then executes that plan. ASDF will check to see whether or not a feature is present **at planning time**, not during the build. It follows that one cannot use `:if-feature` to check features that are set during the course of the build. It can only be used to check the state of features before any build operations have been performed.

This option was added in ASDF 3. For more information, See [required-features], page 32.

6.3.13 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.

6.3.14 feature requirement

This requirement was removed in ASDF 3.1. Please do not use it. In most cases, `:if-feature` (see [if-feature-option], page 20) will provide an adequate substitute.

The `feature` requirement used to ensure that a chain of component dependencies would fail when a key feature was absent. Used in conjunction with `:if-component-dep-fails` this provided a roundabout way to express conditional compilation.

6.4 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.

6.5 The package-inferred-system extension

Starting with release 3.1.2, ASDF supports a one-package-per-file style of programming, whereby each file is its own system, and dependencies are deduced from the `defpackage` form (or its variant `uiop:define-package`).

In this style, packages refer to a system with the same name (downcased); and if a system is defined with `:class package-inferred-system`, then system names that start with that name (using the slash `/` separator) refer to files under the filesystem hierarchy where the system is defined. For instance, if system `my-lib` is defined in `/foo/bar/my-lib/my-lib.asd`, then system `my-lib/src/utility` will be found in file `/foo/bar/my-lib/src/utility.lisp`.

This style was made popular by `faslpath` and `quick-build` before, and at the cost of a stricter package discipline, seems to make for more maintainable code. It is used by ASDF itself (starting with ASDF 3), by `lisp-interface-library`, and a few other libraries.

To use this style, choose a toplevel system name, e.g. `my-lib`, and create a file `my-lib.asd` with the `:class :package-inferred-system` option in its `defsystem`. For instance:

```
#-asdf3.1 (error "my-lib requires ASDF 3.1")
(defsystem "my-lib"
  :class :package-inferred-system
  :depends-on ("my-lib/interface/all"
               "my-lib/src/all"
               "my-lib/extras/all")
  :in-order-to ((test-op (load-op "my-lib/test/all")))
  :perform (test-op (o c) (symbol-call :my-lib/test/all :test-suite)))
```

```
(defsystem "my-lib/test" :depends-on ("my-lib/test/all"))

(register-system-packages "my-lib/interface/all" '(:my-lib-interface))
(register-system-packages "my-lib/src/all" '(:my-lib-implementation))
(register-system-packages "my-lib/test/all" '(:my-lib-test))

(register-system-packages
 "closer-mop"
 '(:c2mop :closer-common-lisp :c2cl :closer-common-lisp-user :c2cl-user))
```

In the code above, the first line checks that we are using ASDF 3.1, which provides `package-inferred-system`.

The function `register-system-packages` has to be called to register packages used or provided by your system and its components where the name of the system that provides the package is not the downcase of the package name.

Then, file `interface/order.lisp` under the `lil` hierarchy, that defines abstract interfaces for order comparisons, starts with the following form, dependencies being trivially computed from the `:use` and `:mix` clauses:

```
(uiop:define-package :lil/interface/order
  (:use :closer-common-lisp
   :lil/interface/definition
   :lil/interface/base
   :lil/interface/eq :lil/interface/group)
  (:mix :fare-utils :uiop :alexandria)
  (:export ...))
```

ASDF can tell that this file depends on system `closer-mop` (registered above), `lil/interface/definition`, `lil/interface/base`, `lil/interface/eq`, and `lil/interface/group` (package and system names match, and they will be looked up hierarchically).

ASDF also detects dependencies from `:import-from` clauses. You may thus import a well-defined set of symbols from an existing package as loaded from suitably named system; or if you prefer to use any such symbol fully qualified by a package prefix, you may declare a dependency on such a package and its corresponding system via an `:import-from` clause with an empty list of symbols, as in:

```
(defpackage :foo/bar
  (:use :cl)
  (:import-from :foo/baz #:sym1 #:sym2)
  (:import-from :foo/quux)
  (:export ...))
```

The form `uiop:define-package` is supported as well as `defpackage`, and has many options that prove useful in this context, such as `:use-reexport` and `:mix-reexport` that allow for "inheritance" of symbols being exported.

7 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, allowing programmers to add new behaviours to ASDF. For example, `cffi` adds support for special FFI description files that interface with C libraries and for wrapper files that embed C code in Lisp. `abcl-jar` supports creating Java JAR archives in ABCL. `poiu` supports compiling code in parallel using background processes.

The key classes in ASDF are `component` and `operation`. A `component` represents an individual source file or a group of source files, and the products (e.g., fasl files) produced from it. An `operation` represents a transformation that can be performed on a component, turning them from source files to intermediate results to final outputs. Components are related by *dependencies*, specified in system definitions.

When ordered to `operate` with some operation on a component (usually a system), ASDF will first compute a *plan* by traversing the dependency graph using function `make-plan`.[1] The resulting plan object contains an ordered list of *actions*. An action is a pair of an `operation` and a `component`, representing a particular build step to be `perform`ed. The ordering of the plan ensures that no action is performed before all its dependencies have been fulfilled.[2]

In this chapter, we describe ASDF's object-oriented protocol, the classes that make it up, and the generic functions on those classes. These generic functions often take both an operation and a component as arguments: much of the power and configurability of ASDF is provided by this use of CLOS's multiple dispatch. We will describe the built-in component and operation classes, and explain how to extend the ASDF protocol by defining new classes and methods for ASDF's generic functions. We will also describe the many *hooks* that can be configured to customize the behaviour of existing *functions*.

7.1 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. Operations are invoked on systems via `operate` (see [operate], page 24).

ASDF contains a number of pre-defined `operation` classes for common, and even fairly uncommon tasks that you might want to do with it. In addition, ASDF contains "abstract"

[1] Historically, the function that built a plan was called `traverse`, and returned a list of actions; it was deprecated in favor of `make-plan` (that returns a plan object) when the `plan` objects were introduced; the old function is kept for backward compatibility and debugging purposes only.

[2] The term *action* was used by Kent Pitman in his article, "The Description of Large Systems," (see [Bibliography], page 80). Although the term was only used by ASDF hackers starting with ASDF 2, the concept was there since the very beginning of ASDF 1, just not clearly articulated.

`operation` classes that programmers can use as building blocks to define ASDF extensions. We discuss these in turn below.

Operations are invoked on systems via `operate`.

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

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

`operate` invokes *operation* on *system*. `oos` is a synonym for `operate` (it stands for operate-on-system).

operation is a symbol that is passed, along with the supplied *initargs*, to `make-operation` (which will call `make-instance`) to create the operation object. *component* is a component designator, usually a string or symbol that designates a system, sometimes a list of strings or symbols that designate a subcomponent of a system.

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 all systems but the system being loaded are forced not to be recompiled even if modified since last compilation (note: this was changed in ASDF 3.1.2). 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.

Both *force* and *force-not* apply to systems that are dependencies and were already compiled. *force-not* takes precedences over *force*, as it should, really, but unhappily only since ASDF 3.1.2. Moreover, systems which have been registered as immutable by `register-immutable-system` (since ASDF 3.1.5) are always considered *forced-not*, and even their `.asd` are not refreshed from the filesystem. See Section 11.3 [Miscellaneous Functions], page 58.

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

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

7.1.1 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.

`load-op` will recursively load all the children of a system or module.

`load-op` also 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 the compiled fasl output. It has a `prepare-source-op` analog to `prepare-op`, that ensures the dependencies are themselves loaded via `load-source-op`.

`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. Its `operation-done-p` method returns `nil`, which means 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, and given the fact that ASDF operations do not return a value indicating success or failure. For those willing to go to the effort, we suggest defining conditions to signal when a `test-op` fails, and storing in those conditions information that describes which tests fail.

People typically define a separate test *system* to hold the tests. Doing this avoids unnecessarily adding a test framework as a dependency on a library. For example, one might have

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

(defsystem "foo/test"
  :depends-on ("foo" "fiveam") ; fiveam is a test framework library
  ...)
```

Then one defines `perform` methods on `test-op` such as the following:

```
(defsystem "foo/test"
  :depends-on ("foo" "fiveam") ; fiveam is a test framework library
  :perform (test-op (o s)
                    (uiop:symbol-call :fiveam '#:run!
                       (uiop:find-symbol* '#:foo-test-suite
                                          :foo-tests)))
  ...)
```

`compile-bundle-op`, `monolithic-compile-bundle-op`, `load-bundle-op`, `monolithic-load-bundle-op`, `deliver-asd-op`, `monolithic-deliver-asd-op`, `lib-op`, `monolithic-lib-op`, `dll-op`, `monolithic-dll-op`, `image-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.

`compile-bundle-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-compile-bundle-op` will create a single fasl file for the target system and all its dependencies, so you can deliver your entire application as a single fasl. `load-bundle-op` will load the output of `compile-bundle-op`. Note that if it the output is not up-to-date, `compile-bundle-op` may load the intermediate fasls as a side-effect. Bundling fasls together 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.

NB: `compile-bundle-op`, `monolithic-compile-bundle-op`, `load-bundle-op`, `monolithic-load-bundle-op`, `deliver-asd-op`, `monolithic-deliver-asd-op` were respectively called `fasl-op`, `monolithic-fasl-op`, `load-fasl-op`, `monolithic-load-fasl-op`, `binary-op`, `monolithic-binary-op` before ASDF 3.1. The old names still exist for backward compatibility, though they poorly label what is going on.

Once you have created a fasl with `compile-bundle-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 `deliver-asd-op` to let ASDF create such a system for you as well as the `compile-bundle-op` output, or `monolithic-deliver-asd-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`. `image-op` will dump an image that may not be standalone and does not start its own function, but follows the usual execution convention of the underlying Lisp, just with more code pre-loaded, for use as an intermediate build result or with a wrapper invocation script.

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 behaviour 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`, [Operation]
`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`

These operations, as their respective names indicate, will concatenate all the `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 load the result, or compile and then load 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 and then loads properly.

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

7.1.2 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 `input-files` and `output-files` (see below) to locate its inputs and outputs, 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 mul-

tiple ultimate input files. Most operations inherit from `selfward-operation`, which appropriately sets the input-files to include the source file itself.

`input-files` *operation component* [Function]
Return a list of pathnames that represent the input to *operation* performed on *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.

7.2 Components

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, that can be found via `find-system`. 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.

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, they are implicitly loaded into the `ASDF-USER` package, which uses `ASDF`, `UIOP` and `UIOP/COMMON-LISP`[3] Programmers who do anything non-trivial in a `.asd` file, such as defining new variables, functions or classes, should include `defpackage` and `in-package` forms in this file, so they will not overwrite each others' extensions. Such forms might also help the files behave identically if loaded manually with `cl:load` for development or debugging, though we recommend you use the function `asdf::load-asd` instead, which the `slime-asdf` contrib knows about.

The default value of `*system-definition-search-functions*` is a list of three 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`, but searches the filesystem only once and caches its results. The third function makes the `package-inferred-system` extension work, see Section 6.5 [The package-inferred-system extension], page 21.

Because of the way these search functions are defined, you should put the definition for a system *foo* in a file named `foo.asd`, in a directory that is in the central registry or which can be found using the source registry configuration.

It is often useful to define multiple systems in a same file, but ASDF can only locate a system's definition file based on the system name. For this reason, ASDF 3's system search algorithm has been extended to allow a file `foo.asd` to contain secondary systems named *foo/bar*, *foo/baz*, *foo/quux*, etc., in addition to the primary system named *foo*. The first component of a system name, separated by the slash character, `/`, is called the primary name of a system. The primary name may be extracted by function `asdf::primary-system-name`; when ASDF 3 is told to find a system whose name has a slash, it will first attempt to load the corresponding primary system, and will thus see any such definitions, and/or any definition of a `package-inferred-system`.[4] If your file `foo.asd` also defines systems that do not follow this convention, e.g., a system named *foo-test*, ASDF will not be able to automatically locate a definition for these systems, and will only see their definition if you explicitly find or load the primary system using e.g. `(asdf:find-system "foo")` before you try to use them. We strongly recommend against this practice, though it is currently supported for backward compatibility.

`primary-system-name` *name* [Function]

Internal (not exported) function, `asdf::primary-system-name`. Returns the primary system name (the portion before the slash, `/`, in a secondary system name) from *name*.

---

[3] Note that between releases 2.27 and 3.0.3, only `UIOP/PACKAGE`, not all of `UIOP`, was used; if you want your code to work with releases earlier than 3.1.2, you may have to explicitly define a package that uses `UIOP`, or use proper package prefix to your symbols, as in `uiop:version<`.

[4] ASDF 2.26 and earlier versions do not support this primary system name convention. With these versions of ASDF you must explicitly load `foo.asd` before you can use system *foo/bar* defined therein, e.g. using `(asdf:find-system "foo")`. We do not support ASDF 2, and recommend that you should upgrade to ASDF 3.

`locate-system` *name* [Function]

This function should typically *not* be invoked directly. It is exported as part of the API only for programmers who wish to provide their own `*system-definition-search-functions*`.

Given a system *name* designator, try to locate where to load the system definition from. Returns five values: *foundp*, *found-system*, *pathname*, *previous*, and *previous-time*. *foundp* is true when a system was found, either a new as yet unregistered one, or a previously registered one. The *found-system* return value will be a `system` object, if a system definition is found in your source registry. The system definition will *not* be loaded if it hasn't been loaded already. *pathname* when not null is a path from which to load the system, either associated with *found-system*, or with the *previous* system. If *previous* is not null, it will be a *previously loaded* `system` object of the same name (note that the system *definition* is previously-loaded: the system itself may or may not be). *previous-time* when not null is the timestamp of the previous system definition file, at the time when the *previous* system definition was loaded.

For example, if your current registry has `foo.asd` in `/current/path/to/foo.asd`, but system `foo` was previously loaded from `/previous/path/to/foo.asd` then *locate-system* will return the following values:

1. *foundp* will be `T`,
2. *found-system* will be `NIL`,
3. *pathname* will be `#p"/current/path/to/foo.asd"`,
4. *previous* will be an object of type `SYSTEM` with `system-source-file` slot value of `#p"/previous/path/to/foo.asd"`
5. *previous-time* will be the timestamp of `#p"/previous/path/to/foo.asd"` at the time it was loaded.

`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.

7.2.1 Common attributes of components

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

7.2.1.1 Name

A component name is a string or a symbol. If a symbol, its name is taken and lowercased. This translation is performed by the exported function `coerce-name`.

Unless overridden by a `:pathname` attribute, the name will be interpreted as a pathname specifier according to a Unix-style syntax. See Section 6.3 [Pathname specifiers], page 14.

7.2.1.2 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 6.3 [The defsystem grammar], page 14.

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.

7.2.1.3 Required features

Traditionally defsystem users have used `#+` reader conditionals to include or exclude specific per-implementation files. For example, CFFI, the portable C foreign function interface contained lines like:

```
#+sbcl       (:file "cffi-sbcl")
```

An unfortunate side effect of this approach is that no single implementation can read the entire system. This causes problems if, for example, one wished to design an `archive-op` that would create an archive file containing all the sources, since for example the file `cffi-sbcl.lisp` above would be invisible when running the `archive-op` on any implementation other than SBCL.

Starting with ASDF 3, components may therefore have an `:if-feature` option. The value of this option should be a feature expression using the same syntax as `#+` does. If that feature expression evaluates to false, any reference to the component will be ignored during compilation, loading and/or linking. 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 cannot write it as `:if-feature (and x86 (or sbcl cmu scl))` since the symbols would not be read as keywords.

See [if-feature-option], page 20.

7.2.1.4 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.

7.2.1.5 pathname

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

See Section 6.3 [Pathname specifiers], page 14, 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 6 [Defining systems with defsystem], page 11.

To find the CL pathname corresponding to a component, use

`component-pathname` *component* [Function]

Returns the pathname corresponding to *component*. For components such as source files, this will be a filename pathname. For example:

```
CL-USER> (asdf:component-pathname (asdf:find-system "xmls"))
#P"/Users/rpg/lisp/xmls/"
```

and

```
CL-USER> (asdf:component-pathname
            (asdf:find-component
               (asdf:find-system "xmls") "xmls"))
#P"/Users/rpg/lisp/xmls/xmls.lisp"
```

7.2.1.6 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.

7.2.2 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 (see [required-features], page 32, and see [if-feature-option], page 20).
- `: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 6.3 [The defsystem grammar], page 14.

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

7.2.3 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.

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" ...))
     ...
    )
```

7.3 Dependencies

To be successfully build-able, this graph of actions must be acyclic. If, as a user, extender or implementer of ASDF, you introduce a cycle into the dependency graph, ASDF will fail loudly. 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-bundle-op` remain at the system level, and are not propagated along the hierarchy, but instead do something global on the system.

7.4 Functions

`version-satisfies` *`version version-spec`* [Function]

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 6.3 [The defsystem grammar], page 14 (version specifiers) and Section 7.2.1 [Common attributes of components], page 32.

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.

8 Controlling where ASDF searches for systems

8.1 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 8 [Configuration API], page 38, 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 default user configuration trees `~/common-lisp/` (since ASDF 3.1.2 only), `~/.sbcl/systems/` (on SBCL only), `$XDG_DATA_HOME/common-lisp/systems/` (no recursion, link farm) `$XDG_DATA_HOME/common-lisp/source/`. The `XDG_DATA_HOME` directory defaults to `~/.local/share/`. On Windows, the `local-appdata` and `appdata` directories are used instead.
7. The source registry will be configured from system configuration file `/etc/common-lisp/source-registry.conf` if it exists.
8. The source registry will be configured from system configuration directory `/etc/common-lisp/source-registry.conf.d/` if it exists.
9. 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/`, where `XDG_DATA_DIRS` defaults to `/usr/local/share` and `/usr/share` on Unix, and the `common-appdata` directory on Windows.
10. The source registry may include implementation-dependent directories that correspond to implementation-provided extensions.

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.

8.2 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.

Additionally, 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.

8.3 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, even when not using Cygwin, and starting with ASDF 3.1.5, we still do a best effort at following the XDG base directory specification, even though it doesn't exactly fit common practice for Windows applications. However, we replace the fixed Unix paths `~/.local`, `/usr/local` and `/usr` with their rough Windows equivalent `Local AppData`, `AppData`, `Common AppData`, etc. 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, and are hopefully in synch with the Windows registry. If you care about the details, see `uiop/configuration.lisp` and don't hesitate to suggest improvements.

8.4 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 4 [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.

8.5 Configuration DSL

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

```
;; A configuration is a single SEXP starting with the 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 the following:
    :inherit-configuration |
    ;; splices inherited configuration (often specified last) or
    :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
    ;; the newer required ASDF from an older release that doesn't
    ;; support said features:
    :ignore-invalid-entries |

    ;; 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

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
    ;; applicable).  In output-translations, directory is assumed and
    ;; **/*.*.* added if it's last.  On MCL, a MacOSX-style POSIX
    ;; namestring (for MacOS9 style, use #p"..."); Note that none of the
    ;; above applies to strings used in *central-registry*, 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-foo/")
    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

They keyword :SYSTEM-CACHE is not accepted in ASDF 3.1 and beyond: it
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. sbcl
      :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)

    The keywords :UID and :USERNAME are no longer supported.
```

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)
```

8.6 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.

System-wide or per-user Common Lisp software distributions such as Debian packages or some future version of `clbuild` may then include files such as `/etc/common-lisp/source-registry.conf.d/10-foo.conf` or `~/.config/common-lisp/source-registry.conf.d/10-foo.conf` to easily and modularly register 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 must be `.conf`, which not only simplifies the implementation by allowing for more portable techniques in finding those files, but also makes it trivial to disable a file, by renaming it to a different file type.

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.conf` alone in its directory with the following contents:

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

8.6.1 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.

8.7 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 `(`, otherwise it will be interpreted much like `TEXINPUTS`, as a 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.

8.8 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 Section 8.1 [Configurations], page 38 above).

8.9 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. In practice, the recommended `source-registry` eagerly collects and caches results and you need to explicitly flush the cache for change to be taken into account, whereas the old-style `*central-registry*` mechanism queries the filesystem every time.

To explicitly flush any information cached by the system after a change was made in the filesystem, See Section 8.10 [Configuration API], page 45, and e.g. call `asdf:clear-source-registry`.

Starting with ASDF 3.1.4, you can also explicitly build a persistent cache of the `.asd` files found under a tree: when recursing into a directory declared by `:tree` and its transitive subdirectories, if a file `.cl-source-registry.cache` exists containing a form that is a list starting with `:source-registry-cache` followed by a list of strings, as in `(:source-registry-cache "`*`foo/bar.asd`*`" "`*`path/to/more.asd`*`" ...)`, then the strings are assumed to be `unix-namestring`s designating the available asd files under that tree, and the recursion otherwise stops. The list can also be empty, allowing to stop a costly recursion in a huge directory tree.

To update such a cache after you install, update or remove source repositories, you can run a script distributed with ASDF: `tools/cl-source-registry-cache.lisp` *`/path/to/directory`*. To wholly invalidate the cache, you can delete the file `.cl-source-registry.cache` in that directory. In either case, for an existing Lisp process to see this change, it needs to clear its own cache with e.g. `(asdf:clear-source-registry)`.

Developers may safely create a cache in their development tree, and we recommend they do it at the top of their source tree if it contains more than a small number of files and directories; they only need update it when they create, remove or move `.asd` files.

Software distribution managers may also safely create such a cache, but they must be careful to update it every time they install, update or remove a software source repository or installation package. Finally, advanced developers who juggle with a lot of code in their `source-registry` may manually manage such a cache, to allow for faster startup of Lisp programs.

This persistence cache can help you reduce startup latency. For instance, on one machine with hundreds of source repositories, such a cache shaves half a second at the startup of every `#!/usr/bin/cl` script using SBCL, more on other implementations; this makes a notable difference as to their subjective interactivity and usability. The speedup will only happen if the implementation-provided ASDF is recent enough (3.1.3.7 or later); it is not enough for a recent ASDF upgrade to be present, since the upgrade will itself be found but after the old version has scanned the directories without heeding such a cache. To upgrade the implementation-provided ASDF, use our script `tools/install-asdf.lisp`.

8.10 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` *&optional PARAMETER* [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` *&optional PARAMETER* [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.

8.11 Introspection

8.11.1 *source-registry-parameter* variable

We have made available the variable `*source-registry-parameter*` that can be used by code that wishes to introspect about the (past) configuration of ASDF's source registry. **This variable should never be set!** It will be set as a side-effect of calling `initialize-source-registry`; user code should treat it as read-only.

8.11.2 Information about system dependencies

ASDF makes available three functions to read system interdependencies. These are intended to aid programmers who wish to perform dependency analyses.

`system-defsystem-depends-on` *system* [Function]

`system-depends-on` *system* [Function]

`system-weakly-depends-on` *system* [Function]

Returns a list of names of systems that are weakly depended on by *system*. Weakly depended on systems are optionally loaded only if ASDF can find them; failure to find such systems does *not* cause an error in loading.

Note that the return value for `system-weakly-depends-on` is simpler than the return values of the other two system dependency introspection functions.

8.12 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.

8.13 Rejected ideas

Alternatives I (FRR) considered and rejected while developing ASDF 2 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*` as 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 TEXINPUTS-like `//` to specify recursion down a filesystem tree in the environment variable. It isn't that Lisp friendly either.

8.14 TODO

- Add examples

8.15 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 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

9 Controlling where ASDF saves compiled files

Each Common Lisp implementation has its own format for compiled files or fasls.[1] If you use multiple implementations (or multiple versions of the same implementation), you'll soon find your source directories littered with various `fasls`, `dfsls`, `cfsls` and so on. Worse yet, multiple implementations use the same file extension and some implementations maintain the same file extension while changing formats from version to version (or platform to platform). This can lead to many errors and much confusion as you switch from one implementation to the next. Finally, this requires write access to the source directory, and therefore precludes sharing of a same source code directory between multiple users.

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

9.1 Configurations

Configurations specify mappings from input locations to output locations. Once again we rely on the XDG base directory specification for configuration. See Chapter 8 [XDG base directory], page 38.

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 9 [Configuration API], page 48.) 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 trivial domain-specific language (see Section 8.5 [Configuration DSL], page 40). Additionally, a more shell-friendly syntax is available for the environment variable (see Section 8.7 [Shell-friendly syntax for configuration], page 43).

When processing an entry in the above list of configuration methods, ASDF will stop unless that entry explicitly or implicitly specifies that it includes its inherited configuration.

[1] "FASL" is short for "FASt Loading."

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.[2]

9.2 Backward Compatibility

We purposely 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 APIs of these programs were not designed for easy user configuration through configuration files. Recent versions of `common-lisp-controller` (7.2) and `cl-launch` (3.000) use the new `asdf-output-translations` API as defined below. `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` *&key centralize-lisp-binaries default-toplevel-directory include-per-user-information map-all-source-files source-to-target-mappings* [Function]

This function will initialize the new `asdf-output-translations` facility in a way that emulates the behaviour 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)`.

9.3 Configuration DSL

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

[2] A `CLEAN-OP` would be a partial solution to this problem.

```
;; 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 have
    ;; said features:
    :ignore-invalid-entries |

    ;; 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
        ; maps pathname to itself.
    ABSOLUTE-COMPONENT-DESIGNATOR ; same as in the source-registry language

TRANSLATION-FUNCTION :=
    SYMBOL | ;; symbol naming a function that takes two arguments:
             ;; the pathname to be translated and the matching
             ;; DIRECTORY-DESIGNATOR
    LAMBDA   ;; A form which evaluates to a function taking two arguments:
             ;; 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)
      (: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 algorithm. 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. When invoked, the function should return the translated pathname.

An `:inherit-configuration` statement causes the search to recurse with the path specifications from the next configuration in the bulleted list. See Chapter 9 [Configurations], page 48, 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)`.

9.4 Configuration Directories

Configuration directories consist of files, each of which 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.

System-wide or per-user Common Lisp software distributions such as Debian packages or some future version of `clbuild` may then include files such as `/etc/common-lisp/asdf-output-translations.conf.d/10-foo.conf` or `~/.config/common-lisp/asdf-output-translations.conf.d/10-foo.conf` to easily and modularly register 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 must be `.conf`, which not only simplifies the implementation by allowing for more portable techniques in finding those files, but also makes it trivial to disable a file, by renaming it to a different file type.

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

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

9.5 Shell-friendly syntax for configuration

When considering environment variable `ASDF_OUTPUT_TRANSLATIONS` ASDF will skip to the 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). Thus `"/foo:/bar::/baz:"` means that things under directory `/foo/` are translated to be under `/bar/`, then include the inherited configuration, then specify that things under directory `/baz/` are not translated.

9.6 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).

9.7 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.

9.8 Output location API

The specified functions are exported from package ASDF.

`initialize-output-translations` *&optional PARAMETER* [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` *&optional PARAMETER* [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.

9.9 Credits for output translations

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

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`

10 Error handling

10.1 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`.

10.2 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-failure-behaviour** control the handling of any such events. The valid values for these variables are `:error`, `:warn`, and `:ignore`.

11 Miscellaneous additional functionality

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

11.1 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 either `nil`, an 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).

11.2 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, or if `nil` is explicitly specified, an extensible protocol described below is followed, that ultimately defaults to `:utf-8` since ASDF 3.

The protocol to determine the encoding is to call the function `detect-encoding`, which itself, if provided a valid file, calls the function specified by **encoding-detection-hook**, or else defaults to the **default-encoding**. The **encoding-detection-hook** is by default bound to function `always-default-encoding`, that always returns the contents of **default-encoding**. **default-encoding** is bound to `:utf-8` by default (before ASDF 3, the default was `:default`).

Whichever encoding is returned must be a portable keyword, that will be translated to an implementation-specific external-format designator by function `encoding-external-format`, which itself simply calls the function specified **encoding-external-format-hook**; that function by default is `default-encoding-external-format`, that only recognizes `:utf-8` and `:default`, and translates the former to the implementation-dependent **utf-8-external-format**, and the latter to itself (that itself is portable but has an implementation-dependent meaning).

In other words, there now are plenty of extension hooks, but by default ASDF enforces the previous *de facto* standard behaviour of using `:utf-8`, independently from whatever configuration the user may be using. Thus, system authors can now rely on `:utf-8` being used while compiling their files, even if the user is currently using `:koi8-r` or `:euc-jp` as their interactive encoding. (Before ASDF 3, there was no such guarantee, `:default` was used, and only plain ASCII was safe to include in source code.)

Some legacy implementations only support 8-bit characters, and some implementations provide 8-bit only variants. On these implementations, the **utf-8-external-format** gracefully falls back to `:default`, and Unicode characters will be read as multi-character mojibake. To detect such situations, UIOP will push the `:asdf-unicode` feature on implementations that support Unicode, and you can use reader-conditionalization to protect any `:encoding` *`encoding`* statement, as in `#+asdf-unicode :encoding` `#+asdf-unicode :utf-8`. We recommend that you avoid using unprotected `:encoding` specifications until after ASDF 2.21 or later becomes widespread. As of May 2015, all maintained implementations provide ASDF 3, so you may prudently start using this and other features without such protection.

While it offers plenty of hooks for extension, and one such extension is available (see `asdf-encodings` 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 behaviour, 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. `:utf-8` has been the default value of `*default-encoding*` since ASDF 3.

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 still recommend you use `:utf-8` everywhere, which is the most portable (next to it is `:latin1`).

Recent versions of Quicklisp include `asdf-encodings`; if you're not using it, you may get this extension using git: *`git clone https://gitlab.common-lisp.net/asdf/asdf-encodings.git`* or *`git clone git@gitlab.common-lisp.net:asdf/asdf-encodings.git`*. You can also browse the repository on https://gitlab.common-lisp.net/asdf/asdf-encodings.

When you use `asdf-encodings`, any `.asd` file loaded will use the autodetection algorithm to determine its encoding. If you depend on this detection happening, you should explicitly load `asdf-encodings` early in your build. Note that `:defsystem-depends-on` cannot be used here: by the time the `:defsystem-depends-on` is loaded, the enclosing `defsystem` form has already been read.

In practice, this means that the `*default-encoding*` is usually used for `.asd` files. Currently, this defaults to `:utf-8`, and you should be safe using Unicode characters in those files. This might matter, for instance, in meta-data about author's names. Otherwise, the main data in these files is component (path)names, and we don't recommend using non-ASCII characters for these, for the result probably isn't very portable.

11.3 Miscellaneous Functions

These functions are exported by ASDF for your convenience.

`system-relative-pathname` *system name &key type* [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 6.3 [Pathname specifiers], page 14.

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. For these cases, `(asdf:clear-system :foo)` will 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 be loaded again.[1]

Note that this does not and cannot 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 data structures makes such a thing impossible in the general case. If the software being re-loaded is not conceived with hot upgrade in mind, 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 to extensive effort to make a hot upgrade possible with respect to its own code. If you want, you can reuse some of its utilities such as `uiop:define-package` and `uiop:with-upgradability`, and get inspiration (or disinspiration) from what it does in `header.lisp` and `upgrade.lisp`.

`register-preloaded-system` *name &rest keys* [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 `compile-bundle-op` or `monolithic-compile-bundle-op`, and want to register systems so that dependencies will work uniformly whether you're using your software from source or from fasl.

`register-immutable-system` *name &rest keys* [Function]

A system with name *name*, created by `make-instance` with extra keys *keys* (e.g. `:version`), is registered as *immutable*. That is, its code has already been loaded into the current image, and if at some point some other system `:depends-on` it, it is

[1] Alternatively, you could touch `foo.asd` or remove the corresponding fasls from the output file cache.

considered as already provided, and no attempt will be made to search for an updated version from the source-registry or any other method. There will be no search for an `.asd` file, and no `missing-component` error.

This function (available since ASDF 3.1.5) is particularly useful if you distribute a large body of code as a precompiled image, and want to allow users to extend the image with further extension systems, but without making thousands of filesystem requests looking for inexistent (or worse, out of date) source code for all the systems that came bundled with the image but aren't distributed as source code to regular users.

`run-shell-command` *control-string &rest args* [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 behaviour 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).

11.4 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 &key type 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 non-Unix systems it is only usable for relative pathnames. In order to manipulate relative pathnames portably, it is 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*`. `*nil-pathname*` is 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` so that `parse-unix-namestring` will throw an error if the pathname is absolute.

`merge-pathnames*` *specified &optional defaults* [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 &key type* [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 &key type* [Function]

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

`run-program` *command &key ignore-error-status force-shell input output error-output if-input-does-not-exist if-output-exists if-error-output-exists element-type external-format &allow-other-keys* [Function]

`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 behaviour.

`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 behaviour.)

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]

`slurp-input-stream` is 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 follows:

- 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 captured 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 captured 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`.

12 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 https://gitlab.common-lisp.net/asdf/asdf.git`*

You will find the above referenced tags in this repository. You can also browse the repository on `https://gitlab.common-lisp.net/asdf/asdf`.

Discussion of ASDF development is conducted on the mailing list (see Section 13.2 [Mailing list], page 65).

13 FAQ

13.1 "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 (see Section 13.2 [Mailing list], page 65).

13.2 Mailing list

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`

13.3 "What has changed between ASDF 1, ASDF 2, and ASDF 3?"

We released ASDF 2.000 on May 31st 2010, ASDF 3.0.0 on May 15th 2013, ASDF 3.1.2 on May 6th 2014. Releases of ASDF 2 and now ASDF 3 have since then been included in all actively maintained CL implementations that used to bundle ASDF 1, plus many implementations that previously did not. ASDF has been made to work with all actively maintained CL implementations and even a few implementations that are *not* actively maintained. See Chapter 13 ["What has changed between ASDF 1 ASDF 2 and ASDF 3?"], page 65.

Furthermore, it is possible to upgrade from ASDF 1 to ASDF 2 or ASDF 3 on the fly (though we recommend instead upgrading your implementation or replacing its ASDF module). 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 3 [Loading ASDF], page 3.

Note that in the context of compatibility requirements, ASDF 2.27, released on Feb 1st 2013, and further releases up to 2.33, count as pre-releases of ASDF 3, and define the `:asdf3` feature, though the first stable release of ASDF 3 was release 3.0.1. Significant new or improved functionality were added in ASDF 3.1; the `:asdf3.1` feature is present in recent enough versions to detect this functionality; the first stable release since then was ASDF 3.1.2. New `*features*` are only added at major milestones, and the next one will probably be `:asdf3.2`.

13.3.1 What are ASDF 1, ASDF 2, and ASDF 3?

ASDF 1 refers to any release earlier than 1.369 or so (from August 2001 to October 2009), and to any development revision earlier than 2.000 (May 2010). If your copy of ASDF doesn't even contain version information, it's an old ASDF 1. Revisions between 1.656 and 1.728 may count as development releases for ASDF 2.

ASDF 2 refers to releases from 2.000 (May 31st 2010) to 2.26 (Oct 30th 2012), and any development revision newer than ASDF 1 and older than 2.27 (Feb 1st 2013).

ASDF 3 refers to releases from 2.27 (Feb 1st 2013) to 2.33 and 3.0.0 onward (May 15th 2013). 2.27 to 2.33 count as pre-releases to ASDF 3.

ASDF 3.1 refers to releases from 3.1.2 (May 6th 2014) onward. These releases are also considered part of ASDF 3.

13.3.2 How do I detect the ASDF version?

All releases of ASDF push `:asdf` onto `*features*`. Releases starting with ASDF 2 push `:asdf2` onto `*features*`. Releases starting with ASDF 3 (including 2.27 and later pre-releases) push `:asdf3` onto `*features*`. Furthermore, releases starting with ASDF 3.1.2 (May 2014), though they count as ASDF 3, include enough progress that they also push `:asdf3.1` onto `*features*`. You may depend on the presence or absence of these features to write code that takes advantage of recent ASDF functionality but still works on older versions, or at least detects the old version and signals an error.

Additionally, all releases starting with ASDF 2 define a function `(asdf:asdf-version)` you may use to query the version. All releases starting with 2.013 display the version number prominently on the second line of the `asdf.lisp` source file.

If you are experiencing problems or limitations of any sort with ASDF 1 or ASDF 2, we recommend that you should upgrade to the latest release, be it ASDF 3 or other.

Finally, here is a code snippet to programmatically determine what version of ASDF is loaded, if any, that works on all versions including very old ones:

```
(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 it returns anything older than `"3.0.1"`, you really need to upgrade your implementation or at least upgrade its ASDF. See Section 3.4 [Replacing your implementation's ASDF], page 4.

13.3.3 ASDF can portably name files in subdirectories

Common Lisp namestrings are not portable, except maybe for logical pathname namestrings, 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 11 [Miscellaneous additional functionality], page 56, `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 6.3 [Pathname specifiers], page 14.

13.3.4 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 9 [Controlling where ASDF saves compiled files], page 48.

13.3.5 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 8 [Controlling where ASDF searches for systems], page 38.

13.3.6 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`.

13.3.7 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 cubic time recursive append, for an overall *O(n)* complexity vs *O(n^4)*.
- 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, though some tests are commented out on a few implementations.
- 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.

13.3.8 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.

13.3.9 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.

13.3.10 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.

13.3.11 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 13 ["How can I wholly disable the compiler output cache?"], page 65.
- 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 9 [Backward Compatibility], page 48. 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 13 [FAQ], page 65 below. `source-file-type` is deprecated. To access a component's file-type, use `file-type`, instead. `source-file-type` will be removed.

13.3.12 Pitfalls of the upgrade to ASDF 3

While ASDF 3 is largely compatible with ASDF 2, there are a few pitfalls when upgrading from ASDF 2, due to limitations in ASDF 2.

- ASDF 2 was designed so it could be upgraded; but upgrading it required a special setup at the beginning of your build files. Failure to upgrade it early could result in catastrophic attempt to self-upgrade in mid-build.
- Starting with ASDF 3 (2.27 or later), ASDF will automatically attempt to upgrade itself as the first step before any system operation, to avoid any possibility of such catastrophic mid-build self-upgrade. But that doesn't help if your old implementation still provides ASDF 2.
- It was unsafe in ASDF 2 for a system definition to declare a dependency on ASDF, since it could trigger such catastrophe for users who were not carefully configured. If you declare a dependency on a recent enough ASDF, yet want to be nice with these potentially misconfigured users, we recommend that you not only specify a recent ASDF in your dependencies with `:depends-on ((:version "asdf" "3.1.2"))`, but that you *also* check that ASDF 3 is installed, or else the upgrade catastrophe might happen before that specification is checked, by starting your `.asd` file with a version check as follows:

```
#-asdf3 (error "MY-SYSTEM requires ASDF 3.1.2")
```

- When you upgrade from too old a version of ASDF, previously loaded ASDF extensions become invalid, and will need to be reloaded. Example extensions include CFFI-Grovel, hacks used by ironclad, etc. Since it isn't possible to automatically detect what extensions need to be invalidated and what systems use them, ASDF will invalidate *all* previously loaded systems when it is loaded on top of a forward-incompatible ASDF version.[1]

[1] Forward incompatibility can be determined using the variable `asdf/upgrade::*oldest-forward-compatible-asdf-version*`, which is 2.33 at the time of this writing.

- To write a portable build script, you need to rely on a recent version of UIOP, but until you have ensured a recent ASDF is loaded, you can't rely on UIOP being present, and thus must manually avoid all the pathname pitfalls when loading ASDF itself.
- Bugs in CMUCL and XCL prevent upgrade of ASDF from an old forward-incompatible version. Happily, CMUCL comes with a recent ASDF, and XCL is more of a working demo than something you'd use seriously anyway.
- For the above reasons, your build and startup scripts should load, configure and upgrade ASDF among the very first things they do, and ensure that ASDF 3 or later is present indeed, before they start using ASDF to load anything else.
- Now that all implementations provide ASDF 3 or later (since May 2015), the simple solution is just to use code as below in your setup, and when it fails, upgrade your implementation or replace its ASDF. (see Section 3.4 [Replacing your implementation's ASDF], page 4):

```
(require "asdf")
#-asdf3 (error "ASDF 3 or bust")
```

- For scripts that try to use ASDF simply via `require` at first, and make heroic attempts to load it the hard way if at first they don't succeed, see `tools/load-asdf.lisp` distributed with the ASDF source repository, or the code of `cl-launch`.
- Note that in addition to the pitfalls and constraints above, these heroic scripts (should you wish to write or modify one), must take care to configure ASDF *twice*. One first time, right after you load the old ASDF 2 and before you upgrade to the new ASDF 3, so it may find where you put ASDF 3. One second time, because some implementations can't handle a smooth upgrade to ASDF 3, and lose configuration as they do.

```
(require "asdf")
;; <--- insert heroics here, if that failed to provide ASDF 2 or 3
;; <--- insert configuration here, if that succeeded
(asdf:load-system "asdf")
;; <--- re-configure here, too, in case at first you got ASDF 2
```

13.3.13 What happened to the bundle operations?

`asdf-ecl` and its short-lived successor `asdf-bundle` are no more, having been replaced by code now built into ASDF 3. Moreover, the name of the bundle operations has changed since ASDF 3.1.3. See Section 13.3.13 [What happened to the bundle operations], page 71.

And yet, the feature is not enabled to be used by `load-system` by default on ECL as originally intended, because of a bug in ECL itself found during testing.

Some of the bundle operations were renamed after ASDF 3.1.3, and the old names have been removed. Old bundle operations, and their modern equivalents are:

- `fasl-op` is now `compile-bundle-op`
- `load-fasl-op` is now `load-bundle-op`
- `binary-op` is now `deliver-asd-op`
- `monolithic-fasl-op` is now `monolithic-compile-bundle-op`
- `monolithic-load-fasl-op` is now `monolithic-load-bundle-op`
- `monolithic-binary-op` is now `monolithic-deliver-asd-op`

13.4 Issues with installing the proper version of ASDF

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

If you have a recent implementation, it should already come with ASDF 3 or later. If you need a more recent version than is provided, we recommend you simply upgrade ASDF by installing a recent version in a path configured in your source-registry. See Section 3.3 [Upgrading ASDF], page 4.

If you have an old implementation that does not provide ASDF 3, we recommend you replace your implementation's ASDF. See Section 3.4 [Replacing your implementation's ASDF], page 4.

13.4.2 "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. Though we do try to test ASDF releases against all implementations that we can, we may not be testing against all variants of your implementation, and we may not be running enough tests; we trust you to thoroughly test it with your own 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 since 1.1.13 or MKCL since 1.1.9, have ASDF create bundle FASLs that are provided as modules by your Lisp distribution. You may also, but we don't recommend that anymore, as in SBCL up until 1.1.12, have ASDF be implicitly used to `cl:require` these modules that are provided by your Lisp distribution; if you do, you should add these modules in the beginning of both `wrapping-source-registry` and `wrapping-output-translations`.
- If you have magic systems as above, like SBCL used to do, 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. If they are precompiled, they should also be in the `wrapping-output-translations`.

- Since ASDF 3, the library UIOP comes transcluded in ASDF. But if you want to be nice to users who care for UIOP but not for ASDF, you may package UIOP separately, so that one may `(require "uiop")` and not load ASDF, or one may `(require "asdf")` which would implicitly require and load the former.
- 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.

13.5 Issues with configuring ASDF

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

See Chapter 9 [Controlling where ASDF saves compiled files], page 48.

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.

13.5.2 "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-disable-cache.conf
```

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)
```

Note that this does *NOT* belong in a `.asd` file. Please do not tamper with ASDF configuration from a `.asd` file, and only do this from your personal configuration or build scripts.

13.6 Issues with using and extending ASDF to define systems

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

ASDF provides a predefined test operation, `test-op`. See Section 7.1.1 [Predefined operations of ASDF], page 24. 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 (see Section 13.2 [Mailing list], page 65), and on the launchpad bug-tracker (see Section 13.1 [Where do I report a bug?], page 65). We provide some guidelines in the discussion of `test-op`.

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

Various ASDF extensions provide some kind of `doc-op` operation. See also `https://bugs.launchpad.net/asdf/+bug/479470`.

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

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

13.6.4 "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 or later:

```
(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).

13.6.5 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
  ...)
```

13.6.6 How do I mark a source file to be loaded only and not compiled?

There is no provision in ASDF for ensuring that some components are always loaded as source, while others are always compiled. There is `load-source-op` (see Section 7.1.1 [Predefined operations of ASDF], page 24), but that is an operation to be applied to a system as a whole, not to one or another specific source files. While this idea often comes up in discussions, it doesn't play well with either the linking model of ECL or with various bundle operations. In addition, the dependency model of ASDF would have to be modified incompatibly to allow for such a trick.

13.6.7 How do I work with readtables?

It is possible to configure the lisp syntax by modifying the currently-active readtable. However, this same readtable is shared globally by all software being compiled by ASDF, especially since `load` and `compile-file` both bind **readtable**, so that its value is the same across the build at the start of every file (unless overridden by some `perform :around` method), even if a file locally binds it to a different readtable during the build.

Therefore, the following hygiene restrictions apply. If you don't abide by these restrictions, there will be situations where your output files will be corrupted during an incremental build. We are not trying to prescribe new restrictions for the sake of good style: these restrictions have always applied implicitly, and we are simply describing what they have always been.

- It is forbidden to modifying any standard character or standard macro dispatch defined in the CLHS.
- No two dependencies may assign different meanings to the same non-standard character.
- Using any non-standard character while expecting the implementation to treat some way counts as such an assignment of meaning.
- libraries need to document these assignments of meaning to non-standard characters.
- free software libraries will register these changes on: `http://www.cliki.net/Macro%20Characters`

If you want to use readtable modifications that cannot abide by those restrictions, you *must* create a different readtable object and set **readtable** to temporarily bind it to your new readtable (which will be undone after processing the file).

For that, we recommend you use system `named-readtables` to define or combine such readtables using `named-readtables:defreadtable` and use them using `named-readtables:in-readtable`. Equivalently, you can use system `cl-syntax`, that itself uses `named-readtables`, but may someday do more with, e.g. **print-pprint-dispatch**.

For even more advanced syntax modification beyond what a readtable can express, you may consider either:

- a `perform` method that compiles a constant file that contains a single form `#.*code-read-with-alternate-reader*` in an environment where this special variable was bound to the code read by your alternate reader, or
- using the system `reader-interception`.

Beware that it is unsafe to use ASDF from the REPL to compile or load systems while the readtable isn't the shared readtable previously used to build software. You *must* manually undo any binding of **readtable** at the REPL and restore its initial value whenever you call `operate` (via e.g. `load-system`, `test-system` or `require`) from a REPL that is using a different readtable.

13.6.7.1 How should my system use a readtable exported by another system?

Use from the `named-readtables` system the macro `named-readtables:in-readtable`.

If the other system fails to use `named-readtables`, fix it and send a patch upstream. In the day and age of Quicklisp and clbuild, there is little reason to eschew using such an important library anymore.

13.6.7.2 How should my library make a readtable available to other systems?

Use from the `named-readtables` system the macro `named-readtables:defreadtable`.

13.7 ASDF development FAQs

13.7.1 How do run the tests interactively in a REPL?

This not-so-frequently asked question is primarily for ASDF developers, but those who encounter an unexpected error in some test may be interested, too.

Here's the procedure for experimenting with tests in a REPL:

```
;; BEWARE! Some tests expect you to be in the .../asdf/test directory
;; If your REPL is not there yet, change your current directory:
;; under SLIME, you may: ,change-directory ~/common-lisp/asdf/test/
;; otherwise you may evaluate something like:
(require "asdf") (asdf:upgrade-asdf) ;load UIOP & update asdf.lisp
(uiop:chdir (asdf:system-relative-pathname :asdf "test/"))
(setf *default-pathname-defaults* (uiop:getcwd))

;; Load the test script support.
(load "script-support.lisp")

;; Initialize the script support.
```

```
;; This will also change your *package* to asdf-test.
;; NB: this function is also available from package cl-user,
;; and also available with the shorter name da in both packages.
(asdf-test::debug-asdf)

;; In case you modified ASDF since you last tested it,
;; you need to update asdf.lisp itself by evaluating 'make' in a shell,
;; or (require "asdf") (asdf:load-system :asdf) in another CL REPL,
;; if not done in this REPL above.
;; *Then*, in this REPL, you need to evaluate:
;(asdf-test::compile-load-asdf)

;; Now, you may experiment with test code from a .script file.
;; See the instructions given at the end of your failing test
;; to identify which form is needed, e.g.
(frob-packages)
(asdf::with-asdf-cache () (load "test-utilities.script"))
```

Ongoing Work

For an active list of things to be done, see the `TODO` file in the source repository.

Also, bugs are currently tracked on launchpad: `https://launchpad.net/asdf`.

Bibliography

- Francois-Rene Rideau: "ASDF 3, or Why Lisp is Now an Acceptable Scripting Language", 2014. This article describes the innovations in ASDF 3 and 3.1, as well as historical information on previous versions. https://github.com/fare/asdf3-2013
- Alastair Bridgewater: "Quick-build" (private communication), 2012. `quick-build` is a simple and robust one file, one package build system, similar to `faslpath`, in 182 lines of code (117 of which are not blank, not comments, not docstrings). Unhappily, it remains unpublished and its IP status is unclear as of April 2014. `asdf/package-system` is mostly compatible with it, modulo a different setup for toplevel hierarchies.
- Zach Beane: "Quicklisp", 2011. The Quicklisp blog and Xach's livejournal contain information on Quicklisp. http://blog.quicklisp.org/ http://xach.livejournal.com/
- Francois-Rene Rideau and Robert Goldman: "Evolving ASDF: More Cooperation, Less Coordination", 2010. This article describes the main issues solved by ASDF 2. https://common-lisp.net/project/asdf/doc/ilc2010draft.pdf https://gitlab.common-lisp.org/asdf/ilc2010
- Francois-Rene Rideau and Spencer Brody: "XCVB: an eXtensible Component Verifier and Builder for Common Lisp", 2009. This article describes XCVB, a proposed competitor for ASDF; many of its ideas have been incorporated into ASDF 2 and 3, though many other ideas still haven't. https://common-lisp.net/project/xcvb/
- Peter von Etter: "faslpath", 2009. `faslpath` is similar to the latter `quick-build` and our yet latter `asdf/package-system` extension, except that it uses dot `.` rather than slash `/` as a separator. https://code.google.com/p/faslpath/
- Drew McDermott: "A Framework for Maintaining the Coherence of a Running Lisp," International Lisp Conference, 2005, available in pre-print form at http://www.cs.yale.edu/homes/dvm/papers/lisp05.pdf
- Dan Barlow: "ASDF Manual", 2004. Older versions of this document from the days of ASDF 1; they include ideas laid down by Dan Barlow, and comparisons with older defsystems (`mk-defsystem`) and defsystem (`defsystem-4`, kmp's Memo 801).
- Marco Antoniotti and Peter Van Eynde: "`DEFSYSTEM`: A `make` for Common Lisp, A Thoughtful Re-Implementation of an Old Idea", 2002. The `defsystem-4` proposal available in the CLOCC repository.
- Mark Kantrovitz: "Defsystem: A Portable Make Facility for Common Lisp", 1990. The classic `mk-defsystem`, later variants of which are available in the CLOCC repository as `defsystem-3.x`.
- Richard Elliot Robbins: "BUILD: A Tool for Maintaining Consistency in Modular Systems", MIT AI TR 874, 1985. ftp://publications.ai.mit.edu/ai-publications/pdf/AITR-874.pdf
- Kent M. Pitman (kmp): "The Description of Large Systems", MIT AI Memo 801, 1984. Available in updated-for-CL form on the web at http://nhplace.com/kent/Papers/Large-Systems.html
- Dan Weinreb and David Moon: "Lisp Machine Manual", MIT, 1981. The famous CHINE NUAL describes one of the earliest variants of DEFSYSTEM. https://bitsavers.trailing-edge.com/pdf/mit/cadr/chinual_4thEd_Jul81.pdf

Concept Index

*

features 66

:

:also-exclude source config directive 40
:around-compile 56
:asdf 1
:asdf2 1
:asdf3 1
:compile-check 56
:default-registry source config directive 40
:defsystem-depends-on 16
:directory source config directive 40
:exclude source config directive 40
:if-component-dep-fails component option 20
:if-feature component option 20
:ignore-invalid-entries source config directive ... 40
:include source config directive 40
:inherit-configuration source config directive 40
:require dependencies 18
:tree source config directive 40
:version 12, 18, 32
:weakly-depends-on 17

A

also-exclude source config directive 40
around-compile keyword 56
ASDF versions 1
ASDF-BINARY-LOCATIONS compatibility ... 49
asdf-output-translations 48
ASDF-related features 1
asdf-user 11
ASDF-USER package 30

B

bug tracker 65

C

compile-check keyword 56
component 29
component designator 29

D

default-registry source config directive 40
DEFSYSTEM grammar 14
directory source config directive 40

E

exclude source config directive 40

I

ignore-invalid-entries source config directive 40
immutable systems 24, 60
include source config directive 40
inherit-configuration source config directive 40

L

launchpad 65
logical pathnames 19

M

mailing list 65

O

operation 23

P

pathname specifiers 17
Primary system name 30

R

readtables 76

S

serial dependencies 19
system 29
system designator 29
System names 30

T

Testing for ASDF 1
tree source config directive 40

V

version specifiers 18

Function and Class Index

A

already-loaded-systems 9
apply-output-translations 53
asdf-version 66

B

binary-op (obsolete) 71

C

clear-configuration 7
clear-output-translations 7, 53
clear-source-registry 45
clear-system 59
coerce-name 28, 31, 32
compile-bundle-op 71
compile-bundle-op, ... 26
compile-file* 56
compile-op 24
compile-system 9
component 23
component-depends-on 28
component-pathname 34
concatenate-source-op, ... 27

D

defsystem 11, 12, 14
deliver-asd-op 71
disable-output-translations 53

E

enable-asdf-binary-locations-compatibility 49
ensure-output-translations 53
ensure-source-registry 45

F

fasl-op (obsolete) 71
file-type 70
find-component 28, 31
find-system 29

I

initialize-output-translations 53
initialize-source-registry 45
input-files 28, 29

L

load-asd 11
load-bundle-op 71
load-fasl-op (obsolete) 71
load-op 25
load-source-op, ... 25
load-system 9
locate-system 31

M

make 9
merge-pathnames* 61
module 35
monolithic-binary-op (obsolete) 71
monolithic-compile-bundle-op 71
monolithic-deliver-asd-op 71
monolithic-fasl-op (obsolete) 71
monolithic-load-bundle-op 71
monolithic-load-fasl-op (obsolete) 71

O

oos 9
oos 24
operate 9
operate 24
operation 23
operation-done-p 29
OPERATION-ERROR 55
output-files 28

P

parse-unix-namestring 60
perform 28
prepare-op 25
primary-system-name 30

R

register-immutable-system 24, 59
register-preloaded-system 59
require-system 9
run-program 61
run-shell-command 60

S

slurp-input-stream 63
source-file 34
source-file-type 70
subpathname 61
subpathname* 61

system 35
SYSTEM-DEFINITION-ERROR 55
system-defsystem-depends-on 46
system-depends-on 46
system-relative-pathname 58
system-source-directory 59
system-weakly-depends-on 46

T

test-op 25
test-system 9
traverse 24

V

version-satisfies 32, 36

Variable Index

*

compile-file-failure-behaviour 55
compile-file-warnings-behaviour 55
default-source-registry-exclusions 44
features 1
image-dump-hook 8
load-system-operation 9
nil-pathname 61
oldest-forward-compatible-asdf-version 70
source-registry-parameter 46
system-definition-search-functions 29

A

asdf::*user-cache* 51
ASDF_OUTPUT_TRANSLATIONS 48