asdf: another system definition facility

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

asdf Copyright © 2001-2004 Daniel Barlow and contributors

This manual Copyright © 2001-2004 Daniel Barlow and contributors

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 Using asdf to load systems

This chapter describes how to use asdf to compile and load ready-made Lisp programs and libraries.

1.1 Downloading asdf

Some Lisp implementations (such as SBCL and OpenMCL) come with asdf included already, so you don't need to download it separately. Consult your Lisp system's documentation. If you need to download asdf and install it by hand, the canonical source is the cCLan CVS repository at http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/cclan/asdf/.

1.2 Setting up asdf

The single file '`asdf.lisp`' is all you need to use asdf normally. Once you load it in a running Lisp, you're ready to use asdf. 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.

The variable `asdf:*central-registry*` is a list of "system directory designators"[1]. A *system directory designator* is a form which will be evaluated whenever a system is to be found, and must evaluate to a directory to look in. You might want to set or augment `*central-registry*` in your Lisp init file, for example:

```
(setf asdf:*central-registry*
  (list* '*default-pathname-defaults*
         #p"/home/me/cl/systems/"
         #p"/usr/share/common-lisp/systems/"
         asdf:*central-registry*))
```

1.3 Setting up a system to be loaded

To compile and load a system, you need to ensure that a symbolic link to its system definition is in one of the directories in `*central-registry*`[2].

For example, if `#p"/home/me/cl/systems/"` (note the trailing slash) is a member of `*central-registry*`, you would set up a system *foo* that is stored in a directory '`/home/me/src/foo/`' for loading with asdf with the following commands at the shell (this has to be done only once):

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

[1] When we say "directory" here, we mean "designator for a pathname with a supplied DIRECTORY component".

[2] It is possible to customize the system definition file search. That's considered advanced use, and covered later: search forward for `*system-definition-search-functions*`. See Chapter 2 [Defining systems with defsystem], page 3.

1.4 Loading a system

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

```
(asdf:operate 'asdf:load-op 'foo)
```

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 Lisp software you write yourself.

2 Defining systems with defsystem

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

2.1 The defsystem form

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

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

```
(defpackage hello-lisp-system
  (:use :common-lisp :asdf))

(in-package :hello-lisp-system)

(defsystem "hello-lisp"
    :description "hello-lisp: a sample Lisp system."
    :version "0.2"
    :author "Joe User <joe@example.com>"
    :licence "Public Domain"
    :components ((:file "packages")
                 (:file "macros" :depends-on ("packages"))
                 (:file "hello" :depends-on ("macros"))))
```

Some notes about this example:

- The file starts with `defpackage` and `in-package` forms to make and use a package expressly for defining this system in. This package is named by taking the system name and suffixing `-system` - note that it is *not* the same package as you will use for the application code.

  This is not absolutely required by asdf, but helps avoid namespace pollution and so is considered good form.
- The defsystem form defines a system named `"hello-lisp"` that contains three source files: '`packages`', '`macros`' and '`hello`'.
- The file '`macros`' depends on '`packages`' (presumably because the package it's in is defined in '`packages`'), and the file '`hello`' depends on '`macros`' (and hence, transitively on '`packages`'). This means that asdf will compile and load '`packages`' and '`macros`' before starting the compilation of file '`hello`'.
- The files are located in the same directory as the file with the system definition. asdf resolves symbolic links before loading the system definition file and stores its location in the resulting system[1]. This is a good thing because the user can move the system sources without having to edit the system definition.

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

2.2 A more involved example

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

```
(defsystem "foo"
  :version "1.0"
  :components ((:module "foo" :components ((:file "bar") (:file"baz")
                                           (:file "quux"))
        :perform (compile-op :after (op c)
  (do-something c))
:explain (compile-op :after (op c)
  (explain-something c)))
              (:file "blah")))
```

The method-form tokens need explaining: essentially, this part:

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

has the effect of

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

where `...` is the component in question; note that although this also supports `:before` methods, they may not do what you want them to – a `:before` method on perform `((op compile-op) (c (eql ...)))` will run after all the dependencies and sub-components have been processed, but before the component in question has been compiled.

2.3 The defsystem grammar

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

option := :components component-list
        | :pathname pathname
        | :default-component-class
        | :perform method-form
        | :explain method-form
        | :output-files  method-form
        | :operation-done-p method-form
        | :depends-on ( {simple-component-name}* )
        | :serial [ t | nil ]
        | :in-order-to ( {dependency}+ )

component-list := ( {component-def}* )

component-def  := simple-component-name
                | ( component-type name {option}* )
```

```
component-type := :module | :file | :system | other-component-type

dependency := (dependent-op {requirement}+)
requirement := (required-op {required-component}+)
             | (feature feature-name)
dependent-op := operation-name
required-op := operation-name | feature
```

2.3.1 Serial dependencies

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

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

is equivalent to

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

2.3.2 Source location

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

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

1. `find-system` will load a system from disk and have its pathname default to the right place
2. this pathname information will not be overwritten with `*default-pathname-defaults*` (which could be somewhere else altogether) if the user loads up the '`.asd`' file into his editor and interactively re-evaluates that form.

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

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

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

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

3 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 protocol. asdf is extensible to new operations and to new component types. This allows the addition of behaviours: for example, a new component could be added for Java JAR archives, and methods specialised on `compile-op` added for it that would accomplish the relevant actions.

This chapter deals with *components*, the building blocks of a system, and *operations*, the actions that can be performed on a system.

3.1 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. *FIXME: document how!* There are a bunch of methods specialised on operation and component type that actually do the grunt work.

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

Operations are invoked on systems via `operate`.

`operate` *operation system* **&rest** *initargs* [Generic function]
`oos` *operation system* **&rest** *initargs* [Generic function]

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

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

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

3.1.1 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` **&key** *proclamations* [Operation]

This operation compiles the specified component. If proclamations are supplied, they will be proclaimed. This is a good place to specify optimization settings.

When creating a new component type, you should provide methods for `compile-op`.

When `compile-op` is invoked, component dependencies often cause some parts of the system to be loaded as well as compiled. Invoking `compile-op` does not necessarily load all the parts of the system, though; use `load-op` to load a system.

`load-op` **&key** *proclamations* [Operation]

This operation loads a system.

The default methods for `load-op` compile files before loading them. For parity, your own methods on new component types should probably do so too.

`load-source-op` [Operation]

This operation will load the source for the files in a module even if the source files have been compiled. Systems sometimes have knotty dependencies which require that sources are loaded before they can be compiled. This is how you do that.

If you are creating a component type, you need to implement this operation - at least, where meaningful.

`test-system-version` **&key** *minimum* [Operation]

Asks the system whether it satisfies a version requirement.

The default method accepts a string, which is expected to contain of a number of integers separated by #\. characters. The method is not recursive. The component satisfies the version dependency if it has the same major number as required and each of its sub-versions is greater than or equal to the sub-version number required.

```
(defun version-satisfies (x y)
  (labels ((bigger (x y)
     (cond ((not y) t)
      ((not x) nil)
      ((> (car x) (car y)) t)
      ((= (car x) (car y))
       (bigger (cdr x) (cdr y))))))
       (and (= (car x) (car y))
    (or (not (cdr y)) (bigger (cdr x) (cdr y))))))
```

If that doesn't work for your system, you can override it. I hope you have as much fun writing the new method as `#lisp` did reimplementing this one.

`feature-dependent-op` [Operation]

An instance of `feature-dependent-op` will ignore any components which have a `features` attribute, unless the feature combination it designates is satisfied by `*features*`. This operation is not intended to be instantiated directly, but other operations may inherit from it.

3.1.2 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.

An operation must provide methods for the following generic functions when invoked with an object of type `source-file`: *FIXME describe this better*

- `output-files`
- `perform` The `perform` method must call `output-files` to find out where to put its files, because the user is allowed to override
- `output-files` for local policy `explain`
- `operation-done-p`, if you don't like the default one

3.2 Components

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

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

`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. The resulting pathname is loaded if either of the following conditions is true:

- there is no system of that name in memory
- the file's last-modified time exceeds the last-modified time of the system in memory

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

The default value of `*system-definition-search-functions*` is a function that 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. Hence, it is strongly advised to define a system *foo* in the corresponding file *foo.asd*.

3.2.1 Common attributes of components

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

3.2.1.1 Name

A component name is a string or a symbol. If a symbol, its name is taken and lowercased. The name must be a suitable value for the `:name` initarg to `make-pathname` in whatever filesystem the system is to be found.

The lower-casing-symbols behaviour is unconventional, but was selected after some consideration. Observations suggest that the type of systems we want to support either have lowercase as customary case (Unix, Mac, windows) or silently convert lowercase to uppercase (lpns), so this makes more sense than attempting to use `:case :common` as argument to `make-pathname`, which is reported not to work on some implementations

3.2.1.2 Version identifier

This optional attribute is used by the test-system-version operation. See Section 3.1.1 [Predefined operations of asdf], page 6. For the default method of test-system-version, the version should be a string of intergers separated by dots, for example '`1.0.11`'.

3.2.1.3 Required features

Traditionally defsystem users have used reader conditionals to include or exclude specific per-implementation files. This means that any single implementation cannot read the entire system, which becomes a problem if it doesn't wish to compile it, but instead for example to create an archive file containing all the sources, as it will omit to process the system-dependent sources for other systems.

Each component in an asdf system may therefore specify features using the same syntax as `#+` does, and it will (somehow) be ignored for certain operations unless the feature conditional is a member of `*features*`.

3.2.1.4 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)}+)

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

component-name := string
                | (:version string 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 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 a `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)

3.2.1.5 pathname

This attribute is optional and if absent will be inferred from the component's name, type (the subclass of source-file), and the location of its parent.

The rules for this inference are:

(for source-files)

- the host is taken from the parent
- pathname type is `(source-file-type component system)`
- the pathname case option is `:local`
- the pathname is merged against the parent

(for modules)

- the host is taken from the parent
- the name and type are `NIL`
- the directory is `(:relative component-name)`
- the pathname case option is `:local`
- the pathname is merged against the parent

Note that the DEFSYSTEM operator (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 2 [Defining systems with defsystem], page 3.

The answer to the frequently asked question "how do I create a system definition where all the source files have a .cl extension" is thus

```
(defmethod source-file-type ((c cl-source-file) (s (eql (find-system 'my-sys))))
   "cl")
```

3.2.1.6 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.

3.2.2 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 child components which don't specify their class explicitly are inferred to be of this type.
- `:if-component-dep-fails` This attribute takes one of the values `:fail`, `:try-next`, `:ignore`, its default value is `:fail`. The other values can be used for implementing conditional compilation based on implementation `*features*`, for the case where it is not necessary for all files in a module to be compiled.
- `: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 2.3 [The defsystem grammar], page 4.

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

3.2.3 Creating new component types

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

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

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

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

A hypothetical function `system-dependent-dirname` 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))
  (let ((pathname (call-next-method))
        (name (string-downcase (system-dependent-dirname))))
    (merge-pathnames
     (make-pathname :directory (list :relative name))
     pathname)))
```

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

4 Error handling

It is an error to define a system incorrectly: an implementation may detect this and 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`.

5 Compilation error and warning handling

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

6 Getting the latest version

1. Decide which version you want. HEAD is the newest version and usually OK, whereas RELEASE is for cautious people (e.g. who already have systems using asdf that they don't want broken), a slightly older version about which none of the HEAD users have complained.
2. Check it out from sourceforge cCLan CVS:

   ```
   cvs -d:pserver:anonymous@cvs.cclan.sourceforge.net:/cvsroot/cclan login
   ```

   (no password: just press ENTER)

   ```
   cvs -z3 -d:pserver:anonymous@cvs.cclan.sourceforge.net:/cvsroot/cclan co
   -r RELEASE asdf
   ```

   or for the bleeding edge, instead

   ```
   cvs -z3 -d:pserver:anonymous@cvs.cclan.sourceforge.net:/cvsroot/cclan co
   -A asdf
   ```

If you are tracking the bleeding edge, you may want to subscribe to the cclan-commits mailing list (see http://sourceforge.net/mail/?group_id=28536) to receive commit messages and diffs whenever changes are made.

For more CVS information, look at http://sourceforge.net/cvs/?group_id=28536.

7 TODO list

* Outstanding spec questions, things to add

** packaging systems

*** manual page component?

** style guide for .asd files

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

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

** extending defsystem with new options

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

** document all the error classes

** what to do with compile-file failure

Should check the primary return value from compile-file and see if that gets us any closer to a sensible error handling strategy

** foreign files

lift unix-dso stuff from db-sockets

** Diagnostics

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

```
(traverse (make-instance '<operation-name>)
          (find-system <system-name>)
          'explain)
```

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

8 missing bits in implementation

** all of the above

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

** rules for system pathname defaulting are not yet implemented properly

** proclamations probably aren't

** when a system is reloaded with fewer components than it previously had, odd things happen

we should do something inventive when processing a defsystem form, like take the list of kids and setf the slot to nil, then transfer children from old to new list as they're found

** traverse may become a normal function

If you're defining methods on traverse, speak up.

** a lot of load-op methods can be rewritten to use input-files

so should be.

** (stuff that might happen later)

*** david lichteblau's patch for symlink resolution?

*** Propagation of the :force option. "I notice that

(oos 'compile-op :araneida :force t)

also forces compilation of every other system the :araneida system depends on. This is rarely useful to me; usually, when I want to force recompilation of something more than a single source file, I want to recompile only one system. So it would be more useful to have make-sub-operation refuse to propagate `:force t` to other systems, and propagate only something like `:force :recursively`.

Ideally what we actually want is some kind of criterion that says to which systems (and which operations) a `:force` switch will propagate.

The problem is perhaps that 'force' is a pretty meaningless concept. How obvious is it that `load :force t` should force *compilation*? But we don't really have the right dependency setup for the user to compile `:force t` and expect it to work (files will not be loaded after compilation, so the compile environment for subsequent files will be emptier than it needs to be)

What does the user actually want to do when he forces? Usually, for me, update for use with a new version of the lisp compiler. Perhaps for recovery when he suspects that something has gone wrong. Or else when he's changed compilation options or configuration in some way that's not reflected in the dependency graph.

Other possible interface: have a 'revert' function akin to 'make clean'

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

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

How would this work?

traverse

There's a difference between a module's dependencies (peers) and its components (children). Perhaps there's a similar difference in operations? For example, `(load "use")`

`depends-on (load "macros")` is a peer, whereas `(load "use") depends-on (compile "use")` is more of a 'subservient' relationship.

9 Inspiration

9.1 mk-defsystem (defsystem-3.x)

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

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

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

9.2 defsystem-4 proposal

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

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

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

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

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

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

9.3 kmp's "The Description of Large Systems", MIT AI Memu 801

Available in updated-for-CL form on the web at http://world.std.com/~pitman/Papers/Large-Systems.ht

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

Concept Index

Function and Class Index

Variable Index