CHAPTER 19

Systems

Introduction

This chapter is about the compilation and loading of "systems"—the collections of files out of which libraries and applications are built. We encountered systems briefly in the ch-image example of Chapter 17 and now's the time to stop and explain them properly. This is important material, essential for working with the vast majority of libraries, whether somebody else's or your own. The message of the chapter is that there is plenty of support for downloading and building libraries in Common Lisp. As you might expect, this support is powered by Lisp forms running within your Lisp image—no need to stop the world when you want to recompile!—and is by and large platform independent.

Maybe even more than Concurrency (Chapter 15), or Lisp's interfaces for talking to other languages (Chapter 29), systems really need a uniform interface across the different implementations. In terms of popularity alone the library which has come closest to meeting this is ASDF (or, *Another System Definition Facility*). An extension to this is *ASDF-Install* which handles the downloading and installation of ASDF systems. We'll introduce these two libraries here, first from the perspective of using somebody else's systems and then from that of defining your own. We'll also take a very brief look at a few alternative approaches to system definition.

Using ASDF

Which ASDF?

ASDF is open source and covered by an MIT license. It has been so widely adopted by library writers that several implementations now bundle it with their distributions. What they won't all do is bundle the same version. For example, at the time of writing this chapter: the Armed Bear CL which I had recently installed had shipped ASDF

version 1.3, Allegro shipped 1.102, SBCL 1.130, and the copy accompanying my Clozure didn't carry any version stamp at all.

Exercise

For each Common Lisp implementation to which you have access, use `(find :asdf *features*)` to determine whether ASDF is pre-loaded on startup. If it isn't, try `(require "asdf")` to see whether or not the library is bundled with your Lisp. If this succeeds, see which version you've got by evaluating `asdf:*asdf-revision*`.

For example, on Clozure:

```
Welcome to Clozure Common Lisp Version 1.3-r12394M  (FreebsdX8632)!
? (find :asdf *features*)
NIL
? (require "asdf")
"asdf"
("ASDF" "asdf")
? (find :asdf *features*)
:ASDF
? asdf:*asdf-revision*
NIL                          ; Hmm, not as informative as I'd like it.
?
```

As downloaded from the project website, ASDF currently supports nine out of the eleven Common Lisp implementations listed in Chapter 1, the exceptions being:

- Armed Bear Common Lisp (ABCL): users should only load the copy bundled with ABCL itself, as recent versions of ASDF have used CLOS fucntionality (the long form of `define-method-combination`) which ABCL doesn't support.
- Embedded Common Lisp (ECL): similarly, users should only load the copy bundled with ECL. This supports a number of ECL-specific extensions: standalone executables, *unified fasls* (merging several different fasl files into one), shared and statically linked libraries, etc. See *http://ecls.sourceforge.net/new-manual/ch16.html* for further information.

My advice is that for all other Lisps you should pick a recent stable release to work with; for uniformity of treatment here, I've based my text on the release which was current when I wrote these words: 1.363. When we hit features which have only become supported in recent releases, I'll draw your attention to them.

Tip: ASDF is going through a phase of active development right now; by the time this book goes to press it may well provide several features that weren't there at time of writing. Don't worry about that: it's your choice whether to ignore them and work from the version documented here or be adventurous and take advantage of recent improvements.

Download and Install

As already noted (and demonstrated in Chapter 17), if ASDF is bundled with your Lisp you can load it by evaluating

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

Otherwise, or if you don't care for the version provided for you, visit the ASDF project homepage:

http://common-lisp.net/project/asdf/

where in addition to download links you'll also find a "getting started" guide even briefer than this one, a rather complicated programmer's manual, and the asdf-devel mailing list which should be your first port of call if you ever need technical help. ASDF itself is distributed as a single Lisp file and you can use anything from copy-and-paste in a web browser to the `curl` utility to make a local copy from

http://common-lisp.net/project/asdf/asdf.lisp

Compile the file and load it:

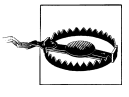
```
? (load (compile-file "/home/ndl/chapter-19/asdf"))
#P"/home/ndl/chapter-19/asdf.fx32fsl"
?
```

and you're good to go.

Loading System Definitions

The macro at the heart of most system definition libraries, ASDF included, is called `defsystem` (and so the libraries themselves tend to be referred to as "defsystems"). Each system is defined by a `defsystem` form which names the system and describes both its contents and how these depend upon each other at compile and load time. We'll postpone discussion of how ASDF defsystems are constructed for later and confine ourselves for now to putting them to use.

The first step is to load the defsystems so that your Lisp knows about them. The traditional approach for defsystem libraries—and one that would certainly work with ASDF—is simply to `cl:load` the file(s) containing your `defsystem` forms.



Don't use SLIME's compile or evaluate commands to load individual defsystem forms from Emacs buffers; ASDF will end up with an incorrect notion of where your source files live. Compiling the whole file (default binding `C-c C-k`) is fine.

ASDF offers an alternative strategy to explicit loading: tell it where to find your defsystems and ASDF will load them lazily (i.e. not until it has to). In either case, if a `defsystem` has been loaded and you subsequently edit its source file, then next time you perform any action on the system ASDF will automatically reload the definition.

ASDF finds `defsystem`s via the global variable `asdf:*central-registry*`. We met this before in Chapter 17. It's a list each of whose members is one of:

- a pathname which represents a *directory location* (the pathname has no name or type),
- the string representation of such a pathname,
- a Lisp form which will evaluate to one of these.

Exercise

ASDF really does call `eval` on members of this list. Is that safe? What's the default value of `asdf:*central-registry*`? How does that work?

In this context a string representing a directory location *must* end with a trailing directory separator (`#\/` on Unix or Mac OS, `#\\` on Windows). So in our ch-image example,

```
(push "~/chapter-17/asdf/" asdf:*central-registry*)        ;; RIGHT
```

was fine, whereas

```
(push "~/chapter-17/asdf" asdf:*central-registry*)         ;; WRONG
```

would have led to errors later.

ASDF locates a system by searching via the `*central-registry*` for a file with the same name as the system and with extension *.asd*. This gives you the freedom to choose between (or combine):

- pushing the location of each of your defsystem files onto the registry, and
- pushing one "central" location onto the registry and linking to your defsystem files from this location.

It's the name of the link which matters, not the name of the file at the other end.

We took the second approach in Chapter 17:

```
[ndl@vanity ~/chapter-17/asdf]$ ln -s ../*/*.asd .
```

accompanied by pushing *~/chapter-17/asdf/* onto the registry. This method has the advantage of being somewhat easier to maintain: all you have to do to add a new system to your application is link its *.asd* file into the central location. There are two potential disadvantages, the first being that symbolic links don't play nicely with all version control managers; the other can be sufficiently unpleasant that it warrants a section all of its own.

Getting Links to Work on Windows

Support by ASDF for Windows shortcuts is a very recent usability gain which at present applies only to `*central-registry*` searches (and in particular not to files named within `defsystem` forms—but ASDF is under very active development and this restriction might go away with future releases).

You may use *Cygwin*[*] to create the links but note that if these are not absolute then they'll be passed around unresolved and eventually merged against `*default-pathname-defaults*`. Unless you've set this variable appropriately, the result won't locate your defsystem file. For the time being, you're advised to use absolute symlinks in Cygwin:

```
ln -s c:/home/nick/chapter-*/*/*.asd .
```

rather than (say):

```
ln -s ../../chapter-*/*/*.asd .
```

This problem doesn't apply on Unix or Mac OS, both of which resolve symbolic links unobtrusively.

If you're stuck with an old version of ASDF for Windows (on Armed Bear CL, for example) and still want to use a single location for the `*central-registry*`, then you're simply going to have to modify your ASDF source. My best suggestion is that you copy in full the "Windows shortcut support" section from a current version, look for `#+win32` in the function `sysdef-central-registry-search` from that version, and then make corresponding changes in your copy of that function.

If you need your application to work correctly with Windows shortcuts, you could do a lot worse than lifting the code for this from ASDF. The MIT license permits you to do this, subject to conditions which are not particularly onerous.

Working with Systems

We can now go ahead and start compiling and loading our systems. If the defsystems have been set up correctly, this is straightforward. In current versions of ASDF, either:

```
(asdf:load-system "ch-image")
```

or equivalently:

```
(asdf:load-system 'ch-image)
```

In previous versions (prior to 1.352) the incantation was a little more verbose:

[*] Unix-like tools and environment from *http://www.cygwin.com/*

```
(asdf:operate 'asdf:load-op "ch-image")
```

The longer form is still supported and is equivalent to calling `asdf:load-system`.

The lookup of ASDF systems from their names is case sensitive, but symbols are downcased beforehand. So in effect it's case sensitive for strings, case insensitive for symbols. I very much recommend that systems should be defined with lower-case names.

A call to `asdf:load-system` will compile files before loading them. It will also compile and load any systems on which the system you're acting upon "depends". (Recall how loading the `ch-image` system in Chapter 17 automatically compiled and loaded five other subsystems.)

In theory this operation supports "incremental builds": it will compile only those files which need this (either because they haven't been compiled before, or because their source has been touched since it was last compiled, or because some dependency requires their recompilation). In the case of very large systems where the dependencies between files and subsystems are particularly complex this can become difficult to manage. The problem is by no means confined to ASDF and remains an area of active research.

Very occasionally, you will have good reason to force recompilation of a whole system. Do so by specifying a non-`nil` value for `:force`, thus:

```
(asdf:load-system "ch-image" :force t)
```

Defining Systems

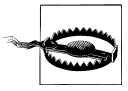
A system definition specifies not just a set of source files and subsystems but also the order in which they need to be compiled and loaded, and the conditions under which recompilation is necessary. It's important to get this right, because:

- Broken dependencies—things not defined before they're needed at compile or load time—are a frequent cause of failure in automated builds. Worse, they're an occasional cause of failure in running applications.
- Large systems can take a long time to rebuild from scratch and so brute-force approaches (recompile everything if anything changed, or even: compile everything "listed after" the one source file that's changed) can be expensive.

Dependencies

Let's stop and consider why load/compilation order matters. In particular:

- A package must be defined before you can refer to it. If a package definition refers to symbols in other packages (via the `:import-from` and `:shadowing-import-from` options) then these symbols must already exist.



Caution: Take care if you modify a package so that the home packages of any of its symbols change. The ANSI standard doesn't define what will happen if you load the new package definition on top of an old one, so you might need to load the system into a fresh Lisp to see any change. But even then, if your defsystem doesn't now force recompilation of every file which refers to these symbols then such references will—in most implementations—be to the wrong symbols.

Exercise

Demonstrate that this is possible.

- Classes must exist before methods can refer to them. However the classes can be redefined later without any need to recompile the methods.
- Macros (along with any supporting code which the macro functions invoke) must be defined before code which refers to them is compiled; and if they're redefined then that code must be recompiled. If inline functions aren't already defined then the calls to them won't be inlined.
- Reader macros and any supporting code, features (controlling `#+` and `#-`), and anything to be evaluated at read-time (consider `#.`) must be defined before the forms which invoke them can be read in, let alone compiled; and if any of these are redefined then those forms must be read in again.
- Any code required to evaluate the initial values for global variables must be defined before these variables' defining forms (`defvar`, `defparameter` or `defconstant`) are loaded. Similarly for the form to be evaluated by `load-time-value`.
- Constants are best defined before you refer to them.

Also, if—as undoubtedly you do—you want your application to compile without any warnings, then:

- Global variables should be defined as such before your code refers to them.

I've used the word "before" liberally here. For most of the above, this means either:

- defined earlier in the current file (such definitions which are available to the compiler without having been `cl:load`ed are said to be part of the *compilation environment*); or
- defined in a file which has already been loaded.

Note that function and method definitions (i.e. `defun` and `defmethod`) do not automatically become part of the compilation environment. So although you can compile a file containing:

```
(defmacro bar (x) `(cons ,x ,x))

(defun foo () (bar 99))
```

you'll get a compile-time error if the file reads:

```
(defun quux (x) `(cons ,x ,x))

(defmacro bar (x) (quux x))

(defun foo () (bar 99))
```

There are two solutions to this quandry. The first is fine when you've got a small amount of utility code in the same file as the form which invokes it. Wrap the definitions which need to be available at compile-time inside an `eval-when` form, thus:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defun quux (x) `(cons ,x ,x)))

(defmacro bar (x) (quux x))
```

Tip: `eval-when` is complex and nasty; let's not go into details. It's unlikely ever to be useful to you other than with the "situation" set as, in the example above, to `(:compile-toplevel :load-toplevel :execute)`. Think of this as meaning that all the forms within the body of the `eval-when` are to be executed at compile-time as well as when the file is loaded.

Your other option is to separate out any code which needs to run at compile-time into different files which ASDF can load earlier on. It's occasionally undesirable to split functionality like this, and that's when you want to consider `eval-when`.

The Defsystem Macro

This beast comes with plenty of bells and whistles. You'll find them all listed in the ASDF Manual

http://common-lisp.net/project/asdf/manual.html

but you don't need all this detail at once and the best way to get started is to look at other people's *.asd* files and adapt these as necessary. So let's do just that; here as they say is one I cooked earlier. (We'll be studying the RabbitMQ system in Chapter 30.)

```
(in-package :asdf)

(defsystem :rabbitmq
    :description "RABBITMQ - interface to RabbitMQ"
    :author "Nick Levine <ndl@ravenbrook.com>"
```

```
:components ((:file "java-classes")
             (:file "rabbitmq")
             (:file "pkg")
             (:file "errors")
             (:file "parameters")
             (:file "connection")
             (:file "channel")
             (:file "message")
             )
:depends-on (:jfli)
:serial t)
```

- The syntax is `(defsystem name &rest options)`.
- Component types `:system`, `:module` and `:file` are supported; you are free to define others.

Typically you'd do that by subclassing one of `asdf:module`, `asdf:system` (itself a subclass of `module`) or `asdf:cl-source-file`. There's an example of a subclass of `cl-source-file` in the ASDF manual.

- The members of systems and modules are specified by the `:components` option; each member is given either by a name or—as here—by a list of the form `(component-type name &rest options)`. Members can be files, modules or even other systems.
- Think of a module as an inlined system. For example:

```
:components ((:file "java-classes")
             (:file "rabbitmq")
             (:file "pkg")
             (:file "errors")
             (:file "parameters")
             (:module "implementation"
              :components ((:file "connection")
                           (:file "channel")
                           (:file "message")))
             )
```

- Some options are applicable to all component types; others (for example, `:description` and `:author` which apply only to systems) are specific to particular types.

The basic behavior of `load-system` is simply to load each member, compiling it first if necessary, all in the order stated.

Exercise

Implement the function `compile-file-if-necessary`. This should behave exactly like `compile-file`, which it will invoke only if the source file has not been compiled yet or has been changed since it was last compiled. Hint: look up what the Common Lisp functions `probe-file`, `file-write-date` and `compile-file-pathname` do. Make sure you handle `:output-file` arguments correctly. Justify your choice of return value(s) in the case where the source file doesn't need compiling.

Locations

By default, ASDF reads source files and writes compiled binaries back to the same location.

- For members of a system, this is the directory in which the *.asd* file was found.
- For members of a module, it's the subdirectory of the *.asd* file's location named after the module. So the components of the `"implementation"` module in this system:

```
:components (...
             (:module "implementation"
              :components (...))
             )
```

  would live in the `"implementation/"` subdirectory of wherever the *.asd* file was.

These defaults can be changed.

- For source files, specify the `:pathname` option. Typically, you'll want to resolve relative pathnames yourself (otherwise they'll be merged against `*default-pathname-defaults*`):

```
:pathname (merge-pathnames "somewhere/" *load-truename*)
```

- Relocate object files by defining `:around` methods on `asdf:output-files`. For example:

```
(defmethod asdf:output-files :around (operation component)
  (let ((files (call-next-method)))
    (mapcar (lambda (file)
              (merge-pathnames "fasls/" file))
            files)))
```

  would write all fasls into the *fasls/* subdirectory of their default location (and would read them back from the same place).

If you need more subtle control, then subclass `asdf:cl-source-file`, specialize `asdf:output-files` accordingly, and provide either the `:class` option for each of the files concerned or the `:default-component-class` option for their parent module.

Since this chapter was written, version 1.365 of ASDF has been released with support for the *ASDF-binary-locations* library which should mitigate these problems.

Dependencies Again

Dependencies determine which systems need to be loaded before this one, and any compilations which need to be performed in addition to the default "compile this file if it's out of date".

There are three ways of specifying dependencies. Two of them appear in the RabbitMQ example above.

- `:depends-on` specifies a list of members which must be compiled and loaded before this one can be either compiled or loaded.
- Setting `:serial` is equivalent to making each member depend on all previous members. In our example it's a shorthand for:

```
:components ((:file "java-classes")
             (:file "rabbitmq"
                    :depends-on ("java-classes"))
             (:file "pkg"
                    :depends-on ("java-classes" "rabbitmq"))
             ...
             (:file "message"
                    :depends-on ("java-classes" ... "channel"))
             )
```

So, in the RabbitMQ system I required two things: that the `jfli` system be compiled up to date and loaded, and that if any file in the system needed compiling then all files following it should be recompiled too. Admittedly this was rather lazy of me, but it was a small system and I could get away with it.

Exercise

The Java classes are used by the last three files; the package definition refers to the first two files and is needed by everything that follows it; there's a macro in *errors.lisp* which is used by the channel interface; the parameters only apply to setting up a connection; connection and channel define macros used both in their own files and in *channel.lisp* and *message.lisp* respectively. Rewrite the defsystem to express these dependencies more accurately. Alternatively either rearrange my code to make the system definition simpler, or do nothing at all and leave everything as it is.

Finally, `:in-order-to` allows you to describe more complex interactions in which the compile-time and load-time dependencies differ. Read the manual or borrow from other people's code if you think you need this.

Namespace Pollution

Let's revisit my definition of the `rabbitmq` system:

```
(in-package :asdf)

(defsystem :rabbitmq ...)
```

I've resisted the temptation to tidy this up just for appearance in a book; these forms are exactly as I wrote them two years ago.

You'll see that, being somewhat on the lazy side, I bashed this out in the `ASDF` package to give me easy access to the symbol defsystem. Of course, I could have chosen some other package (e.g. `COMMON-LISP-USER`) to work in and qualified the symbol explicitly (`asdf:defsystem`); I chose not to do this because I have an aesthetic preference for unqualified symbols at the beginning of defining forms.

ASDF documentation recommends that you do neither of these, as your `defsystem` form might "pollute" the namespace of the current package: any unqualified symbols in the form will be interned in `*package*` and your component names could conceivably collide with components in some pre-existing system. It suggests that you should instead define a fresh package for each `defsystem`; doing so in the same file would be fine. Thus:

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

(in-package :rabbitmq-system)

(defsystem :rabbitmq ...)
```

Opinions differ as to how to refer to components within `defsystem` forms: you can use strings, unqualified symbols (but as just noted these would be interned in the current package), keywords, uninterned symbols `#:like-this` or even† qualified symbols in random packages of your choice.

My own feeling is that, provided you don't intern symbols in system packages (by which I mean: packages belonging to the Lisp implementation or the external libraries you've loaded), none of the above matters very much. If your own application really cares about stray symbols in particular packages then there's probably something wrong with it.

† Just because you can doesn't mean you ought.

There's something of a tendency among Common Lisp implementors to prefer that the `COMMON-LISP` package starts up with no internal symbols and that `COMMON-LISP-USER` starts up empty, but they're not under any obligation. Don't count on it.

You will note that I've been using keywords to refer to packages—and symbols—in `defpackage` and `in-package` forms. Again, you're free to use symbols (and these could be unqualified, keywords, or uninterned) or strings and this is a something of a matter for personal preference. I recommend not using unqualified symbols in a `defpackage` form: this is the one place where sloppiness could get you bitten by unexpected side-effects.

Exercise

Why?

Also, I have a dislike for lists of uninterned symbols: they leave a lot of "ink on the page" and pull the eye off to one side without contributing anything useful.

```
(defpackage #:rabbitmq-system :use (#:common-lisp #:asdf))
```

Be consistent.

Exercise

Revisit the system definition of `ch-image` and figure out exactly how it works. What do you imagine a "static" file is? Seeing as this feature is undocumented, you'll have to flit through ASDF sources to confirm your answer.

ASDF-Install

Get Started

The job of ASDF-Install is to download and install Lisp libraries from the internet, resolving their dependencies on other libraries. Over 500 libraries are supported and you'll find these listed here:

http://www.cliki.net/ASDF-Install#asdf-installable-packages

We'll use a one-line invocation of ASDF-Install in Chapter 21 to download and build the Hunchentoot webserver and the numerous systems upon which it depends.

ASDF-Install is open sourced and covered by a two-clause BSD-like license. It supports seven of the eleven Common Lisp implementations listed in Chapter 1 (not Armed Bear, Corman, Embedded, or GNU CL). Some Lisps (CCL and SBCL, for example) redistribute it; check your documentation for specifics. Failing any joy there, visit the home page for ASDF-Install:

http://common-lisp.net/project/asdf-install/

where you'll find download links, a tutorial and—in case of problems—the asdf-install-devel mailing list.

ASDF-Install uses two command-line utilities: *GNU tar* for unpacking data and *GnuPG* for verifying key signatures.

- You should find these pre-installed on most Linux distributions.
- On Windows both facilities are provided by Cygwin.
- On FreeBSD you may need to install them from the ports collection:

  ```
  # cd /usr/ports/security/gnupg
  # make install clean
  ```

  and similarly for GNU tar from */usr/ports/archivers/gtar*.
- GnuPG is not pre-installed on Mac OS. Visit *http://macgpg.sourceforge.net/docs/howto-build-gpg-osx.txt.asc* for download and build instructions.

ASDF-Install itself is distributed as an ASDF system: follow the instructions earlier in this chapter to compile and load it. (Note that the ASDF distributed with Clozure 1.3 doesn't support `load-system`, so in this example I used the more verbose `(operate 'load-op ...)` form. Note also that I loaded the ASDF-Install which came with Clozure; it turns out that if I'd gone to the project website I would have ended up with the same version.)

```
[ndl@vanity ~]$ ccl
Welcome to Clozure Common Lisp Version 1.3-r12394M  (FreebsdX8632)!
? (require "asdf")
"asdf"
("ASDF" "asdf")
? (load "~/lisps/ccl/tools/asdf-install/asdf-install.asd")
#P"/home/ndl/lisps/ccl/tools/asdf-install/asdf-install.asd"
? (asdf:operate 'asdf:load-op 'asdf-install)
;;; ASDF-Install version 0.6.10
NIL
?
```

Installing a Library

There are three ways to specify a library's location to ASDF-Install:

- From the CLiki: "foo" is downloadable if *http://www.cliki.net/foo?download* works (see also *http://www.cliki.net/ASDF-Install*). Example: `(asdf-install:install :zpng)`.
- Using the download URL of an ASDF-installable system: `(asdf-install "http://www.xach.com/lisp/zpng.tgz")`
- From a local copy of an ASDF-installable system: `(asdf-install "/home/ndl/chapter-17/tarballs/zpng.tgz")`

To be *ASDF-installable* the library has to conform to certain specifications and we'll come to those in "Publishing a Library" on page 17 below.

In all three cases, ASDF-Install will ask you where you'd like to store the library once it's been unpacked:

```
? (asdf-install:install :zpng)
Install where?
1) System-wide install:
   System in /usr/local/asdf-install/site-systems/
   Files in /usr/local/asdf-install/site/
2) Personal installation:
   System in /home/ndl/.asdf-install-dir/systems/
   Files in /home/ndl/.asdf-install-dir/site/
0) Abort installation.
 -->
```

Configure these locations, or add more, by looking at the parameter `asdf-install:*locations*`. Make the installation non-interactive by setting `asdf-install:*preferred-location*`.

After that, in theory, it's all plain sailing. Files are downloaded as necessary, PGP signatures are checked, *.asd* files are loaded and dependencies resolved (possibly by downloading further systems), and the systems are compiled and loaded.

Tip: If you get the warning `"Cannot find tar command NIL."` followed by an error, check that the path to GNU tar is represented on `asdf-install:*shell-search-paths*`, and restart the installation.

```
[ndl@vanity ~]$ which gtar
/usr/local/bin/gtar
[ndl@vanity ~]$

? (asdf-install:install :zpng)
...
;;; ASDF-INSTALL: Installing ZPNG in /home/ndl/.asdf-install-dir/site/,
    /home/ndl/.asdf-install-dir/systems/
; Warning: Cannot find tar command NIL.
; While executing: ASDF-INSTALL::EXTRACT-USING-TAR, in process listener(1).
> Error: Unable to extract tarball
    /home/ndl/asdf-install-1.asdf-install-tmp.
> While executing: ASDF-INSTALL::EXTRACT, in process listener(1).
```

```
> Type :POP to abort, :R for a list of available restarts.
> Type :? for other options.
1 > asdf-install:*shell-search-paths*
((:ABSOLUTE "bin") (:ABSOLUTE "usr" "bin"))
1 > (push '(:absolute "usr" "local" "bin")
          asdf-install:*shell-search-paths*)
((:ABSOLUTE "usr" "local" "bin") (:ABSOLUTE "bin") (:ABSOLUTE "usr" "bin"))
1 > :a
? (asdf-install:install :zpng)
...                                ; should work better this time
```

Next time you start your Lisp,

- Make sure the "systems" directory from the installation step (if—say—in the above I'd chosen "personal installation" then this would have been */home/ndl/.asdf-install-dir/systems/*) is in your `*central-registry*`, and
- Use ASDF to load the system:

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

The Key Check

This is the tricky part. ASDF-Install expects that for every library you download: a PGP signature can be located, the signature can be located on your computer, there's a trust relationship between you and the package signer, and the signer is on your "personal list of valid suppliers of Lisp code". If any of these steps fail, you'll be offered restarts which include not checking the signature for this particular package. If the last step fails, you'll also be offered the option to add the signer to your suppliers list:

```
>   Type (:C <n>) to invoke one of the following restarts:
0. Return to break level 1.
1. #<RESTART ABORT-BREAK #x284E2916>
2. Add to package supplier list
3. Don't check GPG signature for this package
4. Retry GPG check (e.g., after downloading the key)
5. Return to toplevel.
6. #<RESTART ABORT-BREAK #x284E2CA6>
7. Reset this thread
8. Kill this thread
1 > (:c 2)
Invoking restart: Add to package supplier list
;;; ASDF-INSTALL: Installing ZPNG in /home/ndl/.asdf-install-dir/site/,
    /home/ndl/.asdf-install-dir/systems/
...
```

Exercise

Two of these restarts printed horribly. Find out why and fix it.

For help obtaining developers' public keys, visit:

http://www.cliki.net/ASDF-Install and GPG

Publishing a Library

An ASDF-installable library:

- must be bundled as a gzipped tar archive (`tar -cfz`), which unpacks into a single top-level directory (and, optionally, subdirectories);
- must contain a system definition file whose name corresponds to that of the library (e.g. *zpng.asd* for the `zpng` library) and this file must live in the top-level directory;
- must be accompanied by a PGP signature; and
- must be posted on the CLiki.

For full details, visit

http://www.cliki.net/ASDF-Install

and

http://www.cliki.net/GPG for ASDF-Install developers

Alternatives to ASDF

Two of the proprietary Lisps (Allegro CL and LispWorks) ship with their own defsystem implementations. These are mature, stable libraries compared to which ASDF is something of a recent arrival. If you're targeting an application for one of these Lisps and have no intention of porting it elsewhere then you might do better to use the proprietary defsystem.

- The libraries are fully integrated into the implementation and are commercially supported. If you're using one of these Lisps anyway you might as well take advantage of these.
- Both libraries support additional features and in particular `concatenate-system`, a function for assembling the components of a system into a single fasl file. This is equivalent to the "unified fasls" extension to ASDF supported by ECL. If you want to distribute libraries in binary form you may find this useful.
- On the other hand, the notion of a `*central-registry*` is peculiar to ASDF: you'll have to load the defsystem files yourself (and reload them if you make any changes to them).

Read your documentation for specifics; after this chapter you're unlikely to encounter any concepts which are wholly unfamiliar.

Open source "traditional" counterparts to ASDF and the proprietary defsystems include:

- The *CLIM-SYS minimal defsystem* from *http://lisp.erire.com/clim-sys/defsystem.lisp*
- *MK-defsystem* from *http://www.cliki.net/mk-defsystem*

MK-defsystem has been around since the early 1990s. If you're working with libraries which predate ASDF you may well find yourself using MK-defsystem to load them. Note that ASDF-Install supports systems defined by MK-defsystem.

And finally, two different approaches:

- *clbuild* (*http://common-lisp.net/project/clbuild/*) is a shell script roughly equivalent in intent to the download/install phase of ASDF-Install. (In other words, it does not compile anything.) A list of about 140 libraries and their locations is hardwired in, so you don't have to worry about PGP keys.
- *XCVB* seeks to avoid compile-time side-effects altogether. In a new and interesting departure from widespread Common Lisp practice it compiles each system component in a fresh environment; it may scale very well to very large systems. The project homepage (*http://common-lisp.net/project/xcvb/*) speaks of "deterministic separate compilation and enforced locally-declared dependencies".