ASDF 3, or Why Lisp is Now an Acceptable Scripting Language

François-René Rideau
Google
tunes@google.com

Abstract

ASDF, the *de facto* standard build system for Common Lisp, has been vastly improved between 2012 and 2014. These and other improvements finally bring Common Lisp up to par with "scripting languages" in terms of ease of writing and *deploying* portable code that can access and "glue" together functionality from the underlying system or external programs. "Scripts" can thus be written in Common Lisp, and take advantage of its expressive power, well-defined semantics, and efficient implementations. We describe the most salient improvements in ASDF 3 and how they enable previously difficult and portably impossible uses of the programming language. We discuss past and future challenges in improving this key piece of software infrastructure, and what approaches did or didn't work in bringing change to the Common Lisp community.

Introduction

As of 2013, one can use Common Lisp (CL) to *portably* write the programs for which one traditionally uses so-called "scripting" languages: one can write small scripts that glue together functionality provided by the operating system (OS), external programs, C libraries, or network services; one can scale them into large, maintainable and modular systems; and one can make those new services available to other programs via the command-line as well as via network protocols, etc.

The last barrier to making that possible was the lack of a portable way to build and deploy code so a same script can run *unmodified* for many users on one or many machines using one or many different compilers. This was solved by ASDF 3.

ASDF has been the *de facto* standard build system for portable CL software since shortly after its release by Dan Barlow in 2002 (Barlow 2004). **The purpose of a build system is to enable division of labor in software development**: source code is organized in separately-developed components that depend on other components, and the build system transforms the transitive closure of these components into a working program.

ASDF 3 is the latest rewrite of the system. Aside from fixing numerous bugs, it sports a new portability layer. One can now use ASDF to write Lisp programs that may be invoked from the command line or may spawn external programs and capture their output ASDF can deliver these programs as standalone executable files; moreover the companion script `cl-launch` (see section 2.9) can create light-weight scripts that can be run unmodified on many different kinds of machines, each differently configured. These features make portable scripting possible. Previously, key parts of a program had to be configured to match one's specific CL implementation, OS, and software installation paths. Now, all of one's usual scripting needs can be entirely fulfilled using CL, benefitting from its efficient implementations, hundreds of software libraries, etc.

In this article, we discuss how the innovations in ASDF 3 enable new kinds of software development in CL. In section 1, we explain what ASDF is about; we compare it to common practice in the C world. In section 2, we describe the improvements introduced in ASDF 3 and ASDF 3.1 to solve the problem of software delivery; this section requires some familiarity with CL. In section 3, we discuss the challenges of evolving a piece of community software, concluding with lessons learned from our experience.

This is the short version of this article. It sometimes refers to appendices present only in the extended version (Rideau 2014), that also includes a few additional examples and footnotes.

1. What ASDF is

1.1 ASDF: Basic Concepts

1.1.1 Components

ASDF is a build system for CL: it helps developers divide software into a hierarchy of *component*s and automatically generates a working program from all the source code.

Top components are called *system*s in an age-old Lisp tradition, while the bottom ones are source *file*s, typically written in CL. In between, there may be a recursive hierarchy of *module*s.

Users may then `operate` on these components with various build *operation*s, most prominently compiling the source code (operation `compile-op`) and loading the output into the current Lisp image (operation `load-op`).

Several related systems may be developed together in the same source code *project*. Each system may depend on code from other systems, either from the same project or from a different project. ASDF itself has no notion of projects, but other tools on top of ASDF do: Quicklisp (Beane 2011) packages together systems from a project into a *release*, and provides hundreds of releases as a *distribution*, automatically downloading on demand required systems and all their transitive dependencies.

Further, each component may explicitly declare a *dependency* on other components: whenever compiling or loading a component relies on declarations or definitions of packages, macros, variables, classes, functions, etc., present in another component, the programmer must declare that the former component *depends-on* the latter.

1.1.2 Example System Definition

Below is how the `fare-quasiquote` system is defined (with elisions) in a file `fare-quasiquote.asd`. It contains three files, `packages`, `quasiquote` and `pp-quasiquote` (the `.lisp` suffix is automatically added based on the component class; see Appendix C). The latter files each depend on the first file, because this former file defines the CL packages[1]:

[1] Packages are namespaces that contain symbols; they need to be created before the symbols they contain may even be read as valid syntax.

```
(defsystem "fare-quasiquote" ...
  :depends-on ("fare-utils")
  :components
  ((:file "packages")
   (:file "quasiquote"
    :depends-on ("packages"))
   (:file "pp-quasiquote"
    :depends-on ("quasiquote"))))
```

Among the elided elements were metadata such as `:license "MIT"`, and extra dependency information `:in-order-to ((test-op (test-op "fare-quasiquote-test")))`, that delegates testing the current system to running tests on another system. Notice how the system itself *depends-on* another system, `fare-utils`, a collection of utility functions and macros from another project, whereas testing is specified to be done by `fare-quasiquote-test`, a system defined in a different file, `fare-quasiquote-test.asd`, within the same project.

1.1.3 Action Graph

The process of building software is modeled as a Directed Acyclic Graph (DAG) of *action*s, where *each action is a pair of an operation and a component.* The DAG defines a partial order, whereby each action must be *perform*ed, but only after all the actions it (transitively) *depends-on* have already been performed.

For instance, in `fare-quasiquote` above, the *loading* of (the output of compiling) `quasiquote` *depends-on* the *compiling* of `quasiquote`, which itself depends-on the *loading* of (the output of compiling) `package`, etc.

Importantly, though, this graph is distinct from the preceding graph of components: the graph of actions isn't a mere refinement of the graph of components but a transformation of it that also incorporates crucial information about the structure of operations.

`ASDF` extracts from this DAG a *plan*, which by default is a topologically sorted list of actions, that it then *perform*s in order, in a design inspired by Pitman (Pitman 1984).

Users can extend `ASDF` by defining new subclasses of `operation` and/or `component` and the methods that use them, or by using global, per-system, or per-component hooks.

1.1.4 In-image

`ASDF` is an "in-image" build system, in the Lisp `defsystem` tradition: it compiles (if necessary) and loads software into the current CL image, and can later update the current image by recompiling and reloading the components that have changed. For better and worse, this notably differs from common practice in most other languages, where the build system is a completely different piece of software running in a separate process.[2] On the one hand, it minimizes overhead to writing build system extensions. On the other hand, it puts great pressure on `ASDF` to remain minimal.

Qualitatively, `ASDF` must be delivered as a single source file and cannot use any external library, since it itself defines the code that may load other files and libraries. Quantitatively, `ASDF` must minimize its memory footprint, since it's present in all programs that are built, and any resource spent is paid by each program.

For all these reasons, `ASDF` follows the minimalist principle that **anything that can be provided as an extension should be provided as an extension and left out of the core**. Thus it cannot afford to support a persistence cache indexed by the cryptographic digest of build expressions, or a distributed network of workers, etc. However, these could conceivably be implemented as `ASDF` extensions.

[2] Of course, a build system could compile CL code in separate processes, for the sake of determinism and parallelism: our XCVB did (Brody 2009); so does the Google build system.

1.2 Comparison to C programming practice

Most programmers are familiar with C, but not with CL. It's therefore worth contrasting `ASDF` to the tools commonly used by C programmers to provide similar services. Note though how these services are factored in very different ways in CL and in C.

To build and load software, C programmers commonly use `make` to build the software and `ld.so` to load it. Additionally, they use a tool like `autoconf` to locate available libraries and identify their features. In many ways these C solutions are better engineered than `ASDF`. But in other important ways `ASDF` demonstrates how these C systems have much accidental complexity that CL does away with thanks to better architecture.

- Lisp makes the full power of runtime available at compile-time, so it's easy to implement a Domain-Specific Language (DSL): the programmer only needs to define new functionality, as an extension that is then seamlessly combined with the rest of the language, including other extensions. In C, the many utilities that need a DSL must grow it onerously from scratch; since the domain expert is seldom also a language expert with resources to do it right, this means plenty of mutually incompatible, misdesigned, power-starved, misimplemented languages that have to be combined through an unprincipled chaos of expensive yet inexpressive means of communication.
- Lisp provides full introspection at runtime and compile-time alike, as well as a protocol to declare *features* and conditionally include or omit code or data based on them. Therefore you don't need dark magic at compile-time to detect available features. In C, people resort to horribly unmaintainable configuration scripts in a hodge podge of shell script, `m4` macros, C preprocessing and C code, plus often bits of `python`, `perl`, `sed`, etc.
- `ASDF` possesses a standard and standardly extensible way to configure where to find the libraries your code depends on, further improved in `ASDF` 2. In C, there are tens of incompatible ways to do it, between `libtool`, `autoconf`, `kde-config`, `pkg-config`, various manual `./configure` scripts, and countless other protocols, so that each new piece of software requires the user to learn a new *ad hoc* configuration method, making it an expensive endeavor to use or distribute libraries.
- `ASDF` uses the very same mechanism to configure both runtime and compile-time, so there is only one configuration mechanism to learn and to use, and minimal discrepancy.[3] In C, completely different, incompatible mechanisms are used at runtime (`ld.so`) and compile-time (unspecified), which makes it hard to match source code, compilation headers, static and dynamic libraries, requiring complex "software distribution" infrastructures (that admittedly also manage versioning, downloading and precompiling); this at times causes subtle bugs when discrepancies creep in.

Nevertheless, there are also many ways in which `ASDF` pales in comparison to other build systems for CL, C, Java, or other systems:

- `ASDF` isn't a general-purpose build system. Its relative simplicity is directly related to it being custom made to build CL software only. Seen one way, it's a sign of how little you can get away with if you have a good basic architecture; a similarly simple solution isn't available to most other programming languages, that require much more complex tools to achieve a similar purpose. Seen another way, it's also the CL community

[3] There is still discrepancy *inherent* with these times being distinct: the installation or indeed the machine may have changed.

failing to embrace the outside world and provide solutions with enough generality to solve more complex problems.

- At the other extreme, a build system for CL could have been made that is much simpler and more elegant than ASDF, if it could have required software to follow some simple organization constraints, without much respect for legacy code. A constructive proof of that is quick-build (Bridgewater 2012), being a fraction of the size of ASDF, itself a fraction of the size of ASDF 3, and with a fraction of the bugs — but none of the generality and extensibility (See section 2.10).
- ASDF it isn't geared at all to build large software in modern adversarial multi-user, multi-processor, distributed environments where source code comes in many divergent versions and in many configurations. It is rooted in an age-old model of building software in-image, what's more in a traditional single-processor, single-machine environment with a friendly single user, a single coherent view of source code and a single target configuration. The new ASDF 3 design is consistent and general enough that it could conceivably be made to scale, but that would require a lot of work.

2. ASDF 3: A Mature Build

2.1 A Consistent, Extensible Model

Surprising as it may be to the CL programmers who used it daily, there was an essential bug at the heart of ASDF: it didn't even try to propagate timestamps from one action to the next. And yet it worked, mostly. The bug was present from the very first day in 2001, and even before in mk-defsystem since 1990 (Kantrowitz 1990), and it survived till December 2012, despite all our robustification efforts since 2009 (Goldman 2010). Fixing it required a complete rewrite of ASDF's core.

As a result, the object model of ASDF became at the same time more powerful, more robust, and simpler to explain. The dark magic of its traverse function is replaced by a well-documented algorithm. It's easier than before to extend ASDF, with fewer limitations and fewer pitfalls: users may control how their operations do or don't propagate along the component hierarchy. Thus, ASDF can now express arbitrary action graphs, and could conceivably be used in the future to build more than just CL programs.

The proof of a good design is in the ease of extending it. And in CL, extension doesn't require privileged access to the code base. We thus tested our design by adapting the most elaborate existing ASDF extensions to use it. The result was indeed cleaner, eliminating the previous need for overrides that redefined sizable chunks of the infrastructure. Chronologically, however, we consciously started this porting process in interaction with developing ASDF 3, thus ensuring ASDF 3 had all the extension hooks required to avoid redefinitions.

See the entire story in Appendix F.

2.2 Bundle Operations

Bundle operations create a single output file for an entire system or collection of systems. The most directly user-facing bundle operations are compile-bundle-op and load-bundle-op: the former bundles into a single compilation file all the individual outputs from the compile-op of each source file in a system; the latter loads the result of the former. Also lib-op links into a library all the object files in a system and dll-op creates a dynamically loadable library out of them. The above bundle operations also have so-called *monolithic* variants that bundle all the files in a system *and all its transitive dependencies*.

Bundle operations make delivery of code much easier. They were initially introduced as asdf-ecl, an extension to ASDF specific to the implementation ECL, back in the day of ASDF 1. asdf-ecl was distributed with ASDF 2, though in a way that made upgrade slightly awkward to ECL users, who had to explicitly reload it after upgrading ASDF, even though it was included by the initial (require "asdf"). In 2012, it was generalized to other implementations as the external system asdf-bundle. It was then merged into ASDF during the development of ASDF 3: not only did it provide useful new operations, but the way that ASDF 3 was automatically upgrading itself for safety purposes (see Appendix B) would otherwise have broken things badly for ECL users if the bundle support weren't itself bundled with ASDF.

In ASDF 3.1, using deliver-asd-op, you can create both the bundle from compile-bundle-op and an .asd file to use to deliver the system in binary format only.

2.3 Understandable Internals

After bundle support was merged into ASDF (see section 2.2 above), it became trivial to implement a new concatenate-source-op operation. Thus ASDF could be developed as multiple files, which would improve maintainability. For delivery purpose, the source files would be concatenated in correct dependency order, into the single file asdf.lisp required for bootstrapping.

The division of ASDF into smaller, more intelligible pieces had been proposed shortly after we took over ASDF; but we had rejected the proposal then on the basis that ASDF must not depend on external tools to upgrade itself from source, another strong requirement (see Appendix B). With concatenate-source-op, an external tool wasn't needed for delivery and regular upgrade, only for bootstrap. Meanwhile this division had also become more important, since ASDF had grown so much, having almost tripled in size since those days, and was promising to grow some more. It was hard to navigate that one big file, even for the maintainer, and probably impossible for newcomers to wrap their head around it.

To bring some principle to this division, we followed the principle of one file, one package, as demonstrated by faslpath (Etter 2009) and quick-build (Bridgewater 2012), though not yet actively supported by ASDF itself (see section 2.10). This programming style ensures that files are indeed providing related functionality, only have explicit dependencies on other files, and don't have any forward dependencies without special declarations. Indeed, this was a great success in making ASDF understandable, if not by newcomers, at least by the maintainer himself; this in turn triggered a series of enhancements that would not otherwise have been obvious or obviously correct, illustrating the principle that **good code is code you can understand, organized in chunks you can each fit in your brain**.

2.4 Package Upgrade

Preserving the hot upgradability of ASDF was always a strong requirement (see Appendix B). In the presence of this package refactoring, this meant the development of a variant of CL's defpackage that plays nice with hot upgrade: define-package. Whereas the former isn't guaranteed to work and may signal an error when a package is redefined in incompatible ways, the latter will update an old package to match the new desired definition while recycling existing symbols from that and other packages.

Thus, in addition to the regular clauses from defpackage, define-package accepts a clause :recycle: it attempts to recycle each declared symbol from each of the specified packages in the given order. For idempotence, the package itself must be the first in the list. For upgrading from an old ASDF, the :asdf package is always named last. The default recycle list consists in a list of the package and its nicknames.

New features also include :mix and :reexport. :mix mixes imported symbols from several packages: when multiple

packages export symbols with the same name, the conflict is automatically resolved in favor of the package named earliest, whereas an error condition is raised when using the standard `:use` clause. `:reexport` reexports the same symbols as imported from given packages, and/or exports instead the same-named symbols that shadow them. ASDF 3.1 adds `:mix-reexport` and `:use-reexport`, which combine `:reexport` with `:mix` or `:use` in a single statement, which is more maintainable than repeating a list of packages.

2.5 Portability Layer

Splitting ASDF into many files revealed that a large fraction of it was already devoted to general purpose utilities. This fraction only grew under the following pressures: a lot of opportunities for improvement became obvious after dividing ASDF into many files; features added or merged in from previous extensions and libraries required new general-purpose utilities; as more tests were added for new features, and were run on all supported implementations, on multiple operating systems, new portability issues cropped up that required development of robust and portable abstractions.

The portability layer, after it was fully documented, ended up being slightly bigger than the rest of ASDF. Long before that point, ASDF was thus formally divided in two: this portability layer, and the `defsystem` itself. The portability layer was initially dubbed `asdf-driver`, because of merging in a lot of functionality from `xcvb-driver`. Because users demanded a shorter name that didn't include ASDF, yet would somehow be remindful of ASDF, it was eventually renamed UIOP: the Utilities for Implementation- and OS- Portability[4]. It was made available separately from ASDF as a portability library to be used on its own; yet since ASDF still needed to be delivered as a single file `asdf.lisp`, UIOP was *transcluded* inside that file, now built using the `monolithic-concatenate-source-op` operation. At Google, the build system actually uses UIOP for portability without the rest of ASDF; this led to UIOP improvements that will be released with ASDF 3.1.2.

Most of the utilities deal with providing sane pathname abstractions (see Appendix C), filesystem access, sane input/output (including temporary files), basic operating system interaction — many things for which the CL standard lacks. There is also an abstraction layer over the less-compatible legacy implementations, a set of general-purpose utilities, and a common core for the ASDF configuration DSLs.[5] Importantly for a build system, there are portable abstractions for compiling CL files while controlling all the warnings and errors that can occur, and there is support for the life-cycle of a Lisp image: dumping and restoring images, initialization and finalization hooks, error handling, backtrace display, etc. However, the most complex piece turned out to be a portable implementation of `run-program`.

2.6 `run-program`

With ASDF 3, you can run external commands as follows:

```
(run-program `("cp" "-lax" "--parents"
               "src/foo" ,destination))
```

On Unix, this recursively hardlinks files in directory `src/foo` into a directory named by the string `destination`, preserving the prefix `src/foo`. You may have to add `:output t :error-output t` to get error messages on your `*standard-output*` and `*error-output*` streams, since the default value, `nil`, designates `/dev/null`. If the invoked program returns an error code, `run-program` signals a structured CL `error`, unless you specified `:ignore-error-status t`.

This utility is essential for ASDF extensions and CL code in general to portably execute arbitrary external programs. It was a challenge to write: Each implementation provided a different underlying mechanism with wildly different feature sets and countless corner cases. The better ones could fork and exec a process and control its standard-input, standard-output and error-output; lesser ones could only call the `system(3)` C library function. Moreover, Windows support differed significantly from Unix. ASDF 1 itself actually had a `run-shell-command`, initially copied over from `mk-defsystem`, but it was more of an attractive nuisance than a solution, despite our many bug fixes: it was implicitly calling `format`; capturing output was particularly contrived; and what shell would be used varied between implementations, even more so on Windows.

ASDF 3's `run-program` is full-featured, based on code originally from XCVB's `xcvb-driver` (Brody 2009). It abstracts away all these discrepancies to provide control over the program's standard-output, using temporary files underneath if needed. Since ASDF 3.0.3, it can also control the standard-input and error-output. It accepts either a list of a program and arguments, or a shell command string. Thus your previous program could have been:

```
(run-program
 (format nil "cp -lax --parents src/foo ~S"
         (native-namestring destination))
 :output t :error-output t)
```

where (UIOP)'s `native-namestring` converts the `pathname` object `destination` into a name suitable for use by the operating system, as opposed to a CL `namestring` that might be escaped somehow.

You can also inject input and capture output:

```
(run-program '("tr" "a-z" "n-za-m")
 :input '("uryyb, jbeyq") :output :string)
```

returns the string `"hello, world"`. It also returns secondary and tertiary values `nil` and `0` respectively, for the (non-captured) error-output and the (successful) exit code.

`run-program` only provides a basic abstraction; a separate system `inferior-shell` was written on top of UIOP, and provides a richer interface, handling pipelines, `zsh` style redirections, splicing of strings and/or lists into the arguments, and implicit conversion of pathnames into native-namestrings, of symbols into downcased strings, of keywords into downcased strings with a `--` prefix. Its short-named functions `run`, `run/nil`, `run/s`, `run/ss`, respectively run the external command with outputs to the Lisp standard- and error- output, with no output, with output to a string, or with output to a stripped string. Thus you could get the same result as previously with:

```
(run/ss '(pipe (echo (uryyb ", " jbeyq))
               (tr a-z (n-z a-m))))
```

Or to get the number of processors on a Linux machine, you can:

```
(run '(grep -c "^processor.:"
            (< /proc/cpuinfo))
     :output #'read)
```

2.7 Configuration Management

ASDF always had minimal support for configuration management. ASDF 3 doesn't introduce radical change, but provides more usable replacements or improvements for old features.

[4] U, I, O and P are also the four letters that follow QWERTY on an anglo-saxon keyboard.

[5] ASDF 3.1 notably introduces a `nest` macro that nests arbitrarily many forms without indentation drifting ever to the right. It makes for more readable code without sacrificing good scoping discipline.

For instance, ASDF 1 had always supported version-checking: each component (usually, a system) could be given a version string with e.g. `:version "3.1.0.97"`, and ASDF could be told to check that dependencies of at least a given version were used, as in `:depends-on ((:version "inferior-shell" "2.0.0"))`. This feature can detect a dependency mismatch early, which saves users from having to figure out the hard way that they need to upgrade some libraries, and which.

Now, ASDF always required components to use "semantic versioning", where versions are strings made of dot-separated numbers like `3.1.0.97`. But it didn't enforce it, leading to bad surprises for the users when the mechanism was expected to work, but failed. ASDF 3 issues a `warning` when it finds a version that doesn't follow the format. It would actually have issued an `error`, if that didn't break too many existing systems.

Another problem with version strings was that they had to be written as literals in the `.asd` file, unless that file took painful steps to extract it from another source file. While it was easy for source code to extract the version from the system definition, some authors legitimately wanted their code to not depend on ASDF itself. Also, it was a pain to repeat the literal version and/or the extraction code in every system definition in a project. ASDF 3 can thus extract version information from a file in the source tree, with, e.g. `:version (:read-file-line "version.text")` to read the version as the first line of file `version.text`. To read the third line, that would have been `:version (:read-file-line "version.text" :at 2)` (mind the off-by-one error in the English language). Or you could extract the version from source code. For instance, `poiu.asd` specifies `:version (:read-file-form "poiu.lisp" :at (1 2 2))` which is the third subform of the third subform of the second form in the file `poiu.lisp`. The first form is an `in-package` and must be skipped. The second form is an `(eval-when (...) body...)` the body of which starts with a `(def-parameter *poiu-version* ...)` form. ASDF 3 thus solves this version extraction problem for all software — except itself, since its own version has to be readable by ASDF 2 as well as by who views the single delivery file; thus its version information is maintained by a management script using regexps, of course written in CL.

Another painful configuration management issue with ASDF 1 and 2 was lack of a good way to conditionally include files depending on which implementation is used and what features it supports. One could always use CL reader conditionals such as `#+(or sbcl clozure)` but that means that ASDF could not even see the components being excluded, should some operation be invoked that involves printing or packaging the code rather than compiling it — or worse, should it involve cross-compilation for another implementation with a different feature set. There was an obscure way for a component to declare a dependency on a `:feature`, and annotate its enclosing module with `:if-component-dep-fails :try-next` to catch the failure and keep trying. But the implementation was a kluge in `traverse` that short-circuited the usual dependency propagation and had exponential worst case performance behavior when nesting such pseudo-dependencies to painfully emulate feature expressions.

ASDF 3 gets rid of `:if-component-dep-fails`: it didn't fit the fixed dependency model at all. A limited compatibility mode without nesting was preserved to keep processing old versions of SBCL. As a replacement, ASDF 3 introduces a new option `:if-feature` in component declarations, such that a component is only included in a build plan if the given feature expression is true during the planning phase. Thus a component annotated with `:if-feature (:and :sbcl (:not :sb-unicode))` (and its children, if any) is only included on an SBCL without Unicode support. This is more expressive than what preceded, without requiring inconsistencies in the dependency model, and without pathological performance behavior.

2.8 Standalone Executables

One of the bundle operations contributed by the ECL team was `program-op`, that creates a standalone executable. As this was now part of ASDF 3, it was only natural to bring other ASDF-supported implementations up to par: CLISP, Clozure CL, CMUCL, LispWorks, SBCL, SCL. Thus `UIOP` features a `dump-image` function to dump the current heap image, except for ECL and its successors that follow a linking model and use a `create-image` function. These functions were based on code from `xcvb-driver`, which had taken them from `cl-launch`.

ASDF 3 also introduces a `defsystem` option to specify an entry point as e.g. `:entry-point "my-package:entry-point"`. The specified function (designated as a string to be read after the package is created) is called without arguments after the program image is initialized; after doing its own initializations, it can explicitly consult `*command-line-arguments*`[6] or pass it as an argument to some main function.

Our experience with a large application server at ITA Software showed the importance of hooks so that various software components may modularly register finalization functions to be called before dumping the image, and initialization functions to be called before calling the entry point. Therefore, we added support for image life-cycle to `UIOP`. We also added basic support for running programs non-interactively as well as interactively: non-interactive programs exit with a backtrace and an error message repeated above and below the backtrace, instead of inflicting a debugger on end-users; any non-`nil` return value from the entry-point function is considered success and `nil` failure, with an appropriate program exit status.

Starting with ASDF 3.1, implementations that don't support standalone executables may still dump a heap image using the `image-op` operation, and a wrapper script, e.g. created by `cl-launch`, can invoke the program; delivery is then in two files instead of one. `image-op` can also be used by all implementations to create intermediate images in a staged build, or to provide ready-to-debug images for otherwise non-interactive applications.

2.9 `cl-launch`

Running Lisp code to portably create executable commands from Lisp is great, but there is a bootstrapping problem: when all you can assume is the Unix shell, how are you going to portably invoke the Lisp code that creates the initial executable to begin with?

We solved this problem some years ago with `cl-launch`. This bilingual program, both a portable shell script and a portable CL program, provides a nice colloquial shell command interface to building shell commands from Lisp code, and supports delivery as either portable shell scripts or self-contained precompiled executable files.

Its latest incarnation, `cl-launch` 4 (March 2014), was updated to take full advantage of ASDF 3. Its build specification interface was made more general, and its Unix integration was improved. You may thus invoke Lisp code from a Unix shell:

```
cl -sp lisp-stripper \
   -i "(print-loc-count \"asdf.lisp\")"
```

You can also use `cl-launch` as a script "interpreter", except that it invokes a Lisp compiler underneath:

[6] In CL, most variables are lexically visible and statically bound, but *special* variables are globally visible and dynamically bound. To avoid subtle mistakes, the latter are conventionally named with enclosing asterisks, also known in recent years as *earmuffs*.

```
#!/usr/bin/cl -sp lisp-stripper -E main
(defun main (argv)
  (if argv
      (map () 'print-loc-count argv)
      (print-loc-count *standard-input*)))
```

In the examples above, option `-sp`, shorthand for `--system-package`, simultaneously loads a system using ASDF during the build phase, and appropriately selects the current package; `-i`, shorthand for `--init` evaluates a form at the start of the execution phase; `-E`, shorthand for `--entry` configures a function that is called after init forms are evaluated, with the list of command-line arguments as its argument.[7] As for `lisp-stripper`, it's a simple library that counts lines of code after removing comments, blank lines, docstrings, and multiple lines in strings.

`cl-launch` automatically detects a CL implementation installed on your machine, with sensible defaults. You can easily override all defaults with a proper command-line option, a configuration file, or some installation-time configuration. See `cl-launch --more-help` for complete information. Note that `cl-launch` is on a bid to homestead the executable path `/usr/bin/cl` on Linux distributions; it may slightly more portably be invoked as `cl-launch`.

A nice use of `cl-launch` is to compare how various implementations evaluate some form, to see how portable it is in practice, whether the standard mandates a specific result or not:

```
for l in sbcl ccl clisp cmucl ecl abcl \
         scl allegro lispworks gcl xcl ; do
 cl -l $l -i \
 '(format t "'$l': ~S~%" `#5(1 ,@`(2 3)))' \
 2>&1 | grep "^$l:" # LW, GCL are verbose
done
```

`cl-launch` compiles all the files and systems that are specified, and keeps the compilation results in the same output-file cache as ASDF 3, nicely segregating them by implementation, version, ABI, etc. Therefore, the first time it sees a given file or system, or after they have been updated, there may be a startup delay while the compiler processes the files; but subsequent invocations will be faster as the compiled code is directly loaded. This is in sharp contrast with other "scripting" languages, that have to slowly interpret or recompile everytime. For security reasons, the cache isn't shared between users.

2.10 package-inferred-system

ASDF 3.1 introduces a new extension `package-inferred-system` that supports a one-file, one-package, one-system style of programming. This style was pioneered by `faslpath` (Etter 2009) and more recently `quick-build` (Bridgewater 2012). This extension is actually compatible with the latter but not the former, for ASDF 3.1 and `quick-build` use a slash `"/"` as a hierarchy separator where `faslpath` used a dot `"."`.

This style consists in every file starting with a `defpackage` or `define-package` form; from its `:use` and `:import-from` and similar clauses, the build system can identify a list of packages it depends on, then map the package names to the names of systems and/or other files, that need to be loaded first. Thus package name `lil/interface/all` refers to the file `interface/all.lisp` under the hierarchy registered by system `lil`, defined as follows in `lil.asd` as using class `package-inferred-system`:

```
(defsystem "lil" ...
  :description "LIL: Lisp Interface Library"
  :class :package-inferred-system
  :defsystem-depends-on ("asdf-package-system")
  :depends-on ("lil/interface/all"
               "lil/pure/all" ...)
  ...)
```

The `:defsystem-depends-on ("asdf-package-system")` is an external extension that provides backward compatibility with ASDF 3.0, and is part of Quicklisp. Because not all package names can be directly mapped back to a system name, you can register new mappings for `package-inferred-system`. The `lil.asd` file may thus contain forms such as:

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

Then, a 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 ...))
```

This style provides many maintainability benefits: by imposing upon programmers a discipline of smaller namespaces, with explicit dependencies and especially explicit forward dependencies, the style encourages good factoring of the code into coherent units; by contrast, the traditional style of "everything in one package" has low overhead but doesn't scale very well. ASDF itself was rewritten in this style as part of ASDF 2.27, the initial ASDF 3 pre-release, with very positive results.

Since it depends on ASDF 3, `package-inferred-system` isn't as lightweight as `quick-build`, which is almost two orders of magnitude smaller than ASDF 3. But it does interoperate perfectly with the rest of ASDF, from which it inherits the many features, the portability, and the robustness.

2.11 Restoring Backward Compatibility

ASDF 3 had to break compatibility with ASDF 1 and 2: all operations used to be propagated *sideway* and *downward* along the component DAG (see Appendix F). In most cases this was undesired; indeed, ASDF 3 is predicated upon a new operation `prepare-op` that instead propagates *upward*.[8] Most existing ASDF extensions thus included workarounds and approximations to deal with the issue. But a handful of extensions did expect this behavior, and now they were broken.

Before the release of ASDF 3, authors of all known ASDF extensions distributed by Quicklisp had been contacted, to make their code compatible with the new fixed model. But there was no way to contact unidentified authors of proprietary extensions, beside sending an announcement to the mailing-list. Yet, whatever message was sent didn't attract enough attention. Even our co-maintainer Robert Goldman got bitten hard when an extension used at work stopped working, wasting days to figure out the issue.

Therefore, ASDF 3.1 features enhanced backward-compatibility. The class `operation` implements sideway and downward propagation on all classes that do not explicitly inherit from any

[7] Several systems are available to help you define an evaluator for your command-line argument DSL: `command-line-arguments`, `clon`, `lisp-gflags`.

[8] Sideway means the action of operation `o` on component `c` *depends-on* the action of `o` (or another operation) on each of the declared dependencies of `c`. Downward means that it *depends-on* the action of `o` on each of `c`'s children; upward, on `c`'s parent (enclosing module or system).

of the propagating mixins `downward-operation`, `upward-operation`, `sideway-operation` or `selfward-operation`, unless they explicitly inherit from the new mixin `non-propagating-operation`. `ASDF` 3.1 signals a `warning` at runtime when an operation class is instantiated that doesn't inherit from any of the above mixins, which will hopefully tip off authors of a proprietary extension that it's time to fix their code. To tell `ASDF` 3.1 that their operation class is up-to-date, extension authors may have to define their non-propagating operations as follows:

```
(defclass my-op (#+asdf3.1 non-propagating-
operation operation) ())
```

This is a case of "negative inheritance", a technique usually frowned upon, for the explicit purpose of backward compatibility. Now `ASDF` cannot use the CLOS Meta-Object Protocol (MOP), because it hasn't been standardized enough to be portably used without using an abstraction library such as `closer-mop`, yet `ASDF` cannot depend on any external library, and this is too small an issue to justify making a sizable MOP library part of `UIOP`. Therefore, the negative inheritance is implemented in an *ad hoc* way at runtime.

3. Code Evolution in a Conservative Community

3.1 Feature Creep? No, Mission Creep

Throughout the many features added and tenfold increase in size from `ASDF` 1 to `ASDF` 3, `ASDF` remained true to its minimalism — but the mission, relative to which the code remains minimal, was extended, several times: In the beginning, `ASDF` was the simplest extensible variant of `defsystem` that builds CL software (see Appendix A). With `ASDF` 2, it had to be upgradable, portable, modularly configurable, robust, performant, usable (see Appendix B). Then it had to be more declarative, more reliable, more predictable, and capable of supporting language extensions (see Appendix D). Now, `ASDF` 3 has to support a coherent model for representing dependencies, an alternative one-package-per-file style for declaring them, software delivery as either scripts or binaries, a documented portability layer including image life-cycle and external program invocation, etc. (see section 2).

3.2 Backward Compatibility is Social, not Technical

As efforts were made to improve `ASDF`, a constant constraint was that of *backward compatibility*: every new version of `ASDF` had to be compatible with the previous one, i.e. systems that were defined using previous versions had to keep working with new versions. But what more precisely is backward compatibility?

In an overly strict definition that precludes any change in behavior whatsoever, even the most uncontroversial bug fix isn't backward-compatible: any change, for the better as it may be, is incompatible, since by definition, some behavior has changed!

One might be tempted to weaken the constraint a bit, and define "backward compatible" as being the same as a "conservative extension": a *conservative extension* may fix erroneous situations, and give new meaning to situations that were previously undefined, but may not change the meaning of previously defined situations. Yet, this definition is doubly unsatisfactory. On the one hand, it precludes any amendment to previous bad decisions; hence, the jest **if it's not backwards, it's not compatible**. On the other hand, even if it only creates new situations that work correctly where they were previously in error, some existing analysis tool might assume these situations could never arise, and be confused when they now do.

Indeed this happened when `ASDF` 3 tried to better support *secondary systems*. `ASDF` looks up systems by name: if you try to load system `foo`, `ASDF` will search in registered directories for a file call `foo.asd`. Now, it was common practice that programmers may define multiple "secondary" systems in a same `.asd` file, such as a test system `foo-test` in addition to `foo`. This could lead to "interesting" situations when a file `foo-test.asd` existed, from a different, otherwise shadowed, version of the same library, resulting in a mismatch between the system and its tests. To make these situations less likely, `ASDF` 3 recommends that you name your secondary system `foo/test` instead of of `foo-test`, which should work just as well in `ASDF` 2, but with reduced risk of clash. Moreover, `ASDF` 3 can recognize the pattern and automatically load `foo.asd` when requested `foo/test`, in a way guaranteed not to clash with previous usage, since no directory could contain a file thus named in any modern operating system. In contrast, `ASDF` 2 has no way to automatically locate the `.asd` file from the name of a secondary system, and so you must ensure that you loaded the primary `.asd` file before you may use the secondary system. This feature may look like a textbook case of a backward-compatible "conservative extension". Yet, it's the major reason why Quicklisp itself still hasn't adopted `ASDF` 3: Quicklisp assumed it could always create a file named after each system, which happened to be true in practice (though not guaranteed) before this `ASDF` 3 innovation; systems that newly include secondary systems using this style break this assumption, and will require non-trivial work for Quicklisp to support.

What then, is backward compatibility? It isn't a technical constraint. **Backward compatibility is a social constraint**. The new version is backward compatible if the users are happy. This doesn't mean matching the previous version on all the mathematically conceivable inputs; it means improving the results for users on all the actual inputs they use; or providing them with alternate inputs they may use for improved results.

3.3 Weak Synchronization Requires Incremental Fixes

Even when some "incompatible" changes are not controversial, it's often necessary to provide temporary backward compatible solutions until all the users can migrate to the new design. Changing the semantics of one software system while other systems keep relying on it is akin to changing the wheels on a running car: you cannot usually change them all at once, at some point you must have both kinds active, and you cannot remove the old ones until you have stopped relying on them. Within a fast moving company, such migration of an entire code base can happen in a single checkin. If it's a large company with many teams, the migration can take many weeks or months. When the software is used by a weakly synchronized group like the CL community, the migration can take years.

When releasing `ASDF` 3, we spent a few months making sure that it would work with all publicly available systems. We had to fix many of these systems, but mostly, we were fixing `ASDF` 3 itself to be more compatible. Indeed, several intended changes had to be forsaken, that didn't have an incremental upgrade path, or for which it proved infeasible to fix all the clients.

A successful change was notably to modify the default encoding from the uncontrolled environment-dependent `:default` to the *de facto* standard `:utf-8`; this happened a year after adding support for encodings and `:utf-8` was added, and having forewarned community members of the future change in defaults, yet a few systems still had to be fixed (see Appendix D).

On the other hand, an unsuccessful change was the attempt to enable an innovative system to control warnings issued by the compiler. First, the `*uninteresting-conditions*` mechanism allows system builders to hush the warnings they know they don't care for, so that any compiler output is something they care for, and whatever they care for isn't drowned into a sea of uninteresting output. The mechanism itself is included in `ASDF` 3, but disabled by default, because there was no consensually agreeable value except an empty set, and no good way (so far) to configure it

both modularly and without pain. Second, another related mechanism that was similarly disabled is `deferred-warnings`, whereby `ASDF` can check warnings that are deferred by SBCL or other compilers until the end of the current *compilation-unit*. These warnings notably include forward references to functions and variables. In the previous versions of `ASDF`, these warnings were output at the end of the build the first time a file was built, but not checked, and not displayed afterward. If in `ASDF` 3 you `(uiop:enable-deferred-warnings)`, these warnings are displayed and checked every time a system is compiled or loaded. These checks help catch more bugs, but enabling them prevents the successful loading of a lot of systems in Quicklisp that have such bugs, even though the functionality for which these systems are required isn't affected by these bugs. Until there exists some configuration system that allows developers to run all these checks on new code without having them break old code, the feature will have to remain disabled by default.

3.4 Underspecification Creates Portability Landmines

The CL standard leaves many things underspecified about pathnames in an effort to define a useful subset common to many then-existing implementations and filesystems. However, the result is that portable programs can forever only access but a small subset of the complete required functionality. This result arguably makes the standard far less useful than expected (see Appendix C). The lesson is **don't standardize partially specified features**. It's better to standardize that some situations cause an error, and reserve any resolution to a later version of the standard (and then follow up on it), or to **delegate specification to other standards**, existing or future.

There could have been one pathname protocol per operating system, delegated to the underlying OS via a standard FFI. Libraries could then have sorted out portability over N operating systems. Instead, by standardizing only a common fragment and letting each of M implementations do whatever it can on each operating system, libraries now have to take into account N*M combinations of operating systems and implementations. In case of disagreement, it's much better to let each implementation's variant exist in its own, distinct namespace, which avoids any confusion, than have incompatible variants in the same namespace, causing clashes.

Interestingly, the aborted proposal for including `defsystem` in the CL standard was also of the kind that would have specified a minimal subset insufficient for large scale use while letting the rest underspecified. The CL community probably dodged a bullet thanks to the failure of this proposal.

3.5 Safety before Ubiquity

Guy Steele has been quoted as vaunting the programmability of Lisp's syntax by saying: *If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases.* Unhappily, if he were speaking about CL specifically, he would have had to add: *but it can't be the same as anyone else's*.

Indeed, syntax in CL is controlled via a fuzzy set of global variables, prominently including the `*readtable*`. Making non-trivial modifications to the variables and/or tables is possible, but letting these modifications escape is a serious issue; for the author of a system has no control over which systems will or won't be loaded before or after his system — this depends on what the user requests and on what happens to already have been compiled or loaded. Therefore in absence of further convention, it's always a bug to either rely on the syntax tables having non-default values from previous systems, or to inflict non-default values upon next systems. What is worse, changing syntax is only useful if it also happens at the interactive REPL and in appropriate editor buffers. Yet these interactive syntax changes can affect files built interactively, including, upon modification, components that do not depend on the syntax support, or worse, that the syntax support depends on; this can cause catastrophic circular dependencies, and require a fresh start after having cleared the output file cache. Systems like `named-readtables` or `cl-syntax` help with syntax control, but proper hygiene isn't currently enforced by either CL or `ASDF`, and remains up to the user, especially at the REPL.

Build support is therefore strongly required for safe syntax modification; but this build support isn't there yet in `ASDF` 3. For backward-compatibility reasons, `ASDF` will not enforce strict controls on the syntax, at least not by default. But it is easy to enforce hygiene by binding read-only copies of the standard syntax tables around each action. A more backward-compatible behavior is to let systems modify a shared readtable, and leave the user responsible for ensuring that all modifications to this readtable used in a given image are mutually compatible; `ASDF` can still bind the current `*readtable*` to that shared readtable around every compilation, to at least ensure that selection of an incompatible readtable at the REPL does not pollute the build. A patch to this effect is pending acceptance by the new maintainer.

Until such issues are resolved, even though the Lisp ideal is one of ubiquitous syntax extension, and indeed extension through macros is ubiquitous, extension though reader changes are rare in the CL community. This is in contrast with other Lisp dialects, such as Racket, that have succeeded at making syntax customization both safe and ubiquitous, by having it be strictly scoped to the current file or REPL. **Any language feature has to be safe before it may be ubiquitous**; if the semantics of a feature depend on circumstances beyond the control of system authors, such as the bindings of syntax variables by the user at his REPL, then these authors cannot depend on this feature.

3.6 Final Lesson: Explain it

While writing this article, we had to revisit many concepts and pieces of code, which led to many bug fixes and refactorings to `ASDF` and `cl-launch`. An earlier interactive "`ASDF` walkthrough" via Google Hangout also led to enhancements. Our experience illustrates the principle that you should always **explain your programs**: having to intelligibly verbalize the concepts will make *you* understand them better.

Bibliography

Daniel Barlow. ASDF Manual. 2004. `http://common-lisp.net/project/asdf/`

Zach Beane. Quicklisp. 2011. `http://quicklisp.org/`

Alastair Bridgewater. Quick-build (private communication). 2012.

François-René Rideau and Spencer Brody. XCVB: an eXtensible Component Verifier and Builder for Common Lisp. 2009. `http://common-lisp.net/projects/xcvb/`

Peter von Etter. faslpath. 2009. `https://code.google.com/p/faslpath/`

François-René Rideau and Robert Goldman. Evolving ASDF: More Cooperation, Less Coordination. 2010. `http://common-lisp.net/project/asdf/doc/ilc2010draft.pdf`

Mark Kantrowitz. Defsystem: A Portable Make Facility for Common Lisp. 1990. `ftp://ftp.cs.rochester.edu/pub/archives/lisp-standards/defsystem/pd-code/mkant/defsystem.ps.gz`

Kent Pitman. The Description of Large Systems. 1984. `http://www.nhplace.com/kent/Papers/Large-Systems.html`

François-René Rideau. ASDF3, or Why Lisp is Now an Acceptable Scripting Language (extended version). 2014. `http://fare.tunes.org/files/asdf3/asdf3-2014.html`