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

(Extended version)

François-René Rideau

Google

tunes@google.com

Abstract

ASDF, the *de facto* standard build system for Common Lisp, has been vastly improved between 2009 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 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)[1] 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 UIOP. 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; this section does not require previous knowledge of CL. 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 though some of its findings are independent from CL; for a historical perspective you may want to start with appendices A to F below before reading this section. In section 3, we discuss the challenges of evolving a piece of community software, concluding with lessons learned from our experience; these lessons are of general interest to software programmers though the specifics are related to CL.

This is the extended version of this article. In addition to extra footnotes and examples, it includes several appendices with historical information about the evolution of ASDF before ASDF 3. There again, the specifics will only interest CL programmers, but general lessons can be found that are of general interest to all software practitioners. Roughly in chronological order, we have the initial

[1] Common Lisp, often abbreviated CL, is a language defined in the ANSI standard X3.226-1994 by technical committee X3J13. It's a multi-paradigm, dynamically-typed high-level language. Though it's known for its decent support for functional programming, its support for Object-Oriented Programming is actually what remains unsurpassed still in many ways; also, few languages even attempt to match either its syntactic extensibility or its support for interactive development. It was explicitly designed to allow for high-performance implementations; some of them, depending on the application, may rival compiled C programs in terms of speed, usually far ahead of "scripting" languages and their implementations.
There are over a dozen maintained or unmaintained CL implementations. No single one is at the same time the best, shiniest, leanest, fastest, cheapest, and the one ported to the most platforms. For instance, SBCL is quite popular for its runtime speed on Intel-compatible Linux machines; but it's slower at compiling and loading, won't run on ARM, and doesn't have the best Windows support; and so depending on your constraints, you might prefer Clozure CL, ECL, CLISP or ABCL. Or you might desire the technical support or additional libraries from a proprietary implementation. While it's possible to write useful programs using only the standardized parts of the language, fully taking advantage of extant libraries that harness modern hardware and software techniques requires the use of various extensions. Happily, each implementation provides its own extensions and there exist libraries to abstract over the discrepancies between these implementations and provide portable access to threads (bordeaux-threads), Unicode support (cl-unicode), a "foreign function interface" to libraries written in C (cffi), ML-style pattern-matching (optima), etc. A software distribution system, Quicklisp, makes it easy to install hundreds of libraries that use ASDF. The new features in ASDF 3 were only the last missing pieces in this puzzle.

successful experiment in section 4; how it became robust and usable in section 5; the abyss of madness it had to bridge in section 6; improvements in expressiveness in section 7; various failures in section 8; and the bug that required rewriting it all over again in section 9.

All versions of this article are available at `http://fare.tunes.org/files/asdf3/`: extended HTML, extended PDF, short HTML, short PDF (the latter was submitted to ELS 2014).

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 that may contain files or other modules and may or may not map to subdirectories.

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(as contrasted with running it) 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 Definitions

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 section 6). The latter files each depend on the first file, because this former file defines the CL packages[2]:

```
(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.

The `fare-utils.asd` file, in its own project, looks like this (with a lot of elisions):

```
(defsystem "fare-utils" ...
  :components
  ((:file "package")
   (:module "base"
    :depends-on ("package")
    :components
    ((:file "utils")
     (:file "strings" :depends-on ("utils"))
     ...))
   (:module "filesystem"
    :depends-on ("base")
    :components
    ...)
   ...))
```

This example illustrates the use of modules: The first component is a file `package.lisp`, that all other components depend on. Then, there is a module `base`; in absence of contrary declaration, it corresponds to directory `base/`; and it itself contains files `utils.lisp`, `strings.lisp`, etc. As you can see, dependencies name *sibling* components under the same *parent* system or module, that can themselves be files or modules.

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.

Unlike its immediate predecessor `mk-defsystem`, ASDF makes a *plan* of all actions needed to obtain an up-to-date version of the build output before it *performs* these actions. In ASDF itself, this plan is a topologically sorted list of actions to be performed sequentially: a total order that is a linear extension of the partial order of dependencies; performing the actions in that order ensures that the actions are always performed after the actions they depend on.

It's of course possible to reify the complete DAG of actions rather than just extracting from it a single consistent ordered sequence. Andreas Fuchs did in 2006, in a small but quite brilliant ASDF extension called POIU, the "Parallel Operator on Independent Units". POIU compiles files in parallel on Unix multiprocessors using `fork`, while still loading them sequentially into a main

[2] Packages are namespaces that contain symbols; they need to be created before the symbols they contain may even be read as valid syntax.Each CL process has a global flat two-level namespace: symbols, named by strings, live in packages, also named by strings; symbols are read in the current `*package*`, but the package may be overridden with colon-separated prefix, as in `other-package:some-symbol`. However, this namespace isn't global across images: packages can import symbols from other packages, but a symbol keeps the name in all packages and knows its "home" package; different CL processes running different code bases may thus have a different set of packages, where symbols have different home packages; printing symbols on one system and reading them on another may fail or may lead to subtle bugs.

image, minimizing latency. We later rewrote POIU, making it both more portable and simpler by co-developing it with ASDF. Understanding the many clever tricks by which Andreas Fuchs overcame the issues with the ASDF 1 model to compute such a complete DAG led to many aha moments, instrumental when writing ASDF 3 (see section 9).

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.[3] 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.[4]

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.

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.[5] 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.[6] 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 failing to embrace the outside world and provide solutions with enough generality to solve more complex problems.[7]
- 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

[3] 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.

[4] This arguably mattered more in 2002 when ASDF was first released and was about a thousand lines long: By 2014, it has grown over ten times in size, but memory sizes have increased even faster.

[5] ASDF 3 also provides functionality which would correspond to small parts of the libc and of the linker ld.

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

[7] ASDF 3 could be easily extended to support arbitrary build actions, if there were an according desire. But ASDF 1 and 2 couldn't: their action graph was not general enough, being simplified and tailored for the common use case of compiling and loading Lisp code; and their ability to call arbitrary shell programs was a misdesigned afterthought (copied over from mk-defsystem) the implementation of which wasn't portable, with too many corner cases.

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 section 9.

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.[8] 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 May 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 (2.26.7, December 2012): not only did it provide useful new operations, but the way that ASDF 3 was automatically upgrading itself for safety purposes (see section 5.1) 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.

Note that compile-bundle-op, load-bundle-op and deliver-asd-op were respectively called fasl-op, load-fasl-op and binary-op in the original asdf-ecl and its successors up until ASDF 3.1. But those were bad names, since every individual compile-op has a fasl (a fasl, for FASt Loading, is a CL compilation output file), and since deliver-asd-op doesn't generate a binary. They were eventually renamed, with backward compatibility stubs left behind under the old name.

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 section 5.1). 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 (2.26.62), 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;[9] 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 section 5.1). In the presence of this package refactoring, this meant the development of a variant of CL's def-package 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.

[8] Most CL implementations maintain their own heap with their own garbage collector, and then are able to dump an image of the heap on disk, that can be loaded back in a new process with all the state of the former process. To build an application, you thus start a small initial image, load plenty of code, dump an image, and there you are. ECL, instead, is designed to be easily embeddable in a C program; it uses the popular C garbage collector by Hans Boehm & al., and relies on linking and initializer functions rather than on dumping. To build an application with ECL (or its variant MKCL), you thus link all the libraries and object files together, and call the proper initialization functions in the correct order. Bundle operations are important to deliver software using ECL as a library to be embedded in some C program. Also, because of the overhead of dynamic linking, loading a single object file is preferable to a lot of smaller object files.

[9] On the other hand, a special setup is now required for the debugger to locate the actual source code in ASDF; but this price is only paid by ASDF maintainers.

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 reminded of `ASDF`, it was eventually renamed `UIOP`: the Utilities for Implementation- and OS- Portability[10]. 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 section 6), 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.[11] 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`.

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

[11] `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.

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.[12]

`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

[12] Actually, our first reflex was to declare the broken `run-shell-command` deprecated, and move `run-program` to its own separate system. However, after our then co-maintainer (and now maintainer) Robert Goldman insisted that `run-shell-command` was required for backward compatibility and some similar functionality expected by various `ASDF` extensions, we decided to provide the real thing rather than this nuisance, and moved from `xcvb-driver` the nearest code there was to this real thing, that we then extended to make it more portable, robust, etc., according to the principle: **Whatever is worth doing at all is worth doing well** (Chesterfield).

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.

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*`[13] 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 based on a variable `*lisp-interaction*`: 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.

[13] 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*.

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.[14]

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:[15]

```
#!/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.[16] 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.[17] 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`[18] 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,

[14] `cl-launch` and the scripts it produces are bilingual: the very same file is accepted by both language processors. This is in contrast to self-extracting programs, where pieces written in multiple languages have to be extracted first before they may be used, which incurs a setup cost and is prone to race conditions.

[15] The Unix expert may note that despite most kernels coalescing all arguments on the first line (stripped to 128 characters or so) into a single argument, `cl-launch` detects such situations and properly restores and interprets the command-line.

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

[17] Historically, it's more accurate to say that `ASDF` imported the cache technology previously implemented by `cl-launch`, which itself generalized it from `common-lisp-controller`.

[18] Since in CL, packages are traditionally designated by symbols that are themselves traditionally upcased, case information is typically not meaningful. When converting a package name to system name or filename, we downcase the package names to follow modern convention.

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 section 9). In most cases this was undesired; indeed, ASDF 3 is predicated upon a new operation prepare-op that instead propagates *upward*.[19] 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 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 section 4). With ASDF 2, it had to be upgradable, portable, modularly configurable, robust, performant, usable (see section 5). Then it had to be more declarative, more reliable, more predictable, and capable of supporting language extensions (see section 7). 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.[20] 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

[19] 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).

[20] Even more "interesting" was a case when you'd load your foo.asd, which would define a secondary system foo-test, at the mere reference of which ASDF would try to locate a canonical definition; it would not find a foo-test.asd, instead Quicklisp might tell it to load it from its own copy of foo.asd, at which the loading of which would refer to system foo, for which ASDF would look at the canonical definition in its own foo.asd, resulting in an infinite loop. ASDF 2 was robustified against such infinite loops by memoizing the location of the canonical definition for systems being defined.

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 section 7.3).

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 section 6). 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. Note that for full support of readtable modification, other tools beside ASDF will have to be updated too: SLIME, the Emacs mode for CL, as well as its competitors for VIM, climacs, hemlock, CCL-IDE, etc.

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.

Appendices

4. Appendix A: ASDF 1, a defsystem for CL

4.1 A brief history of ASDF

In the early history of Lisp, back when core memory was expensive, all programs fit in a deck of punched cards. As computer systems grew, they became files on a tape, or, if you had serious money, on a disk. As they kept growing, programs would start to use libraries, and not be made of a single file; then you'd just write a quick script that loaded the few files your code depended on. As software kept growing, manual scripts proved unwieldy, and people started developing *build systems*. A popular one, since the late 1970s, was make.

Ever since the late 1970s, Lisp Machines had a build system called DEFSYSTEM. In the 1990s, a portable free software reimplementation, mk-defsystem, became somewhat popular. By 2001, it had grown crufty and proved hard to extend, so Daniel Barlow created his own variant, ASDF, that he published in 2002, and that became an immediate success. Dan's ASDF was an experiment in many ways, and was notably innovative in its extensible object-oriented API and its clever way of locating software. See section 4.

By 2009, Dan's ASDF 1 was used by hundreds of software systems on many CL implementations; however, its development cycle was dysfunctional, its bugs were not getting fixed, those bug fixes that existed were not getting distributed, and configuration was noticeably harder that it should have been. Dan abandoned CL development and ASDF around May 2004; ASDF was loosely maintained until Gary King stepped forward in May 2006. After Gary King resigned in November 2009, we took over his position, and produced a new version ASDF 2, released in May 2010, that turned ASDF from a successful experiment to a product, making it upgradable, portable, configurable, robust, performant and usable. See section 5.

The biggest hurdle in productizing ASDF was related to dealing with CL pathnames. We explain a few salient issues in section 6.

While maintaining ASDF 2, we implemented several new features that enabled a more declarative style in using ASDF and CL in general: declarative use of build extensions, selective control of the build, declaration of file encoding, declaration of hooks around compilation, enforcement of user-defined invariants. See section 7.

Evolving ASDF was not monotonic progress, we also had many failures along the way, from which lessons can be learned. See section 8.

After fixing all the superficial bugs in ASDF, we found there remained but a tiny bug in its core. But the bug ended up being a deep conceptual issue with its dependency model; fixing it required a complete reorganization of the code, yielding ASDF 3. See section 9.

4.2 DEFSYSTEM before ASDF

Ever since the late 1970s, Lisp implementations have each been providing their variant of the original Lisp Machine DEFSYSTEM (Moon 1981). These build systems allowed users to define *systems*, units of software development made of many *files*, themselves often grouped into *modules*; many *operations* were available to transform those systems and files, mainly to compile the files and to load them, but also to extract and print documentation, to create an archive, issue hot patches, etc.; DEFSYSTEM users could further declare dependency rules between operations on those files, modules and systems, such that files providing definitions should be compiled and loaded before files using those definitions.

Since 1990, the state of the art in free software CL build systems was mk-defsystem (Kantrowitz 1990).[21] Like late 1980s variants of DEFSYSTEM on all Lisp systems, it featured a declarative model to define a system in terms of a hierarchical tree of *components*, with each component being able to declare dependencies on other components. The many subtle rules constraining build operations could be automatically deduced from these declarations, instead of having to be manually specified by users.

However, mk-defsystem suffered from several flaws, in addition to a slightly annoying software license. These flaws were probably justified at the time it was written, several years before the CL standard was adopted, but were making it less than ideal in the world of universal personal computing. First, installing a system required editing the system definition files to configure pathnames, and/or editing some machine configuration file to define "logical pathname translations" that map to actual physical pathnames the "logical pathnames" used by those system definition files. Back when there were a few data centers each with a full time administrator, each of whom configured the system once for tens of users, this was a small issue; but when hundreds of amateur programmers were each installing software on their home computer, this situation was less than ideal. Second, extending the code was very hard: Mark Kantrowitz had to restrict his code to functionality universally available in 1990 (which didn't include CLOS), and to add a lot of boilerplate and magic to support many implementations. To add the desired features in 2001, a programmer would have had to modify the carefully crafted file, which would be a lot of work, yet eventually would probably still break the support for now obsolete implementations that couldn't be tested anymore.

4.3 ASDF 1: A Successful Experiment

In 2001, Dan Barlow, a then prolific CL hacker, dissatisfied with mk-defsystem, wrote a new defsystem variant, ASDF.[22]

[21] The variants of DEFSYSTEM available on each of the major proprietary CL implementations (Allegro, LispWorks, and formerly, Genera), seem to have been much better than mk-defsystem. But they were not portable, not mutually compatible, and not free software, and therefore mk-defsystem became *de facto* standard for free software.

[22] In a combined reverence for tradition and joke, ASDF stands for "Another System Definition Facility", as well as for consecutive letters on a QWERTY keyboard.

Thus he could abandon the strictures of supporting long obsolete implementations, and instead target modern CL implementations. In 2002, he published ASDF, made it part of SBCL, and used it for his popular CL software. It was many times smaller than mk-defsystem (under a thousand line of code, instead of five thousand), much more usable, easy to extend, trivial to port to other modern CL implementations, and had an uncontroversial MIT-style software license. It was an immediate success.

ASDF featured many brilliant innovations in its own right.

Perhaps most importantly as far as usability goes, ASDF cleverly used the *load-truename* feature of modern Lisps, whereby programs (in this case, the defsystem form) can identify from which file they are loaded. Thus, system definition files didn't need to be edited anymore, as was previously required with mk-defsystem, since pathnames of all components could now be deduced from the pathname of the system definition file itself. Furthermore, because the truename resolved Unix symlinks, you could have symlinks to all your Lisp systems in one or a handful directories that ASDF knew about, and it could trivially find all of them. Configuration was thus a matter of configuring ASDF's *central-registry* with a list of directories in which to look for system definition files, and maintaining "link farms" in those directories — and both aspects could be automated. (See section 5.3 for how ASDF 2 improved on that.)

Also, following earlier suggestions by Kent Pitman (Pitman 1984), Dan Barlow used object-oriented style to make his defsystem extensible without the need to modify the main source file.[23] Using the now standardized Common Lisp Object System (CLOS), Dan Barlow defined his defsystem in terms of *generic functions* specialized on two arguments, operation and component, using multiple dispatch, an essential OO feature unhappily not available in lesser programming languages, i.e. sadly almost all of them — they make do by using the "visitor pattern". Extending ASDF is a matter of simply defining new subclasses of operation and/or component and a handful of new methods for the existing generic functions, specialized on these new subclasses. Dan Barlow then demonstrated such simple extension with his sb-grovel, a system to automatically extract low-level details of C function and data structure definitions, so they may be used by SBCL's foreign function interface.

4.4 Limitations of ASDF 1

ASDF was a great success at the time, but after many years, it was also found to have its downsides: Dan Barlow was experimenting with new concepts, and his programming style was to write *the simplest code that would work in the common case*, giving him most leeway to experiment. His code had a lot of rough edges: while ASDF worked great on the implementation he was using for the things he was doing with it, it often failed in ugly ways when using other implementations, or exercising corner cases he had never tested. The naïve use of lists as a data structure didn't scale to large systems with thousands of files. The extensibility API while basically sane was missing many hooks, so that power users had to redefine or override ASDF internals with modified variants, which made maintenance costly.

Moreover, there was a vicious circle preventing ASDF bugs from being fixed or features from being added (Rideau 2009): Every implementation or software distribution (e.g. Debian) had its own version, with its own bug fixes and its own bugs; so developers of portable systems could not assume anything but the lowest common denominator, which was very buggy. On the other hand, because users were not relying on new features, but instead wrote kluges and workarounds that institutionalized old bugs, there was no pressure for providers to update; indeed the pressure was to not update and risk be responsible for breakage, unless and until the users would demand it. Thus, one had to assume that no bug would ever be fixed everywhere; and for reliability one had to maintain one's own copy of ASDF, and closely manage the entire build chain: start from a naked Lisp, then get one's fixed copy of ASDF compiled and loaded before any system could be loaded.[24] In the end, there was little demand for bug fixes, and supply followed by not being active fixing bugs. And so ASDF development stagnated for many years.

[23] Dan Barlow may also have gotten from Kent Pitman the idea of reifying a plan then executing it in two separate phases rather than walking the dependencies on the go.

5. Appendix B: ASDF 2, or Productizing ASDF

In November 2009, we took over ASDF maintainership and development. A first set of major changes led to ASDF 2, released in May 2010. The versions released by Dan Barlow and the maintainers who succeeded him, and numbered 1.x are thereafter referred to as ASDF 1. These changes are explained in more detail in our ILC 2010 article (Goldman 2010).

5.1 Upgradability

The first bug fix was to break the vicious circle preventing bug fixes from being relevant. We enabled hot upgrade of ASDF, so that users could always load a fixed version on top of whatever the implementation or distribution did or didn't provide. [25] Soon enough, users felt confident relying on bug fixes and new features, and all implementations started providing ASDF 2.

These days, you can (require "asdf") on pretty much any CL implementation, and start building systems using ASDF. Most implementations already provide ASDF 3. A few still lag with ASDF 2, or fail to provide ASDF; the former can be upgraded with (asdf:upgrade-asdf); all but the most obsolete ones can be fixed by an installation script we provide with ASDF 3.1.

Upgradability crucially decoupled what ASDF users could rely on from what implementations provided, enabling a virtuous circle of universal upgrades, where previously everyone was waiting for others to upgrade, in a deadlock. **Supporting divergence creates an incentive towards convergence**.

[24] It was also impossible to provide a well configured ASDF without preloading it in the image; and it was impossible to upgrade ASDF once it was loaded. Thus Debian couldn't reliably provide "ready to go" images that would work for everyone who may or may not need an updated ASDF, especially not with stability several years forward.

[25] In ASDF 3, some of the upgrade complexity described in our 2010 paper was done away with: even though **CL makes dynamic data upgrade extraordinarily easy** as compared to other languages, we found that it's not easy enough to maintain; therefore instead of trying hard to maintain that code, we "punt" and drop in-memory data if the schema has changed in incompatible ways; thus we do not try hard to provide methods for update-instance-for-redefined-class. The only potential impact of this reduction in upgrade capability would be users who upgrade code in a long-running live server; but considering how daunting that task is, properly upgrading ASDF despite reduced support might be the least of their problems. To partly compensate this issue, ASDF 3 preemptively attempts to upgrade itself at the beginning of every build (if an upgrade is available as configured) — that was recommended but not enforced by ASDF 2. This reduces the risk of either having data to drop from a previous ASDF, or much worse, being caught upgrading ASDF in mid-flight. In turn, such special upgrading of ASDF itself makes code upgrade easier. Indeed, we had found that **CL support for hot upgrade of code may exist but is anything but seamless**. These simpler upgrades allow us to simply use fmakunbound everywhere, instead of having to unintern some functions before redefinition.

5.2 Portability

A lot of work was spent on portability. Originally written for `sbcl`, `ASDF` 1 eventually supported 5 more implementations: `allegro`, `ccl`, `clisp`, `cmucl`, `ecl`. Each implementation shipped its own old version, often slightly edited; system definition semantics often varied subtly between implementations, notably regarding pathnames. `ASDF` 2.000 supported 9 implementations, adding: `abcl`, `lispworks`, `gcl`; system definition semantics was uniform across platforms. `ASDF` 2.26 (last in the `ASDF` 2 series) supported 15, adding: `cormanlisp`, `genera`, `mkcl`, `rmcl`, `scl`, `xcl`. Since then, new implementations are being released with `ASDF` support: `mocl`, and hopefully soon `clasp`.

`ASDF` as originally designed would only reliably work on Unix variants (Linux, BSD, etc., maybe cygwin, and now also MacOS X, Android, iOS). It can now deal with very different operating system families: most importantly Windows, but also the ancient MacOS 9 and Genera.

Portability was achieved by following the principle that **we must abstract away semantic discrepancies between underlying implementations**. This is in contrast with the principle apparently followed by `ASDF` 1, to "provide a transparent layer on top of the implementation, and let users deal with discrepancies". `ASDF` 2 thus started growing an abstraction layer that works around bugs in each implementation and smoothes out incompatibilities, which made the `ASDF` code itself larger, but allowed user code to be smaller for portable results.

The greatest source of portability woes was in handling *pathnames*: the standard specification of their behavior is so lacking, and the implementations so differ in their often questionable behavior, that instead of the problem being an abundance of corner cases, the problem was a dearth of common cases. So great is the disaster of CL pathnames, that they deserve their own appendix to this article (see section 6).

Lisp programmers can now "write once, run anywhere", as far as defining systems go. They still have to otherwise avoid non-standardized behavior and implementation-specific extensions if they want their programs to be portable. There are a lot of portability libraries to assist programmers, but neither `ASDF` nor any of them can solve these issues intrinsic to CL, short of portably implementing a new language on top of it.

Portability decoupled which implementation and operating system were used to develop a system from which it could be compiled with, where previously any non-trivial use of pathnames, filesystem access or subprocess invocation was a portability minefield.

5.3 Configurability

`ASDF` 1 was much improved over what preceded it, but its configuration mechanism was still lacking: there was no *modular* way for whoever installed software systems to register them in a way that users could see them; and there was no way for program writers to deliver executable scripts that could run without knowing where libraries were installed.

One key feature introduced with `ASDF` 2 (Goldman 2010) was a new configuration mechanism for programs to find libraries, the *source-registry*, that followed this guiding principle: **Each can specify what they know, none need specify what they don't**.

Configuration information is taken from multiple sources, with the former partially or completely overriding the latter: argument explicitly passed to `initialize-source-registry`, environment variable, central user configuration file, modular user configuration directory, central system configuration files, modular system configuration directories, implementation configuration, with sensible defaults. Also, the source-registry is optionally capable of recursing through subdirectories (excluding source control directories), where `*central-registry*` itself couldn't. Software management programs at either user or system level could thus update independent configuration files in a modular way to declare where the installed software was located; users could manually edit a file describing where they manually downloaded software; users could export environment variables to customize or override the default configuration with context-dependent information; and scripts could completely control the process and build software in a predictable, deterministic way; it's always possible to take advantage of a well-configured system, and always possible to avoid and inhibit any misconfiguration that was out of one's control.

A similar mechanism, the *output-translations*, also allows users to segregate the output files away from the source code; by default it uses a cache under the user's home directory, keyed by implementation, version, OS, ABI, etc. Thus, whoever or whichever software manages installation of source code doesn't have to also know which compiler is to be used by which user at what time. The configuration remains modular, and code can be shared by all who trust it, without affecting those who don't. The idea was almost as old as ASDF itself, but previous implementations all had configurability issues.[26]

Configurability decoupled use and installation of software: multiple parties could now each modularly contribute some software, whether applications, libraries or implementations, and provide configuration for it without being required to know configuration of other software; previously, whoever installed software couldn't notify users, and users had to know and specify configuration of all software installed.

[26] Debian's `common-lisp-controller` (CLC) popularized the technique as far back as 2002, and so did the more widely portable `asdf-binary-locations` (ABL) after it: by defining an `:around` method for the `output-files` function, it was possible for the user to divert where `ASDF` would store its output. The fact that this technique could be developed as an obvious extension to `ASDF` without the author explicitly designing the idea into it, and without having to modify the source code, is an illustration of how expressive and modular CLOS can be.
But apart from its suffering from the same lack of modularity as the `*central-registry*`, CLC and ABL also had a chicken-and-egg problem: you couldn't use `ASDF` to load it, or it would itself be compiled and loaded without the output being properly diverted, negating any advantage in avoiding clashes for files of other systems.
ABL thus required special purpose loading and configuration in whichever file did load `ASDF`, making it not modular at all. CLC tried to solve the issue by managing installation or all CL software; it failed eventually, because these efforts were only available to CL programmers using Debian or a few select other Linux software distributions, and only for the small number of slowly updated libraries, making the maintainers a bottleneck in a narrow distribution process. CLC also attempted to institute a system-wide cache of compiled objects, but this was ultimately abandoned for security issues; a complete solution would have required a robust and portable build service, which was much more work than justified by said narrow distribution process.
The solution in `ASDF` 2 was to merge this functionality in `ASDF` itself, according to the principle to **make it as simple as possible, but no simpler**. But whereas `ASDF` 1 followed this principle under the constraint that the simple case should be handled correctly, `ASDF` 2 updated the constraint to include handling all cases correctly. Dan Barlow's weaker constraint may have been great for experimenting, it was not a good one for a robust product.
Another, more successful take on the idea of CLC, is Zach Beane's Quicklisp (2011): it manages the loading and configuration of `ASDF`, and can then download and install libraries. Because it does everything in the user's directory, without attempts to share between users and without relying on support from the system or software distribution, it can be actually ubiquitous. Thanks to `ASDF` 2's modular configuration, the Quicklisp-managed libraries can complement the user's otherwise configured software rather than one completely overriding the other.

5.4 Robustness

ASDF 1 used to pay little attention to robustness. A glaring issue, for instance, was causing much aggravation in large projects: killing the build process while a file was being compiled would result in a corrupt output file that would poison further builds until it was manually removed: ASDF would fail the first time, then when restarted a second time, would silently load the partially compiled file, leading the developer to believe the build had succeeded when it hadn't, and then to debug an incomplete system. The problem could be even more aggravating, since a bug in the program itself could be causing a fatal error during compilation (especially since in CL, developers can run arbitrary code during compilation). The developer, after restarting compilation, might not see the issue; he would then commit a change that others had to track down and painfully debug. This was fixed by having ASDF compile into a temporary file, and move the outputs to their destination only in case of success, atomically where supported by implementation and OS. A lot of corner cases similarly had to be handled to make the build system robust.

We eventually acquired the discipline to **systematically write tests for new features and fixed bugs**. The test system itself was vastly improved to make it easier to reproduce failures and debug them, and to handle a wider variety of test cases. Furthermore, we adopted the policy that the code was not to be released unless every regression test passed on every supported implementation (the list of which steadily grew), or was marked as a known failure due to some implementation bugs. Unlike ASDF 1, that focused on getting the common case working, and letting users sort out non-portable uncommon cases with their implementation, ASDF 2 followed the principle that code should either work of fail everywhere the same way, and in the latter case, **fail early for everyone rather than pass as working for some** and fail mysteriously for others. These two policies led to very robust code, at least compared to previous CL build systems including ASDF 1.

Robustness decoupled the testing of systems that use ASDF from testing of ASDF itself: assuming the ASDF test suite is complete enough, (sadly, all too often a preposterous assumption), systems defined using ASDF 2 idioms run the same way in a great variety of contexts: on different implementations and operating systems, using various combinations of features, after some kind of hot software upgrade, etc. As for the code in the system itself — it might still require testing on all supported implementations in case it doesn't strictly adhere to a portable subset of CL (which isn't automatically enforceable so far), since the semantics of CL are not fully specified but leave a lot of leeway to implementers, unlike e.g. ML or Java.

5.5 Performance

ASDF 1 performance didn't scale well to large systems: Dan Barlow was using the `list` data structure everywhere, leading to worst case planning time no less than $O(n^4)$ where n is the total size of systems built. We assume he did it for the sake of coding simplicity while experimenting, and that his minimalism was skewed by the presence of many builtin CL functions supporting this old school programming style. ASDF did scale reasonably well to a large number of small systems, because it was using a hash-table to find systems; on the other hand, there was a dependency propagation bug in this case (see section 9). In any case, ASDF 2 followed the principle that **good data structures and algorithms matter**, and should be tailored to the target problem; it supplemented or replaced the lists used by ASDF 1 with hash-tables for name lookup and append-trees to recursively accumulate actions, and achieved linear time complexity. ASDF 2 therefore performed well whether or not the code was split in a large number of systems.

Sound performance decoupled the expertise in writing systems from the expertise in how systems are implemented. Now, developers could organize or reorganize their code without having to shape it in a particular way to suit the specific choice of internals by ASDF itself.

5.6 Usability

Usability was an important concern while developing ASDF 2. While all the previously discussed aspects of software contribute to usability, some changes were specifically introduced to improve the user experience.

As a trivial instance, the basic ASDF invocation was the clumsy `(asdf:operate 'asdf:load-op :foo)` or `(asdf:oos 'asdf:load-op :foo)`. With ASDF 2, that would be the more obvious `(asdf:load-system :foo)`[27].

ASDF 2 provided a portable way to specify pathnames by adopting Unix pathname syntax as an abstraction, while using standard CL semantics underneath. It became easy to specify hierarchical relative pathnames, where previously doing it portably was extremely tricky. ASDF 2 similarly provided sensible rules for pathname types and type overrides. (See section 6.) ASDF made it hard to get pathname specifications right portably; ASDF 2 **made it hard to get it wrong** or make it non-portable.

Usability decoupled the knowledge of how to use ASDF from both the knowledge of ASDF internals and CL pathname idiosyncrasies. Any beginner with a basic understanding of CL and Unix pathnames could now use ASDF to portably define non-trivial systems, a task previously reserved to experts and/or involving copy-pasting magical incantations. The principle followed was that **the cognitive load on each kind of user must be minimized**.

6. Appendix C: Pathnames

6.1 Abstracting over Pathnames

The CL standard specifies a `pathname` class that is to be used when accessing the filesystem. In many ways, this provides a convenient abstraction, that smoothes out the discrepancies between implementations and between operating systems. In other ways, the standard vastly underspecifies the behavior in corner cases, which makes pathnames quite hard to use. The sizable number of issues dealt by ASDF over the years were related to pathnames and either how they don't match either the underlying operating system's notion, or how their semantics vary between implementations. We also found many outright bugs in the pathname support of several implementations, many but not all of which have been fixed after being reported to the respective vendors.

To make ASDF portable, we started writing abstraction functions during the end year of ASDF 1, they grew during the years of ASDF 2, and exploded during the heavy portability testing that took place before ASDF 3. The result is now part of the UIOP library transcluded in ASDF 3. However, it isn't possible to fully recover the functionality of the underlying operating system in a portable way from the API that CL provides. This API is thus what Henry Baker calls an *abstraction inversion* (Baker 1992) or, in common parlance, *putting the cart before the horse*.

UIOP thus provides its functionality on a best-effort basis, within the constraint that it builds on top of what the implementation provides. While the result is robust enough to deal with the

[27] `load-system` was actually implemented by Gary King, the last maintainer of ASDF 1, in June 2009; but users couldn't casually *rely* on it being there until ASDF 2 in 2010 made it possible to rely on it being ubiquitous. Starting with ASDF 3.1, `(asdf:make :foo)` is also available, meaning "do whatever makes sense for this component"; it defaults to `load-system`, but authors of systems not meant to be loaded can customize it to mean different things.

kind of files used while developing software, that makes UIOP not suitable for dealing with all the corner cases that may arise when processing files for end-users, especially not in adversarial situations. For a complete and well-designed reimplementation of pathnames, that accesses the operating system primitives and libraries via CFFI, exposes them and abstracts over, in a way portable across implementations (and, to the extent that it's meaningful, across operating systems), see IOLib instead. Because of its constraint of being self-contained and minimal, however, ASDF cannot afford to use IOLib (or any library).

All the functions in (UIOP) have documentation strings explaining their intended contract.

6.2 Pathname Structure

In CL, a pathname is an object with the following components:

- A *host* component, which is often but not always nil on Unix implementations, or sometimes :unspecific; it can also be a string representing a registered "logical pathname host" (see section 6.7); and in some implementations, it can be a string or a structured object representing the name of a machine, or elements of a URL, etc.
- A *device* component, which is often but not always nil on Unix implementations, can be a string representing a device, like "c" or "C" for the infamous C: on Windows, and can be other things on other implementations.
- A *directory* component, which can be nil (or on some implementations :unspecific), or a list of either :absolute or :relative followed by "words" that can be strings that each name a subdirectory, or :wild (wildcard matching any subdirectory) or :wild-inferiors (wildcard matching any recursive subdirectory), or some string or structure representing wildcards. e.g. (:absolute "usr" "bin"), or (:relative ".svn"). For compatibility with ancient operating systems without hierarchical filesystems, the directory component on some implementations can be just a string, which is then an :absolute.
- A *name* component, which is often a string, can be nil (or on some implementations :unspecific), or :wild or some structure with wildcards, though the wildcards can be represented as strings.
- A *type* component, which is often a string, can be nil (or on some implementations :unspecific), or :wild or some structure with wildcards, though the wildcards can be represented as strings.
- A *version* component, which can be nil (or on some implementations :unspecific), and for backward compatibility with old filesystems that supported this kind of file versions, it can be a positive integer, or :newest. Some implementations may try to emulate this feature on a regular Unix filesystem.

Previous versions of SCL tried to define additional components to a pathname, so as to natively support URLs as pathnames, but this ended up causing extra compatibility issues, and recent versions fold all the additional information into the *host* component.

:unspecific is only fully or partly supported on some implementations, and differs from nil in that when you use merge-pathnames, nil means "use the component from the defaults" whereas :unspecific overrides it.

Already, UIOP has to deal with normalizing away the ancient form of the directory component and bridging over ancient implementations that don't support pathnames well with function make-pathname*, or comparing pathnames despite de-normalization with function pathname-equal.

6.3 Namestrings

The CL standard accepts a strings as a pathname designator in most functions where pathname is wanted. The string is then parsed into a pathname using function parse-namestring. A pathname can be mapped back to a string using namestring. Due to the many features regarding pathnames, not all strings validly parse to a pathname, and not all pathnames have a valid representation as a namestring. However, the two functions are reasonably expected to be inverse of each other when restricted to their respective output domains.[28]

These functions are completely implementation-dependent, and indeed, two implementations on the same operating system may do things differently. Moreover, parsing is relative to the host of a default pathname (itself defaulting to *default-pathname-defaults*), and even on Unix, this host can be a "logical pathname", in which case the syntax is completely different from the usual syntax.

Behavior that vary notably include: escaping of wildcard characters (if done at all) or other characters (dots, colons, backslash, etc.) handling of filenames that start with a dot ".", catching of forbidden characters (such as a slash "/" in a *name* component, of a dot "." in a type component), etc.

Notably because of wildcard characters, CL namestrings do not always match the namestrings typically used by the operating system and/or by other programming languages, or indeed by other CL implementations on the same operating system. Also, due to character encoding issues, size limits, special character handling, and depending on Lisp implementations and underlying operating system, it's possible that not all byte sequences that the OS may use to represent filenames may have a corresponding namestring on the CL side, while not all well-formed CL namestrings have a corresponding OS namestring. For instance, Linux accepts any sequence of null-terminated bytes as a namestring, whereas Apple's MacOS tends to only accept sequence of Unicode characters in UTF-8 Normalization Form D and what Windows wants varies with the API used. And so UIOP provides parse-native-namestring and native-namestring to map between pathname objects and strings more directly usable by the underlying OS. On good implementations, these notably do a better job than the vanilla CL functions at handling wildcard characters. Unhappily, on other implementations they don't do anything. CL is notably missing a portable way to escape a namestring to avoid wildcards.

Finally, so that .asd files may portably designate pathnames of recursive subdirectories and files under a build hierarchy, we implemented our own parsing infrastructure. Thus, even if the current pathname host of *default-pathname-defaults* isn't a Unix host, indeed even if the operating system isn't Unix but Windows, MacOS 9 or Genera, the same .asd file keeps working unmodified, and correctly refers to the proper filenames under the current source directory. In UIOP, the main parsing function is parse-unix-namestring. It's designed to match the expectations of a Unix developer rather than follow the CL API.

For instance, you can specify an optional type suffix in a way that is added to the provided stem rather than "merged" in the style of make-pathname or merge-pathnames; thus (parse-unix-namestring "foo-V1.2" :type "lisp") returns

[28] Still, on many implementations, filesystem access functions such as probe-file or truename will return a subtly modified pathname that looks the same because it has the same namestring, but doesn't compare as equal because it has a different version of :version :newest rather than :version nil. This can cause a lot of confusion and pain. You therefore should be careful to never compare pathnames or use them as keys in a hash-table without having normalized them, at which point you may as well use the namestring as the key.

`#p"foo-V1.2.lisp"`. This contrasts with the idioms previously used by ASDF 1 in similar situations, which for the longest time would return `#p"foo-V1.lisp"`. The type may also be specified as `:directory`, in which case it treats the last "word" in the slash-separated path as a directory component rather than a name and type components; thus, `(parse-unix-namestring "foo-V1.2" :type :directory)` returns `#p"foo-V1.2/"`, at least on a Unix filesystem where the slash is the directory separator.

Each function in UIOP tries to do the Right Thing™, in its limited context, when passed a string argument where a pathname is expected. Often, this Right Thing consists in calling the standard CL function `parse-namestring`; sometimes, it's calling `parse-unix-namestring`; rarely, it's calling `parse-native-namestring`. And sometimes, that depends on a `:namestring` argument, as interpreted by UIOP's general pathname coercing function `ensure-pathname`. To be sure, you should read the documentation and/or the source code carefully.

6.4 Trailing Slash

Because CL pathnames distinguishes between directory, name and type components, it must distinguish `#p"/tmp/foo.bar"` from `#p"/tmp/foo.bar/"`. Assuming that `*default-pathname-defaults*` is a physical Unix pathname, the former is about the same as `(make-pathname :directory '(:absolute "tmp") :name "foo" :type "bar")` and denotes a file, whereas the latter is about the same as `(make-pathname :directory '(:absolute "tmp" "foo.bar") :name nil :type nil)` and denotes a directory. They are very different things. Yet to people not familiar with CL pathname, it's a common issue to not differentiate between the two, then get bitten by the system. UIOP implements predicates `file-pathname-p` and `directory-pathname-p` to distinguish between the two kinds of pathnames (and there are yet more kinds of pathname objects, because of wildcards). It also provides a function `ensure-directory-pathname` to try to add the equivalent trailing slash after the fact when it was previously omitted.

ASDF 1 notably located systems thanks to a `*central-registry*`, a list of expressions that `eval`uate to directories, in which to look for `.asd` files. That was a great innovation of ASDF compared to its predecessors, that made it much easier to configure, and ASDF 2 and ASDF 3 still support this feature. But it was also somewhat quite fragile and easy to misconfigure, the most recurrent issue being that users familiar with Unix but not CL pathnames would insert the name of a directory without a trailing slash `"/"`. CL would parse that as the name of a file in the parent directory, which, when later merged (see section 6.5) with the name of a prospective `.asd` file, would behave as if it were the parent directory, leading to confusion, defeat and a lot of frustration before the issue is identified.

Gary King fixed that issue in the late days of ASDF 1 in 2009. But users couldn't rely on the fix being present everywhere, since the availability of ASDF upgrades was limited. Since ASDF 2, you can.

6.5 Merging Pathnames

CL has an idiom `(merge-pathnames pathname defaults)` that ostensibly serves to merge a relative pathname with an absolute pathname. However, using it properly isn't trivial at all.

First, CL makes no guarantee that `nil` is a valid pathname *host* component. Indeed, since the pathname syntax may itself vary depending on hosts, there might be no such thing as a pathname with "no host"; same thing with the *device* component. Now, `merge-pathnames` takes the host from the first argument, if not `nil` (and `nil` isn't allowed on many implementation); if the intent was supposed to designate a pathname relative to the second argument, which might be on a host unknown at the time the first argument was made or parsed, this will be totally the wrong thing: care must be taken to pass to `merge-pathnames` a first argument that has the correct host and device components, as taken from the second argument — unless the first argument is itself an absolute pathname that should *not* default these components.

To resolve these issues, as well as various other corner cases and implementation bugs, and to make things fully portable and support logical pathnames correctly, we ended up completely reimplementing our own variant of `merge-pathnames` that indeed considers that the host and device components of a relative pathname are not to be merged, and called it `merge-pathnames*` where `*` is a common suffix traditionally used in CL to denote variants of a function.

Also, to make it easier to portably specify in one go the merging of a relative pathname under a given main directory, UIOP also provides `(subpathname main relative)`. See the UIOP documentation for additional options to this function, and yet more functions.

6.6 `nil` as a Pathname

The functions defined by the CL standard do not recognize `nil` as valid input where a pathname is required. At the same time, they provide no way to explicit specify "no pathname information available". This is compounded by the fact that most functions implicitly call `merge-pathnames` with the default `*default-pathname-defaults*` (that thus cannot be `nil`). Yet, when the resulting pathname is relative, it may result in the Unix operating system implicitly merging it with the result of `getcwd`, according to its own rules different from those of CL.

Some people may be tempted to try use `#p""` or `(make-pathname :host nil :device nil :directory nil :name nil :type nil :version nil)` as a neutral pathname, but even though either or both might work on many implementations most of the time, the former fails catastrophically if parsed in a context when the `*default-pathname-defaults*` is a logical pathname or otherwise non-standard, and the latter isn't guaranteed to be valid, since some implementations may reject `nil` as a host or device component, or may merge them implicitly from the defaults. UIOP tries hard to provide function `nil-pathname` and variable `*nil-pathname*`, yet on some implementations they are not neutral for the standard CL `merge-pathnames`, only for UIOP's `merge-pathnames*`.

Most pathname-handling functions in UIOP tend to accept `nil` as valid input. `nil` is a neutral element for `merge-pathnames*`. Other functions that receive `nil` where pathname information is required tend to return `nil` instead of signaling an error. For instance, `subpathname` if its first argument is `nil` returns the second argument parsed but unmerged, whereas its variant `subpathname*` in this case returns `nil`. When in doubt, you should read the documentation and/or the source code.

As an aside, one issue that `*central-registry*` had during the days of ASDF 1 was that computed entries (as evaluated by `eval`) had to always return a pathname object, even when the computation resulted in not finding anything useful; this could lead to subtle unwanted complications. Gary King also fixed that in the late days of ASDF 1, by making ASDF accept `nil` as a result and then ignore the entry.

6.7 Logical Pathnames

A logical pathname is a way to specify a pathname under a virtual "logical host" that can be configured independently from the *physical pathname* where the file is actually stored on a machine.

Before it may be used, a logical pathname host must be registered, with code such as follows:

```
(setf (logical-pathname-translations
       "SOME-HOST")
  '(("SOURCE;**;*.LISP.*"
       "/home/john/src/**/*.lisp.*")
    ("SOURCE;**;*.ASD.*"
       "/home/john/src/**/*.asd.*")
    ("**;*.FASL.*"
       "/home/john/.fasl-cache/**/*.fasl.*")
    ("**;*.TEMP.*"
       "/tmp/**/*.tmp.*")
    ("**;*.*.*"
       "/home/john/data/**/*.*.*")))
```

The first two lines map Lisp source files and system definitions under the absolute directory source to a subdirectory in John's home; The third line maps fasl files to a cache; the fourth maps files with a temporary suffix to `/tmp`; and the fifth one maps all the rest to a data directory. Thus, the case-insensitive pathname `#p"some-host:source;foo;bar.lisp"` (internally stored in uppercase) would be an absolute logical pathname that is mapped to the absolute physical pathname `/home/john/src/foo/bar.lisp` on that machine; on different machines, it might be be configured differently, and for instance on a Windows machine might be mapped to `C:\Users\jane\Source\foo\bar.lsp`.

Problem is, this interface is only suitable for power users: it requires special setup before anything is compiled that uses them, typically either by the programmer or by a system administrator, in an implementation-dependent initialization file (and the `#p"..."` syntax is implementation-dependent, too). Moreover, once a string is registered as a logical pathname host, it may shadow any other potential use that string might have in representing an actual host according to some implementation-dependent scheme. Such a setup is therefore not modular, and not robust: as an author, to be sure you're not interfering with any other piece of software, you'd need to avoid all the useful hostnames on all Lisp installations on the planet; more likely, as a system administrator, you'd need to audit and edit each and every piece of Lisp software to rename any logical pathname host that would clash with a useful machine name. All of this made sense in the 1970s, but already no more in the mid 1990s, and not at all in the 2010s. "Logical pathnames" are totally inappropriate for distributing programs as source code "scripts" to end users. Even programmers who are not beginners will have trouble with "logical pathnames".

Importantly, the standard specifies that only a small subset of characters is portably accepted: uppercase letters, digits, and hyphens. When parsed, letters of a logical pathname are converted to uppercase; once mapped to physical pathnames, the uppercase letters are typically converted to whatever is conventional on the destination pathname host, which these days is typically lowercase, unlike in the old days. Logical pathnames also use the semi-colon `";"` as directory separator, and, in a convention opposite to that of Unix, a leading separator indicates a `:relative` pathname directory whereas absence thereof indicates an `:absolute` pathname directory. This makes the printing of standard logical pathnames look quite unusual and the distraction generated is a minor nuisance.

Most implementations actually accept a preserved mix of lowercase and uppercase letters without mapping them all to uppercase. On the one hand, that makes these logical pathnames more useful to users; on the other hand, this doesn't conform to the standard. One implementation, SBCL, strictly implements the standard, in the hope of helping programmers not accidentally write non-conformant programs, but, actually makes it harder to portably use logical pathnames, especially since it's the only implementation doing this (that I know of).

Finally, because the specification involves matching patterns in a sequence until a first match is found, which is inefficient, and because the feature isn't popular for all the above reasons, implementations are unlikely to be fast at translating logical pathnames, and especially not for large lists of translations; even if optimized translation tables were made, there is no incremental interface to modifying such tables. Logical pathnames thus intrinsically do not scale.

Until we added more complete tests, we found that logical pathname support tended to bitrot quickly. Adding tests revealed a lot of discrepancies and bugs in implementations, that entailed a lot of painful work. For instance, subtle changes in how we search for `.asd` files may have caused logical pathnames being translated to physical pathnames earlier than users might expect. The use of `truename`, implicitly called by `directory` or `probe-file`, would also translate away the logical pathname as well as symlinks. Some implementations, probably for sanity, translate the logical pathname to a physical pathname before they bind `*load-pathname*`: indeed code that includes literal `#p"pathname/objects"` when read while the `*default-pathname-defaults*` is "logical", or when such namestrings are merged with the `*load-pathname*`, may fail in mysterious ways (including `ASDF` itself at some point during our development).

Many implementations also had notable bugs in some corner cases, that we discovered as we added more tests that `ASDF` worked well with logical-pathnames; this suggests that logical pathnames are not a widely used feature. Indeed, we estimate that only a handful or two of old school programmers worldwide may be using this feature still. Yet, despite our better sense, we sunk vast amounts of time into making `ASDF` support them[29] for the sake of this sacred backward compatibility and the pride of hushing the criticism by this handful of ungrateful old school programmers who still use them. The days of work poured into getting logical pathnames to work were probably not well spent.

In any case, this all means that nowadays, logical pathnames are *always* a bad idea, and *we strongly recommend against using these ill "logical" pathnames*. They are a hazard to end users. They are a portability nightmare. They can't reliably name arbitrary files in arbitrary systems. In locating source code, it's vastly inferior to the `ASDF` 2 `source-registry`. As a programmer interface, it's inferior to `ASDF`'s `asdf:system-relative-pathname` function that uses a system's base directory as the first argument to `UIOP`'s `subpathname` function.

6.8 Portability Done Right

Because of all the parsing issues above, trying to specify relative pathnames in `ASDF` 1 was very hard to do in a portable way. Some would attempt to include a slash `"/"` in places that `ASDF` passed as a name component to `make-pathname`, notably the name of an `ASDF component` (confusingly, a completely different concept despite the same name). This would work on some lax implementations on Unix but would fail outright on stricter implementations and/or outside Unix (remarkably, SBCL, that is usually stricter than other implementations, counts as lax in this case). Some would try to use `#.(merge-pathnames ...)` to construct pathnames at read-time, but few would understand the complexity of pathname merging well enough to do it just right, and the results would be highly non-portable, with corner cases galore.

[29] We always believe it's a small bug that will be fixed in the next half-hour. After hours of analysis and false tracks, we finally understand the issue for good, and just do it... until we find the next issue, and so on.

ASDF 2 solved the entire situation by standardizing on its own portable Unix-like syntax for pathnames. Only then, could the same specification be used on all supported platforms with the same semantics. Once again, portability was achieved by systematically *abstracting away semantic discrepancies between underlying implementations*.

For instance, the common case of a simple component definition `(:file "foo")`, with no special character of any kind, only letters, digits and hyphens, was always portably treated by ASDF as meaning a file named `"foo.lisp"` in the current system's or module's directory. But nothing else was both portable and easy.

If you wanted your file to have a dot in its name, that was particularly tricky. In Dan Barlow's original ASDF, that used a simple `make-pathname`, you could just specify the name with the dot as in `(:file "foo.bar")`, which would yield file `#p"foo.bar.lisp"`, but trying that on a class that didn't specify a type such as `(:static-file "READ.ME")` would yield an error on SBCL and other implementations, because it isn't acceptable to have a dot in the name yet no type you should instead separate out a type component — unless it's a single dot is at the beginning, indicating a Unix hidden file. In latter variants of ASDF 1, the above static-file example would work, with an elaborate system to extract the type; however, the former example would now fail subtly, with the type `"bar"` ending up overridden by the type `"lisp"` from the class. I eventually fixed the issue in the build system at work by overriding key functions of ASDF, and got the fix accepted upstream shortly before I became maintainer.[30]

If you wanted your file to have a different type, that was also quite hard. You could specify an explicit `:pathname #p"foo.l"` option to each and every component, redundantly specifying the file name in an error-prone way, or you had to define a new component class using a cumbersome protocol (see section 8.8). But you couldn't specify a file with no type when the class specified one.

The situation was much worse when dealing with subdirectories. You could naïvely insert a slash `"/"` in your component name, and ASDF would put it in the *name* component of the pathname, which would happen to work on SBCL (that would be lax about it, contrary to its usual style), but would be illegal on many other implementations, and was not generally expected to work on a non-Unix operating system. You could try to provide an explicit pathname option to your component definition as in `(:file "foo/bar" :pathname #p"foo/bar.lisp")`, but in addition to being painfully redundant, it would still not be portable to non-Unix operating systems (or to a "logical pathname" setup). A portable solution involved using `merge-pathnames` inside a reader-evaluation idiom `#.(merge-pathnames ...)`, which in addition to being particularly verbose and ugly, was actually quite tricky to get right (see section 6.5). As for trying to go back a level in the filesystem hierarchy, it was even harder: `#p"../"` was at least as unportable as the pathname literal syntax in general, and to use `merge-pathnames` you'd need to not use `".."` as a directory component word, but instead use `:back`, except on implementations that only supported `:up`, or worse.

Finally, there was no portable way to specify the current directory: none of `""`, `"."`, `"./"`, `#p""`, `#p"."`, `#p"./"` led to portable outcome, and they could all mean something completely different and usually wrong in "logical pathname" context. Thus, if you were in a module `"this-module"` and wanted to define a submodule `"that-submodule"` that doesn't define a subdirectory but shares that of `"this-module"` with ASDF 2 you can portably specify `:pathname ""`, but back in the days of ASDF 1, if you wanted to be completely portable, you had to specify the following, though you could define a variable rather than repeat the computation:

```
:pathname
#.(make-pathname
    :name nil :type nil :version nil
    :defaults *load-truename*)
```

If you wanted to make it into a different directory, with ASDF 2 or later you could use `"../foo/bar"` but with ASDF 1 the portable way to do it was:

```
:pathname
#.(merge-pathnames
   (make-pathname
     :name nil :type nil :version nil
     :directory '(:relative :back "foo" "bar")
     :defaults *load-truename*)
   *load-truename*)
```

Except that `:back` isn't completely portable to all implementations, and you might have to use the subtly different `:up` instead (if supported).

Starting with ASDF 2, things became much simpler: Users specify names that are uniformly parsed according to Unix syntax on all platforms. Each component name (or explicit pathname override, if given as a string), is combined with the specified file type information and correctly parsed into relative subdirectory, name and type pathname components of a relative pathname object that relative pathname is merged into whichever directory is being considered, that it is relative to. Under the hood, the proper combinations of `make-pathname` and `merge-pathnames` are used, taking into account any quirks and bugs of the underlying implementation, in a way that works well with either logical pathnames or non-Unix physical pathnames. A type, if provided by the component or its class and not `nil`, is always added at the end of the provided name. If the type is `nil`, nothing is added and the type is extracted from the component name, if applicable. You could always explicitly override the class-provided type with `:type "l"` or `:type nil`. No surprise, no loss of information, no complex workarounds.

7. Appendix D: ASDF 2.26, more declarative

7.1 `defsystem` Dependencies

ASDF 2 introduced a `:defsystem-depends-on` option to `defsystem`, whereby a system could declaratively specify dependencies on build extensions. Previously, users would imperatively load any extension they need: their `.asd` system definition file would include `(asdf:load-system "cffi-grovel")` before the extension may be used by `defsystem`. Indeed, a `.asd` file is just a CL source file that is loaded in a controlled context and may contain arbitrary side-effects; now such side-effects are frowned upon and a declarative style is more maintainable, hence this improvement.

However, this feature was only made usable in 2.016 (June 2011), when ASDF started to accept keywords as designators for classes defined in an extension in the `:asdf` package. Before then, there was a chicken-and-egg problem: the `defsystem` form containing the `:defsystem-depends-on` declaration was read before the extension was loaded (what's more, ASDF 1 and 2 read it in a temporary package); therefore, the extension had nowhere to intern or export any symbol that the rest of the `defsystem` form could use.

These days, this feature is the recommended way of loading extensions. But from the story of it, we can learn that **a feature**

[30] It is remarkable that CL makes it possible to patch code without having to modify source files. This makes it possible to the use unmodified source code of libraries or compilers with uncooperative or unavailable authors, yet get bugs fixed. It is also important for temporary fixes in "release" branches of your code. Maintenance costs are of course much reduced if you can get your fix accepted by the upstream software maintainer.

isn't finished until it's tested and used in production. Until then, there are likely issues that need to be addressed.

As an example use, the proper way to use the CFFI library is to use `:defsystem-depends-on ("cffi-grovel")` as below, which defines the class `asdf::cffi-grovel`, that can be designated by the keyword `:cffi-grovel` amongst the components of the system:

```
(defsystem "some-system-using-ffi"
  :defsystem-depends-on ("cffi-grovel")
  :depends-on ("cffi")
  :components
  ((:cffi-grovel "foreign-functions")
   ...))
```

7.2 Selective System Forcing

Since the beginning, ASDF has had a mechanism to force recompilation of everything:

```
(asdf:oos 'asdf:load-op 'my-system :force t)
```

In ASDF 2 that would be more colloquially:

```
(asdf:load-system 'my-system :force :all)
```

In 2003, Dan Barlow introduced a mechanism to *selectively* `:force` recompilation of some systems, but not others: `:force :all` would force recompilation of all systems; `:force t` would only force recompilation of the requested system; and `:force '(some list of systems)` would only force recompilation of the specified systems. However, his implementation had two bugs: `:force t` would continue to force everything, like `:force :all`; and `:force '(some list of systems)` would cause a runtime error (that could have been found at compile-time with static strong typing).

The bugs were found in 2010 while working on ASDF 2; they were partially fixed, but support for the selective syntax was guarded by a continuable error message inviting users to contact the maintainer.[31] Despite the feature demonstrably not ever having had any single user, it had been partially documented, and so was finally fixed and enabled in ASDF 2.015 (May 2011) rather than removed.

The feature was then extended in ASDF 2.21 (April 2012) to cover a negative `force-not` feature, allowing the fulfillment of a user feature request: a variant `require-system` of `load-system` that makes no attempt to upgrade already loaded systems. This is useful in some situations: e.g. where large systems already loaded and compiled in a previously dumped image are known to work, and need to promptly load user-specified extensions, yet do not want to expensively scan every time the (configured subset of the) filesystem for updated (or worse, outdated) variants of their source code. The hook into the `require` mechanism was then amended to use it.[32]

This illustrates both Dan Barlow's foresight and his relative lack of interest in developing ASDF beyond the point where it got the rest of his software off the ground; and by contrast the obsession to detail of his successor.

[31] CL possesses a mechanism for continuable errors, `cerror`, whereby users can interactively or programmatically tell the system to continue despite the error.

[32] The two mechanisms were further enhanced in ASDF 3, then in ASDF 3.1. One conceptual bug was having the `:force` mechanism take precedence over `:force-not`; this didn't fit the common use cases of users having a set of immutable systems that shouldn't be refreshed at all, and needing to stop a `:force :all` from recursing into them. This was only fixed in ASDF 3.1.

7.3 Encoding Support

Back in 2002, most programmers were still using 8-bit characters in various encodings (latin1, koi8-r, etc.), and Emacs did not support Unicode very well. ASDF 1 in its typical minimalist manner, just didn't specify any `:external-format` and let the programmer deal with the implementation-dependent configuration of character encodings, if such an issue mattered to them.

By 2012, however, Unicode was ubiquitous, UTF-8 was a *de facto* standard, and Emacs supported it well. A few authors had started to rely on it (if only for their own names). Out of over 700 systems in Quicklisp, most were using plain ASCII, but 87 were implicitly using `:utf-8`, and 20 were using some other encoding, mostly latin1.

Now, one would sometimes attempt loading a latin1 encoded file in a Lisp expecting strictly UTF-8 input, resulting in an error, or loading a UTF-8 or Shift-JIS encoded file in a Lisp expecting latin1, resulting in mojibake. The SBCL implementation was notable for simultaneously (and legitimately) (1) setting the default encoding in a given session from the same environment variables as the `libc` locale, which could vary wildly between developers, even more so hypothetical end-users, and (2) issuing an error rather than accept invalid UTF-8. Unhappily, the person who chose the encoding was whoever wrote the code, and had no control on what environment was used at compile-time; whereas the user, who may or may not be aware of such encoding issues, had no idea what encoding an author used, and didn't care until an error was raised from an unknown library that was depended on by a program he used or wrote.

To make the loading of library code more predictable, ASDF 2 added an `:encoding` option to `defsystem`, so that files may be loaded in the encoding they were written in, as determined by the author, irrespective of which encoding the user may otherwise be using. Once again, the principle *each can specify what they know, none need specify what they don't.*

The encoding option of a system or module is inherited by its components, if not overridden. The accepted syntax of the option is a keyword, abstracting over the implementation-dependent `:external-format`, which isn't specified by the CL standard.[33] The only encoding supported out of the box is `:utf-8`, because that's the only universally accepted encoding that's useful; but if your system specifies `:defsystem-depends-on ("asdf-encodings")`, it can use any encoding your implementation supports. However, the only other completely portable option is `:latin1`, the previous implicit standard being evolved from. On old implementations without support for Unicode or external-formats, ASDF falls back to using the 8-bit implementation `:default`.

Though `:utf-8` was already the *de facto* standard, the default was initially left to `:default` for backward-compatibility, to give time to adapt to the twenty or so authors of systems that were using incompatible encodings, half of which fixed their code within a few days or weeks, and half of which never did. This default changed to `:utf-8` one year later, with the pre-release of ASDF 3, under the theory that **it's good practice to release all small backward-incompatible changes together with a big one**, since that's the time users have to pay attention, anyway. Though it did break the few remaining unmaintained systems, the new defaults actually made things more reliable for a hundred or so other systems, as witnessed by the automated testing tool `cl-test-grid`.

Because we had learned that a feature isn't complete until it's tested, we published a system that demonstrates how to put this

[33] And indeed, though all other implementations that support Unicode accept the keyword `:utf-8` as an external format, GNU CLISP, always the outlier, wants the symbol `charset:utf-8` in a special package `charset`.

new infrastructure to good use: `lambda-reader`, a utility that lets you use the Unicode character λ instead of `lambda` in your code.[34] Originally based on code by Brian Mastenbrook, `lambda-reader` was modified to fall back gracefully to working mojibake where Unicode isn't supported, and to offer the syntax modification via the *de facto* standard `named-readtables` extension. Users still have to enable the modified syntax at the beginning of every file, and carefully disable it at the end, least they cause havoc in other files or at the REPL (see section 3.5).

7.4 Hooks around Compilation

A recurrent question to `ASDF` developers was about how to properly modify the CL syntax for some files, without breaking the syntax for other files: locally giving short nicknames to packages, changing the readtable, or the reader function, etc.

The original answer was to define a new subclass `my-cl-file` of `cl-source-file`, then a method on `perform :around ((o compile-op) (c my-cl-file))`, wrapping the usual methods inside a context with modified syntax. However, not only was it a cumbersome interface, the seldom used operation `load-source-op` also redundantly needed the same method yet was often forgotten, and so did any future such imaginable operation involving reading the file.

A better, more declarative interface was desirable, and implemented in `ASDF` 2.018 (October 2011): each component can specify an `:around-compile` option or inherit it from its parent; if not `nil`, this designates a function to be called around compilation (but not loading, to preserve the semantics of bundle operations). An explicit `nil` is often needed in the first few files of a system, before the usual function was defined.

Actually, the function usually cannot be named by a symbol, because at the time the `.asd` file is read, none of the code has been compiled, and the package in which the symbol will be interned doesn't exist yet; therefore, `ASDF` 2.019 (November 2011) made it possible to designate a function by a string that will be `read` later. Hence, for instance, systems defined in Stelian Ionescu's IOLib,[35] use `:around-compile "iolib/asdf:compile-wrapper"`, except for the system `iolib/asdf` itself, that defines the package and the function.

7.5 Enforcing User-Defined Invariants

Relatedly `ASDF` 2.23 (July 2012) added the ability for users to define invariants that are enforced when compiling their code. Indeed, a file might be compliant CL code, and compile correctly, yet fail to satisfy application-specific invariants essential to the correct behavior of the application. Without the build system checking after every file's compilation, users would be left with an invalid system; after they eventually get a runtime error, they would have to chase which of thousands of files broke the invariant. Thanks to the `:compile-check` feature, the `:around-compile` hook can tell `ASDF` to check the invariant before to accept some compilation output that would otherwise poison future builds (see section 5.4 above about poisoned builds).

There were two notable use cases at ITA Software. In the simpler one, the error logging infrastructure was registering at compile-time all strings that could be seen by end-users, to build a database that could be localized to another language, as per legal requirements of the customer. But it was not enough to register strings at compile-time, because unless you were building everything from scratch in the same process, the compile-time state was lost before the final build image was dumped. And it was not possible to register them as part of the macro's expansion, because this expansion was not for code to be evaluated at the toplevel, but only for code called conditionally, in exceptional situations. One solution would have been to side-effect external files; a better solution was for the macro to defer registration to a cleanup form, evaluated at the toplevel before the end of the file's compilation. Since there is no standard mechanism to achieve this effect, this required users to explicitly include a `(final-forms)` at the end of their file. Now, users are prone to forgetting to include such a statement, when they are aware at all that they need to. But thanks to the new `:compile-check` feature, the system could automatically check the invariant that no deferred form should be left dangling without a `final-forms`, and reject the file with a helpful error message instructing the programmer to insert said form. `asdf-finalizers`, a separately distributed `ASDF` extension, provides such an infrastructure: its `eval-at-toplevel` both evaluates a form and defers it for later inclusion at the top-level, and its `final-forms` includes all registered such forms at the top-level; user code can then specify in their `defsystem` the `:around-compile "asdf-finalizers:check-finalizers-around-compile"` hook for `ASDF` to enforce the invariant.

The other use case was similarly solved with `asdf-finalizers`. Our data schema included hundreds of parametric types such as `(list-of passenger)` of `(ascii-string 3 5)` (for strings of ASCII characters length between 3 and 5). Checking that data verified the proper invariants to avoid inserting corrupted data records in the database or messaging them to partners was an essential robustness feature. But to define the type via the CL `deftype` mechanism, these types had to expand to things like `(and list (satisfies list-of-passenger-p))`, where the predicate function `list-of-passenger-p` could not be provided additional parameters, and had to be independently defined by a form `(declare-list-of passenger)`; there again, this form could not be part of the type expansion, and was not enough to `eval`uate at compile-time, for it had to be explicitly included at the top-level. Manually managing those forms was a maintenance burden, and `asdf-finalizers` eliminated this burden.

The principle we recognized was that **every large enough application is a Domain-Specific Language with its own invariants**, and the programming language is but the implementation language of the DSL. This implementation is extremely fragile if it cannot automatically enforce the invariants of the DSL. A good programming language lets you define new invariants, and a good build system enforces them. In CL, thanks to `ASDF`, this can all happen without leaving the language.

8. Appendix E: Failed Attempts at Improvement

8.1 Failed Experiments

The road from `mk-defsystem` to `ASDF` 3 is undeniably one of overall improvements. Yet, along the way, many innovations were attempted that didn't pan out in the end.

For instance, Gary King, when newly made maintainer of `ASDF` 1, attempted to define some concept of preference files, so that users may customize how the build takes place, or fix some systems without modifying their source code. The feature was never used, and Gary King eventually removed it altogether. Maybe the lack of a reliable shared version of `ASDF`, combined with the relative paucity of hooks in `ASDF` 1, made the proposition unattractive and more pain to maintain that it helped. Also, the unconditional loading of preferences was interfering with the reproducible build

[34] Yes, it does feel good to write λ this way, and it does improve code that uses higher-order functions.

[35] IOLib is a comprehensive general purpose I/O library for CL, written by Stelian Ionescu, that strives at doing the Right Thing™ where many other libraries sacrifice code quality, feature coverage or portability for the sake of expediency.

of software, and some users complained, notably the authors of SBCL itself. On the other hand, sometimes things are broken, and you do need a non-intrusive way of fixing them. Thus ASDF will probably grow at some point some way to configure fixes to builds without patching code, but it isn't there yet.

Later versions of ASDF 1 also introduced their own generalized asdf:around method combination, that wrapped around the traditional :around method combination, so it may define some methods without blocking users from defining their own extensions. This was causing portability issues with implementations that didn't fully implement this corner of CLOS. ASDF 2 removed this feature, instead dividing in two the function perform that was using it, with the method around it being explicitly called perform-with-restarts. Indeed, in a cross-compilation environment, you'd want your restarts in the master Lisp, whereas the perform method takes place on the target compiler, so it really makes sense. ASDF 1 authors liked to experiment with how far they could push the use of CLOS; but at some point there can be too much fanciness.

As another smaller example of this experimental mindset, Dan Barlow made a lot of uses of anaphoric macros as then popularized by Paul Graham: ASDF notably made copious use of aif, a variant of if that implicitly (and "non-hygienically") binds a variable it to the condition expression in its success branch. But the experiment was eventually considered a failure, and the rough community consensus of the CL community is that anaphoric macros are in poor taste, and so in ASDF 3, all remaining occurrences of aif where replaced by an explicitly binding macro if-let copied from the alexandria library.

An experiment during the ASDF 2 days was to introduce a variable *compile-file-function* so that ECL could override ASDF's compile-file* to introduce behavioral variations between its C generating compiler and its bytecode generating compiler. This proved to be a hard to maintain attractive nuisance, that only introduced new failure modes (particularly during upgrade) and required either duplication of code with compile-file* or ugly refactorings, without bringing any actual meaningful user extensibility. The real solution was to make ASDF's compile-file* itself more clever and aware of the peculiarities of ECL. *Know the difference between the need for extensibility and the need for correctness*; if there's only one correct behavior, what you need isn't extensibility, it's correctness.

8.2 Partial Solutions

The asdf-binary-locations extension ultimately failed because it didn't fully solve its configuration problem, only concentrated it in a single point of failure. The *system-cache* feature to share build outputs between users and associated get-uid function, introduced by common-lisp-controller and used by ASDF 2's output-translation layer, were removed because of security issues. See section 5.3. A :current-directory keyword in the configuration DSL was removed, because not only did its meaning vary wildly with implementation and operating system, this meaning varied with what the value of that global state at the time the configuration file was read, yet because of lazy loading and implicit or explicit reloading of configuration, no one was really in control of that value. On the other hand, the :here keyword was a successful replacement: it refers to the directory of the configuration file being read, the contents of which are clearly controlled by whoever writes that file.

In an attempt to solve namespace clashes between .asd files, Dan Barlow had each of them loaded in its own automatically created private package asdf0, asdf1, etc., automatically deleted afterward. But this didn't help. If the file contained no new definition, this hassle wasn't needed; and if there were new definitions, either users were using the same kind of prefixing conventions as were necessary anyway to avoid clashes in existing packages, or they were defining their own package foo-system, to hold the definitions. Otherwise, when the definitions were left in the default package, their symbol became unreachable and the definitions impossible to debug. In the end, to solve the namespace issues of CL would have required a complete intrusive change of the package system, and that was not a task for ASDF. If anything, faslpath, quick-build and asdf/package-inferred-system seem to have a better approach at enforcing namespace discipline.

These failures were all partial solutions, that solved an issue in the common case (output redirection, namespace hygiene), while leaving it all too problematic in a concentrate case that didn't make the overall issue ultimately easier to solve. More like hiding the dirt under the carpet than vacuuming it away. The eventual solutions required confronting the issues head on.

8.3 Attempted Namespace Grabs

In the last days of ASDF 1, there was an attempt to export its small set of general purpose utilities as package asdf-extensions, quickly renamed asdf-utilities before the release of ASDF 2, to avoid a misnomer. Still, because ASDF had been changing so much in the past, and it was hard to rely on a recent version, no one wanted to depend on ASDF for utilities, especially not when the gain was so small in the number of functions used. A brief attempt was make these (now more numerous) utilities available as a completely separate system asdf-utils with its own copy of them in its own package. But the duplication felt like both a waste of both runtime resources and maintainer time. Instead, asdf-driver, once renamed UIOP, was relatively successful, because it was also available as a system that could be updated independently from the rest of ASDF, yet shared the same source code and same package as the version used by ASDF itself. No duplication involved. That's a case where two namespaces for the same thing was defeating the purpose, and one namespace necessitated the two things to actually be the same, which could not be the case until this transclusion made it possible.

In a different failure mode, a brief attempt to give asdf-driver the nickname d was quickly met with reprobation, as many programmers feel that that short a name should be available for a programmer's own local nicknames while developing. Trying to homestead the :DBG keyword for a debugging macro met the same opposition. Some (parts of) namespaces are in the commons and not up for grabs.

8.4 Not Successful Yet

Some features were not actively rejected, but haven't found their users yet.

ASDF 3 introduced build-op as a putative default build operation that isn't specialized for compiling CL software. But it hasn't found its users yet. The associated function asdf:build-system was renamed asdf:make in ASDF 3.1 in an effort to make it more usable. Maybe we should add an alias asdf:aload for asdf:load-system, too.

During the ASDF 2 days, the *load-system-operation* was designed so that ECL may use load-bundle-op instead of load-op by default; but that's still not the case, and won't be until ECL users more actively test it, which they might not do until it's the default, since they haven't otherwise heard of it. Indeed, it seems there are still bugs in corner cases.

8.5 Ubiquity or Bust!

CL possesses a standard but underspecified mechanism for extending the language: (require "module") loads given "mod-

ule", as somehow provided by the implementation, if not present yet. Dan Barlow hooked ASDF into SBCL's require mechanism ASDF 2 eventually did likewise for ABCL, GNU CLISP, Clozure CL, CMUCL, ECL, MKCL as well as SBCL. — the list coincides with that of all maintained free software implementations. Thus, on all these implementations, users could, after they (require "asdf"), implicitly rely on ASDF to provide systems that are not yet loaded.

However, users ended up mostly not using it, we presume for the following reasons:

- This mechanism is still not ubiquitous enough, therefore for portability and reliability, you have to know about ASDF and be able to fall back to it explicitly, anyway; thus trying to "optimize" the easy case with require is just gratuitous cognitive load for no gain. There again, an underspecified standard ended being counter-productive.
- The require mechanism purposefully avoids loading a module that has already been provided, thereby making it unpopular in a culture of ubiquitous modifiable source code; if you modified a file, you really want it to be reloaded automatically.[36]

8.6 Interface Rigidity

There were many cases during ASDF development where we wanted to rename a function or change the behavior of a class. Often, we could do it, but sometimes, we found we couldn't: when a generic function was simultaneously called by users and extended by users; or when a class was simultaneously used as a base class to inherit from and as a mixin class to get behavior from.

For instance, we found that component-depends-on was a complete misnomer, and should have been action-depends-on or something similar. But since there were user systems that defined methods on this function, our action-depends-on would have had to call component-depends-on at least as a fallback. Conversely, because some users do call component-depends-on, that function would have to call action-depends-on. To avoid infinite recursion would then require complex machinery that could prove error-prone, for little gain beside a name change. The rename was not to happen.

Similarly, we wanted to remove some behavior from the abstract class operation, but found that some users relied on that behavior, so we couldn't remove it, yet our software relied on that behavior being removed, so we had to remove it. In the end, we implemented an ugly mechanism of "negative inheritance", to selectively disable the behavior for appropriate subclasses of operation while keeping it for legacy operations (see section 2.11).

The perform function also has this general problem: the right thing would be for users to keep defining methods on perform, but to never call it directly, instead calling perform-with-restarts, which allows more modular extensibility.

By contrast, the CLOS protocol was cleverly designed so that users do not usually call the functions on which they define methods (such as initialize-instance, or update-instance-for-redefined-class), and do not usually define methods on the functions they call.

Do not impose overly rigid interfaces on yourself.

8.7 Cognitive Load Matters

While developing ASDF, we sometimes made many things more uniform at the cost of a slight backward incompatibility with a few existing systems using kluges. For instance, ASDF 2 made pathname arguments uniformly non-evaluated in a defsystem form, when they used to be evaluated for toplevel systems but not for other (most) components; this evaluation was used by a few users to use merge-pathnames to portably specify relative pathnames, a task made unnecessary by ASDF 2 being capable of specifying these pathnames portably with Unix syntax.

ASDF 3 also removed the magic undocumented capability that a system could specify a dependency on another system foo by having (:system "foo") in its list of children components, rather than "foo" in its *depends-on* option. One system relied on it, which had been ported from mk-defsystem where this a valid documented way of doing things. In ASDF 1 and 2, it seems this happened to work by accident rather than design, and this accident had been eliminated in the ASDF 3 refactorings.

At the cost of a few users having to cleanup their code a bit, we could thus notably **reduce the cognitive load on users** for all future systems. No more need to learn complex syntactic and semantic constraints and even more complex tricks to evade those constraints.

8.8 Verbosity Smells Bad

Back in the bad old days of ASDF 1, the official recipe, described in the manual, to override the default pathname type .lisp for a Lisp source file to e.g. .cl, used to be to define a method on the generic function source-file-type, specialized on the class cl-source-file and on your system (in this example, called my-sys):

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

Some people advertised this alternative, that also used to work, to define your own sub-class foo-file of cl-source-file, and use: (defmethod source-file-type ((c foo-file) (s module)) "foo"). This caused much grief when we tried to make system not a subclass of module anymore, but both be subclasses of new abstract class parent-component instead.

In ASDF 2.015, two new subclasses of cl-source-file were introduced, cl-source-file.cl and cl-source-file.lsp, that provide the respective types .cl and .lsp, which covers the majority of systems that don't use .lisp. Users need simply add to their defsystem the option :default-component-class :cl-source-file.cl and files will have the specified type. Individual modules or files can be overridden, too, either by changing their class from :file to :cl-source-file.cl, or more directly by specifying a :pathname parameter.

If needed, users can define their own subclass of cl-source-file and override its default type, as in:

```
(defclass my-source-file (cl-source-file)
  ((type :initform "l")))
```

Or they can directly override the type while defining a component, as in:

```
(:file "foo" :type "l")
```

In any case, the protocol was roundabout both for users and implementers, and a new protocol was invented that is both simpler

[36] At the same time, ASDF wasn't reliable in avoiding to reload provided modules, since most systems don't call provide with their name to signal that such call to require was successful, and therefore next call to require would cause a new load attempt — this was fixed with the introduction of the above-mentioned require-system in ASDF 2.21 in 2012, and its use instead of load-system. Maybe the more general point is that ASDF did not have a good story with regards to extending the set of things that are considered "system" versus "user" defined. ASDF 3.1 adds a notion of "immutable systems" that should not be refreshed from filesystem once loaded into memory.

to use and easier to extend. **Verbosity is a bad smell, it suggests lack of abstraction, or bad abstractions**.

8.9 Underspecified Features

While discussing pathnames in section 6, we mentioned how a lot of the issues were related to the CL standard leaving the semantics of pathnames underspecified.

We experienced a bit of the same trouble with ASDF itself. For the sake of extensibility, Dan Barlow added to ASDF 1 a catch-all "component-properties" feature: system authors could specify for each component (including systems) an association list of arbitrary key-value pairs, with `:properties ((key1 . value1) (key2 . value2))`[37]. The idea was that extensions could then make use of this data without having to explicitly define storage for it. The problem was, there was no way to associate shared meaning to these properties to any key-value pair across systems defined by multiple authors. Amongst the tens of authors that were using the feature in Quicklisp, no two agreed on the meaning of any key. Sometimes, general-purpose metadata was made available under different keys (e.g. `#:author-email` vs `("system" "author" "email")`. Most of the time, the data was meant to be processed by a specific extension from the same author.

When we released ASDF 3, we declared the feature as deprecated: we defined a few new slots in class `system` to hold useful common metadata found previously in such properties: `homepage`, `mailto`, `bug-tracker`, `long-name`, `source-control`. Otherwise, we recommended that system authors should specify the `:defsystem-depends-on` and `:class` options, so that their systems could use regular object-oriented programming to define extended classes with well-defined semantics. What if a list of key-value pairs (aka alist or association-list) is exactly what a programmer wants? He should define and use a subclass of `system` to hold this alist, and then be confident that his keys won't clash with anyone else's. Unhappily, for the sake of backward-compatibility, we couldn't actually remove the `:properties` feature yet; we also refrained from neutering it; we just marked it as deprecated for now.

The `:properties` interface created a commons, that was mismanaged. The `:class` interface instead establishes semantic ownership of extended data elements, and opens a market for good system extensions.

8.10 Problems with CL itself

Besides the issues with standardization, another general problem with CL is that its semantics are defined in terms of *irreversible side-effects to a global environment*. A better principle would be to **define a programming language's semantics in terms of pure transformations with local environments**.

There are many lessons to be learned by studying the successes and failures of the Lisp community. The CL language and community are probably too rigid to apply these lessons; but maybe your current or next programming language can.

9. Appendix F: A `traverse` across the build

9.1 The End of ASDF 2

While the article itself describes the *features* introduced by the various versions of ASDF, this appendix focuses on the *bugs* that were the death of ASDF, and its rebirth as ASDF 3.

The ASDF 2 series culminated with ASDF 2.26 in October 2012, after a few months during which there were only minor cleanups, portability tweaks, or fixes to remote corner cases. Only one small bug remained in the bug tracker, with maybe two other minor annoyances; all of them were bugs as old as ASDF itself, related to the `traverse` algorithm that walks the dependency DAG.

The minor annoyances were that a change in the `.asd` system definition file ought to trigger recompilation in case dependencies changed in a significant way, and that the `traverse` algorithm inherited from ASDF 1 was messy and could use refactoring to allow finer and more modular programmatic control of what to build or not to build. The real but small bug was that dependencies were not propagated across systems. Considering that my co-maintainer Robert Goldman had fixed the same bug earlier in the case of dependencies across modules within a system, and that one reason he had disabled the fix across systems was that some people claimed they enjoyed the behavior, it looked like the trivial issue of just enabling the obvious fix despite the conservative protests of some old users. It was a wafer thin mint of an issue.

And so, of course, since this was the "last" bug standing, and longstanding, I opened it... except it was a Pandora's Box of bigger issues, where the fixing of one quickly led to another, etc., which resulted in the explosion of ASDF 2.

9.2 The `traverse` Algorithm

In the last release by Dan Barlow, ASDF 1.85 in May 2004, the `traverse` algorithm was a 77-line function with few comments, a terse piece of magic at the heart of the original 1101-line build system.[38] Shortly before I inherited the code, in ASDF 1.369 in October 2009, it had grown to 120 lines, with no new comment but with some commented out debugging statements. By the time of ASDF 2.26 in October 2012, many changes had been made, for correctness (fixing the incorrect handling of many corner cases), for robustness (adding graceful error handling), for performance (enhancing asymptotic behavior from $O(n^4)$ to $O(n)$ by using better data structures than naïve lists), for extensibility (moving away support for extra features such as `:version` and `:feature`), for portability (a trivial tweak to support old Symbolics Lisp Machines!), for maintainability (splitting it into multiple smaller functions and commenting everything). There were now 8 functions spanning 215 lines. Yet the heart of the algorithm remained essentially unchanged, in what was now a heavily commented 86-line function `do-traverse`. Actually, it was one of a very few parts of the ASDF 1 code base that we hadn't completely rewritten.

Indeed, no one really understood the underlying design, why the code worked when it did (usually) and why it sometimes didn't. The original author was long gone and not available to answer questions, and it wasn't clear that he fully understood the answers himself — Dan Barlow had been experimenting, and how successfully! His ASDF illustrates the truth that **code is discovery at least as much as design**; he had tried many things, and while many failed, he struck gold once or twice, and that's achievement enough for anyone.

Nevertheless, the way `traverse` recursed into children components was particularly ugly; it involved an unexplained special kind of dependency, *do-first*, and propagation of a *force* flag. But of course, any obvious attempt to simplify these things caused the algorithm to break somehow.

Here is a description of ASDF 1's `traverse` algorithm, reusing the vocabulary introduced in section 1.1.3.

`traverse` recursively visits all the nodes in the DAG of actions, marking those that are visited, and detecting circularities. Each action consists of an operation on a component; for a simple CL system with regular Lisp files, these actions are `compile-`

[37] An experienced Lisp programmer will note that calling it properties then making it an alist rather than a plist was already bad form.

[38] A git checkout of the code has a `make` target `extract` that will extract notable versions of the code, so you can easily look at them and compare them.

op for compiling the code in the component, and load-op for loading the compiled code; a component is a system, a recursive module, or a file (actually a cl-source-file).

When visiting the action of an operation on a component, traverse propagates the operations along the component hierarchy, first sideway amongst siblings, then specially downwards toward children: if A *depends-on* B (in the component DAG), then any operation on A *depends-on* same operation on B (this being a dependency in the distinct action DAG); then, any operation on A *depends-on* same operation on each of A's children (if any). Thus, to complete the load-op (resp. compile-op) of a module, you must first complete the load-op (resp. compile-op) of all the components it was declared as *depends-on*, then on all its own children. Additionally, a load-op on A *depends-on* a compile-op on A; this is actually encoded in the extensible function component-depends-on:[39] user-defined operation classes can be defined, with according new methods for the component-depends-on function.

Now here comes the tricky part. The action of a compile-op on A has a special *do-first* dependency on a load-op of each of A's sideway dependencies. New *do-first* dependencies can otherwise be specified in the defsystem form, though no one does it and there is no extensible component-do-first function. These dependencies are included in the plan not only before the action, but also before any of the operations on the component's children; yet they are not visited to determine whether the action needs to be performed, and so the children are specially visited after the siblings but before the *do-first*, yet the *do-first* are inserted before the children. And this careful sequencing is baked into the traverse algorithm rather than reified in dependencies of the action graph.

What if you transform these *do-first* dependencies into regular *in-order-to* dependencies? Then there is no incremental compilation anymore, for the first time you attempt to load-op a system, any file that has dependencies would have a compile-op action that *depends-on* the load-op actions on its dependencies, that obviously haven't been completed yet; and so any file with dependencies would be recompiled every time.

9.3 *Force* Propagation

Now, as it traversed the action graph, ASDF was propagating a *force* flag indicating whether an action needed to be performed again in the current session due to some of its dependencies itself needing to be updated.

The original bug was that this flag was not propagated properly. If some of the sideway dependencies were outdated, then all children needed to be forced; but ASDF 1 failed to do so. For instance, if module A *depends-on* B, and B is flagged for (re)compilation, then all the children of A need to be flagged, too. And so Robert Goldman had fixed this bug in the lead-up to the ASDF 2 release, by correctly propagating the flag; except for many reasons, he had declined at the time to propagate it for systems, propagating it only for modules inside systems. Glancing at the bug three years later, I naïvely figured it was just a matter of removing this limitation (2.26.8). Except that fix didn't work reliably between systems, and that was why he hadn't just done it.

If system A *depends-on* system B, both were once compiled, B was subsequently modified and separately recompiled, and you'd ask ASDF to compile A again, then it would not flag B for recompilation, and therefore not flag A. Indeed, each compiled file in A looked up to date, when comparing it to the corresponding source file, as ASDF did; since no *force* flag from B was issued, ASDF would think it was done. Bug.

For modules within a system, the problem mostly did not arise, because the granularity of an operate request was a system, and so there was no way to request compilation of B without triggering compilation of A. For the bug to be visible within a system, it took an external build interruption such as a machine crash or power loss, or an angry programmer killing the process because it is hosed; in case of such obvious event, programmers would somehow learn to rebuild from clean if experiencing some seeming filesystem corruption. On the other hand, across systems, the problem arose quite naturally: working on a system B, compiling it and debugging it, then working on a client system A, was not only possible but the *usual* workflow.

Seeing no way to fix the bug reliably, Robert had disabled propagation of the flag between systems, which at least was predictable behavior. The usual workaround was for programmers to force recompilation of A using :force t; due to another bug (see section 7.2), this was actually recompiling everything, thus eschewing any other such issue in the current session. The problem, when diagnosed, was easily solved in wetware. Except of course it wasn't always easy to diagnose, resulting in hours wasted trying to debug changes that didn't happen, or worse, to committing bugs one was not seeing to a shared repository, and having other programmers try to figure out why their code stopped working after they updated their checkout.

9.4 *Timestamp* Propagation

ASDF should have been propagating timestamps, not just force flags for whether recompilation was needed in the current session! So we painfully rewrote the existing algorithm to support timestamps rather than a flag (2.26.9, 2.26.10).

As for *do-first* dependencies such as loading a file, we would stamp a load-op not with the time at which the file was loaded, but with the timestamp of the file being loaded. As a side benefit, this wholly eliminated the previous need for kluges to avoid clock skew between the processor clock and the fileserver clock (though not clock skew between multiple file servers used during the build).

This wasn't enough, though. To wholly get rid of *do-first*, we had to distinguish between actions that were done in the current image, versus actions that weren't done in the current image, but that might still be up-to-date, because their effects were all in the filesystem. (This distinction since ASDF 1 had been present in the function operation-done-p that checked for timestamps without propagation when there were both input and output files, and had to come up with an answer when there weren't.) Therefore, when examining an action, we must separate the propagated timestamp from a non-propagated flag telling whether the action needs to be done in the current image or not. The generic function compute-action-stamp looks at the dependencies of an action, its inputs and outputs on disk, and possible stamps from it being done earlier in the current image, and returns a stamp for it and a flag for whether it needs to be done (or redone) in the current image. Thus, if a compiled file is up-to-date on disk and an up-to-date version was loaded, the compute-action-stamp function returns its timestamp and t (true); if the file is up-to-date on disk but either it wasn't loaded yet or an outdated version was loaded, the compute-action-stamp function returns its timestamp and nil (false); if the file is missing or out-of-date on disk, then no up-to-date version could be loaded yet, and compute-action-stamp returns an infinity marker and nil. The infinity marker (implemented as boolean t) is so that no timestamp is up-to-date in comparison, and corresponds to the force flag of ASDF 1. A neg-

[39] This function is quite ill-named, since it describes dependencies between *actions*, not between components. But the original ASDF 1 code and documentation doesn't include an explicit notion of action, except to mention "visited nodes" in comments about traverse. The notion was made explicit while implementing ASDF 3, reusing the word from an earlier technical report by Robbins (Robbins 1985).

ative infinity marker (implemented as boolean `nil`) also serves to mark as no dependency.[40] (Of course, *do-first* would come back with a vengeance, see below section 9.6).

Then, we started to adapt `POIU` to use timestamps. `POIU` is an `ASDF` extension, originally written by Andreas Fuchs, but or which we had inherited the maintenance, and that computes a complete action graph of the build to compile in parallel (see section 1.1.3). However, our attempt to run the modified `POIU` would fail, and we'd be left wondering why, until we realized that was because we had previously deleted what looked like an unjustified kluge: `POIU`, in addition to the dependencies propagated by `ASDF`, was also having each node in the action graph depend on the dependencies of each of its transitive parents. Indeed, the loading of dependencies (both *in-order-to* and *do-first*) of a component's parent (and transitively, ancestors), were all implicitly depended upon by each action. In an undocumented stroke of genius, Andreas Fuchs had been making explicit in the DAG the implicit sequencing done by traverse! However, these parent dependencies were being passed around inefficiently and inelegantly in a list, updated using `append` for a quadratic worst time cost. This cost wouldn't explode as long as there were few systems and modules; but removing the magic sequencing of `traverse` to replace it with a different and inefficient kluge didn't seem appealing, especially after having optimized `traverse` into being of linear complexity only.

And the solution was of course to explicitly reify those implicit dependencies in the action graph, making it a complete explicit model.

9.5 Prepare Operation

And so we introduced a new operation, initially called `parent-load-op` (2.26.14), but eventually renamed `prepare-op` (2.26.21), corresponding to the steps required to be taken in preparation for a `load-op` or `compile-op`, namely to have completed a `load-op` on all the sideway dependencies of all the transitive parents.

Now, unlike `load-op` and `compile-op` that both were propagated *downward* along the dependency graph, from parents to children, `prepare-op` had to be propagated *upward*, from children to parents. And so, the `operation` class had a new special subclass `upward-operation`, to be specially treated by `traverse`...

Or better, the propagation could be moved entirely out of `traverse` and delegated to methods on `component-depends-on`! A mixin class `downward-operation` would handle the downward propagation along the component hierarchy for `load-op`, `compile-op` and the likes, whereas `upward-operation` would handle `prepare-op`; `sideway-operation` would handle the dependency from `prepare-op` to the `load-op` of a component's declared *depends-on*, whereas `selfward-operation` would handle the dependency of `load-op` and `compile-op` to `prepare-op`. Thanks to CLOS multiple inheritance and double dispatch, it all fell into place (2.26.21).

For instance, the good old downward propagation was implemented by this mixin:

```
(defclass downward-operation (operation) ())
(defmethod component-depends-on
     ((o downward-operation)
      (c parent-component))
  `((,o ,@(component-children c))
    ,@(call-next-method)))
```

The current version is more complex, with all of nine (full-length) lines of code plus comments and doctrings, for additional backward compatibility and extensibility, but this gives the gist: The action of a downward operation on a parent component *depends-on* the same operation `,o` on each of the component's children, followed by other dependencies from other aspects of the action. Had backward-compatibility not been required, the function would have been called `action-depends-on`, and its `method-combination` would have been `append`, so that it wouldn't be necessary to write that `,@(call-next-method)` in each and every method definition. But backward-compatibility was required.

In any case, classes like `load-op` and `compile-op` just inherit from this mixin, and voilà, no need for any magic in `traverse`, which at that point had been broken down in neat small functions, none more than fifteen lines long. If anything, some complexity had been moved to the function `compute-action-stamp` that computes timestamps and deals with corner cases of missing inputs or missing outputs, which was 48 heavily commented lines of code (67 as of 3.1.2), just slightly more than the misdesigned function `operation-done-p` it was superseding.

Now everything was much cleaner. But of course, it was a mistake to call it a victory yet, for *do-first* came back to enact revenge for my killing it; and once again, Andreas Fuchs had prophesized the event and provided a weapon to successfully defend against the undead.

9.6 Needed In Image

Former *do-first* dependencies of an action used to not partake in the forcing, but were nevertheless to be done before the action. Reminder: in practice, they were the loading of dependencies before compiling a file. With the new, saner, action graph, they were now regular dependencies; the only difference was that they don't contribute anything to the action stamp (and thus to forcing) that wasn't already contributed by the action creating the file they loaded. Still, they had to be done, in order, in the current image.

Now, this last constraint was utterly defeating the purpose of some bundle operations, where the whole point of using a bundle fasl was to not have to load the individual fasls (see section 2.2). In the old `ASDF` 1 model, the `load-bundle-op` *depends-on* `compile-bundle-op`[41] which *depends-on* a lot of individual `compile-op`, which only *do-first* the `load-op` of their dependencies. Therefore, if the individual files look up to date, no individual loading takes place. Except of course `ASDF` 1 will fail to detect that files are out of date when the system's dependencies have changed. In the new `ASDF` 3 model, the fact that the `compile-op` actions are out of date is detected thanks to recursing through their `prepare-op` and `load-op` dependencies; but with the naïve approach to building a plan that always load dependencies, this causes all those individual `load-op` to be issued.

The solution was again suggested by `POIU`. For the sake of determining whether an action could be performed in parallel in a fork, or had to be done in the image of the main process, `POIU` had introduced a predicate `needed-in-image-p`. The notion

[40] Interestingly, this `compute-action-stamp` could be very easily updated to use cryptographic digests of the various files instead of timestamps, or any other kind of stamp. Because it's the only function for which the contents of stamps isn't opaque, and is a generic function that takes a plan class as parameter, it might be possible to override this function either for a new plan class and make that the `*default-plan-class*`), without destructively modifying any code. However, this hasn't been tested, so there's probably a bug lurking somewhere. Of course, such a modification cannot be part of the standard `ASDF` core, because it has to be minimal and ubiquitous and can't afford to pull a cryptographic library (for now), but an extension to `ASDF`, particularly one that tries to bring determinism and scalability, could use this very simple change to upgrade from timestamps to using a persistent object cache addressed by digest of inputs.

[41] They were then respectively named `load-fasl-op` and `fasl-op`, but have since be renamed.

was actually suggested by the old method `operation-done-p` from Dan Barlow's original `ASDF` 1: If an action has any `output-files`, then `ASDF` considers that the operation is worth it for its output, and should have no meaningful or reliable side-effects in the current image; it thus counts as *not* `needed-in-image-p`. If on the other hand, an action has no `output-files`, then `ASDF` considers that the operation is worth its side-effects in the current image; it thus counts as `needed-in-image-p`.

What the new `traverse-action` action had to do (2.26.46), was to associate to each visited node a status depending on whether or not the action was needed in the current image. When visiting an action in a context where the goal isn't (known to be) needed in image, or where the action is intrinsically not `needed-in-image-p` because its value resides in filesystem side-effects, then all the action's dependencies would themselves be visited in a mode where the goal isn't (known to be) needed in image. In that mode, the action is consulted for its timestamp, but won't be included in the plan as long as it's up-to-date. However, if the action is found to be out of date, before it would be planned, all its dependencies are visited a second time, in a mode where the goal is known to be needed in image. The top level action is initially requested with a goal of being needed in image, which only applies of course if it's itself a `needed-in-image-p` action.[42]

The code was then refactored by introducing an explicit plan object (2.26.47), to hold this action status information during the planning phase, as distinguished from the execution phase during which action status refers to what is actually done.

9.7 The Birth of `ASDF` 3

`ASDF` ended up being completely rewritten, several times over, to correctly address these core issues. The unintended result of these rewrites was to turn it into a much more robust and versatile product than it was: not only does it cover the robust building of CL software, it also includes runtime software management functionality and integration both ways with the Unix command line.

Considering the massive changes, I decided it should be called `ASDF` 3, even though a few months ago, I was convinced I would never write such a thing, since `ASDF` 2 was quite stable and I had no interest in making big changes. `ASDF` 3 was pre-released as 2.27 in February 2013, then officially released as 3.0.0 on May 15th 2013.

The numbering change itself triggered an interesting bug, because `ASDF` had adopted without documenting it the version compatibility check from Unix libraries, whereby an increase in the major version number indicates incompatibility. `ASDF` 3 thus considered itself incompatible with its `ASDF` 2, including with its pre-releases from 2.27 to 2.33. Since this compatibility check was undocumented and no one relied on it, and since it didn't make sense for CL software distributed as source code the way it did for Unix software distributed as object files, this led to a 3.0.1 release the next day, replacing the compatibility check so that a higher major version number still signifies compatibility.

Robert Goldman assumed maintainership in July 2013, a few months after the release of `ASDF` 3.0.1, and has since released 3.0.2 and 3.0.3. I remained the main developer until release 3.1.2, in May 2014, that culminates a new series of significant improvement.

All known bugs have been fixed except for wishlist items, but there will always be portability issues to fix. Also, while the regression test suite has swollen, many functions remain untested, and many of them probably include bugs. A big TODO file lists suggested improvements but it's uncertain whether a new active developer will ever implement them.

[42] The principle of visiting the action graph multiple times is generalizable to other situations, and the maximum number of visits of a given node is the height of the semi-lattice of node states during the traversal. For instance, in a Google build system extension I wrote to support CL, visited files would be upgraded between being not needed in image, needed loaded as cfasl, needed loaded from source, or needed loaded from fasl. The same technique could be used to improve XCVB.

9.8 Why Oh Why?

Some may ask: how did `ASDF` survive for over 11 years with such an essential birth defect? Actually, the situation is much worse: the very same bug was present in `mk-defsystem`, since 1990. Worse, it looks like the bug might have been as old as the original `DEFSYSTEM` from the 1970s.

The various proprietary variants of `defsystem` from Symbolics, Franz, and LispWorks all include fixes to this issue. However, the variants from Symbolics and Franz, require using a *non-default* kind of dependency, `:definitions`, as opposed to the regularly advertised `:serial`; also, the variant from Franz still has bugs in corner cases. Meanwhile the variant from LispWorks also requires the programmer to follow a non-trivial and under-documented discipline in defining build `:rules`, so you need to declare your dependencies in two related rules `:caused-by` and `:requires` that are akin to the original *depends-on* vs *do-first* in `ASDF` 1 (but probably predate it). What is worse, the *live* knowledge about this bug and its fix never seems to have made it out to the general Lisp programming public, and so most of those who are using those tools are probably doing it wrong, even when the tools allow them to do things right. **The problem isn't solved unless the bug is fixed by default**.

This is all very embarrassing indeed: in the world of C programming, `make` solved the issue of timestamp propagation, correctly, since 1976. Though historical information is missing at this point, it seems that the original `DEFSYSTEM` was inspired by this success. Even in the Lisp world the recent `faslpath` and `quick-build`, though they were much simpler than any `defsystem` variant, or quite possibly *because* they were much simpler, got it right on the first attempt. How come the bug was not found earlier? Why didn't most people notice? Why didn't the few who noticed *something* care enough to bother fixing it, and fixing it good?

We can offer multiple explanations to this fact. As a first explanation, to put the bug back in perspective, an analogy in the C world would be that sometimes when a `.h` file is modified in a different library (and in some more elaborate cases, in the same library, if it's divided in multiple modules), the `.c` files that use it are not getting recompiled. Put that way, you find that most C builds actually have the same problem: many simple projects fail to properly maintain dependency information between `.c` and `.h` files, and even those that do don't usually account for header files in other libraries, unless they bother to use some automated dependency analysis tools. Still, the situation is somewhat worse in the CL world: first because every file serves the purpose of both `.c` and `.h` so these dependencies are ubiquitous; second because because CL software is much more amenable to modification, indeed, dynamic interactive modification, so these changes happen more often; third because CL software libraries are indeed often lacking in finish, since tinkering with the software is so easy that users are often *expected* to do so rather than have all the corner cases painfully taken care of by the original author. In C, the development loop is so much longer, jumping from one library to the next is so expensive, that building from clean is the normal thing to do after having messed with dependencies, which often requires reconfiguring the software to use a special writable user copy instead of the read-only system-provided copy. The price usually paid in awkwardness of the development process in C is vastly larger than the price paid to cope with this bug in CL. Users of languages like Python or Java, where installation and modification of libraries is more streamlined by various tools, do not have this problem. But

then their programs don't have any kind of macros, so they lose, a lot, in expressiveness, as compared to CL, if admittedly not to C.

As a second explanation, most CL programmers write software interactively in the small, where the build system isn't a big factor. This is both related to the expressive power of the language, that can do more with less, and to the size of the community, which is smaller. In the small, there are fewer files considered for build at a time; only one file changes at a time, in one system, on one machine, by one person, and so the bug isn't seen often; when a dependency changes incompatibly, clients are modified before the system is expected to work anyway. Those who have written large software in the past tended to use proprietary implementations, that provided a `defsystem` where this bug was fixed. ITA Software was one of the few companies using `ASDF` to write really large software, and indeed, it's by managing the build there that we eventually cared enough to fix `ASDF`. In the mean time, and because of all the issues discussed above, the policy had long been to build from clean before running the tests that would qualify a change for checkin into the code repository.

As third, and related, explanation, Lisp has historically encouraged an interactive style of development, where programs compile very fast, while the programmer is available at the console. In the event of a build failure, the programmer is there to diagnose the issue, fix it, and interactively abort or continue the build, which eliminates most cases of the bug due to an externally interrupted build. Utter build failures and interruptions are obvious, and programmers quickly learn that a clean rebuild is the solution in case of trouble. They don't necessarily suspect that the bug is the build system, rather than in their code or in the environment, especially since the bug usually shows only in conjunction with such other bug in their code or in the environment.

As a fourth explanation, indeed for the `defsystem` bug to show without the conjunction of an obvious other bug, it takes quite the non-colloquial use of "stateful" or "impure" macros, that take input from the environment (such as the state of packages or some special variables) into account in computing their output expansions. Then, a change in a dependency can lead in expecting a change in the macro expansion, without the client site being modified, and that change will fail to take place due to the `defsystem` bug. But most macros are "stateless", "pure", and have no such side-effect. Then, a meaningful change in a macro defined in a dependency usually requires a change in the client file that depends on it, in which case the client will be recompiled after that change and no bug will be seen. The one case that the programmer may notice, then, is when the macro interface didn't change, but a bug in its implementation was fixed, and the clients were not recompiled. But the programmer is usually too obsessed with his bug and fixing it to pay close attention to a bug in the build system.

9.9 The Aftermath

At the end of this epic battle against a tiny old bug, `ASDF` was found completely transformed: much more sophisticated, yet much simpler. For instance, the commented `traverse-action` function is 43 lines long, which is still significantly less than the original `traverse` function. Reading the `ASDF` 3 source code requires much less figuring out what is going on, but much more understanding the abstract concepts — at the same time, the abstract concepts are also well documented, when they were previously implicit.

Interestingly, this new `ASDF` 3 can still meaningfully be said to be "but" a debugged version of Dan Barlow's original `ASDF` 1. Dan probably had no idea of all the sophistication required to make his `defsystem` work *correctly*; if he had, he might have been scared and not tried. Instead, he was daringly experimenting many ideas; many of them didn't pan out in the end, but most were clear improvement on what preceded, and he had quite a knack for finding interesting designs.

And the design of `ASDF` is undoubtly interesting. It masterfully takes advantage of the multiple inheritance and multiple dispatch capabilities of CLOS to deliver in a thousand lines or so a piece of software that is extremely extensible, and unlike anything written in languages missing these features. `ASDF` 3 is ten times this thousand lines, because of all the infrastructure for robustness and portability, because of all the burden of hot upgrade and backward compatibility, because of all the builtin documentation and comments, and because of all the extensions that it bundles. But the core is still a thousand lines of code or so, and these extensions, built on top of this core, illustrate its expressive power, as well as provide essential services to CL programmers.

In the end, we find that **software designs are discovered**, not created *ex nihilo*. Dan extracted a raw design from the mud of conceptual chaos, and gave birth to `ASDF`. Tasked with maintaining the software, I refined the design, removing the mud, until what was left was a polished tool. I certainly won't claim that my task was harder or more worthwhile than his, or that `ASDF` 3 is a jewel among build systems. But I believe that it has a clean and original design worth explaining, yet that neither Dan Barlow nor I can honestly be said to have designed this design; we merely stumbled upon it.

Bibliography

Henry Baker. Critique of DIN Kernel Lisp Definition Version 1.2. 1992. `http://www.pipeline.com/~hbaker1/CritLisp.html`

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`

Dan Weinreb and David Moon. Lisp Machine Manual. 1981. `https://bitsavers.trailing-edge.com/pdf/mit/cadr/chinual_4thEd_Jul81.pdf`

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

François-René Rideau. Software Irresponsibility. 2009. `http://fare.livejournal.com/149264.html`

Richard Elliot Robbins. BUILD: A Tool for Maintaining Consistency in Modular Systems. 1985. `ftp://publications.ai.mit.edu/ai-publications/pdf/AITR-874.pdf`