Symbols as Namespaces in Common Lisp

Alessio Stalla
alessiostalla@gmail.com

ABSTRACT

In this paper, we propose to extend Common Lisp's fundamental symbol data type to act as a namespace for other symbols, thus forming a hierarchy of symbols containing symbols and so on recursively. We show how it is possible to retrofit the existing Common Lisp package system on top of this new capability of symbols in order to maintain compatibility to the Common Lisp standard. We present the rationale and the major design choices behind this effort along with some possible use cases and we describe an implementation of the feature on a modified version of Armed Bear Common Lisp (ABCL).

CCS CONCEPTS

• **Software and its engineering → Modules / packages**; *Data types and structures*; Semantics.

KEYWORDS

Symbols, Namespaces, Packages, Hierarchy, Common Lisp, ABCL

ACM Reference Format:
Alessio Stalla. 2019. Symbols as Namespaces in Common Lisp. In *Proceedings of the 12th European Lisp Symposium (ELS'19)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.2648195

1 INTRODUCTION

1.1 Symbols

One of the distinguishing features of Lisp, that has set the language apart since its inception in the late 1950's [5], is the symbol data type. In Lisp, symbols are names for things, including parts of a program such as variables, functions, types, operators, macros, classes, etc. and including user-defined concepts and data. Each symbol has a name, which is a string, and a *property list*, an associative data structure where the language implementation, as well as libraries and programs, can store data and meta-data related to the symbol or associated concepts. The Lisp reader ensures that, when it encounters the same symbol name twice, it will return the same, identical symbol; so, when reading the textual representation of some source code, the same name appearing as a different string in several positions in the source text will refer to the same concept or program element.

Symbols are a data type usually found in compilers and interpreters. Lisp exposes the symbol concept to the user of the system and is itself built upon it – it is, in its core, a system for the manipulation of lists of symbols. This makes programming in Lisp qualitatively different from other programming languages [3], be they object-oriented, functional, imperative, and so on. Usually we tend to concentrate on lists (or, better, on conses) and to forget about the importance of symbols.

1.2 Packages

Early Lisps had a single namespace for symbols [6]. That is, a single global hash table that the Lisp reader consults when it encounters a symbol name: if it's already present in the table, the associated symbol is used, otherwise a fresh symbol is created and stored in the table. This operation is called `INTERN`. The table itself is called an `obarray` in Maclisp [8], Emacs Lisp [2], and perhaps other Lisps.

A single namespace suffers from the possibility of symbol clashes – that is, two programs assigning incompatible meanings to the same symbol. Eventually, this problem was addressed in Common Lisp with its *package system* [9], derived from an earlier system introduced in Lisp Machine Lisp [12].

In Common Lisp, packages are objects that map symbol names (strings) to symbols. More than one package can be defined; indeed, the Common Lisp standard defines three built-in packages, `COMMON-LISP` for the language itself, `KEYWORD` for keyword symbols that are constants that evaluate to themselves, and `COMMON-LISP-USER` for user symbols. However, exactly one package is current at any one time (per thread): it is, by definition, the value of the special variable `*PACKAGE*`.

The reader always interns unqualified symbol names in the current package. To refer to a symbol in another package, the symbol name is prefixed by the package name followed by a colon – or two, in certain cases which we'll explain shortly. For example, `ALEXANDRIA:WHEN-LET`.

Packages, like symbols, have a name which is a string. Additionally, packages can have multiple secondary names called *nicknames*. For example, the `COMMON-LISP` package is nicknamed `CL`. And, just like symbols in earlier Lisps, packages are registered in a single global map keyed by their names and nicknames. It is possible to remove a package from the map, using the `DELETE-PACKAGE` operator (which removes at once all the entries referring to that package, including nicknames). However, the Common Lisp standard does not specify any meaningful way in which a package object can be used once it's been deleted, and it does not define any operator to put a package in the global map back again. In practice, in a portable Common Lisp program, a deleted package is no longer usable in any way.

The package system also acts as a simple but effective read-time module system. A package can export some of its symbols; other packages can import them individually or they can *use* another package, thus automatically importing all its exported symbols. The names of non-exported symbols need to be prefixed with two colons when referring to them from other packages.

ELS'19, April 01–02 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2648195

2 PACKAGE PROBLEMS

Packages are arguably one of the most criticized (or poorly understood) features of the Common Lisp language:

- One issue is that the package system is not a very advanced module system, or not much of a module system at all. By design, it is just a system for organizing names and avoiding or handling clashes.
- Another issue is that the package system works at read-time, thus it relies on, and suffers from, read-time side effects [4].
- Then, as we've previously said, package names and nicknames live in a global shared namespace. With the ever-increasing amount of libraries (the Quicklisp distribution contains more that 1500 libraries [1]), each defining at least one package but quite often more than one, the possibility for package name clashes is real, especially considering the existence of nicknames which are often short mnemonic names or acronyms.
- Finally, since packages cannot portably exist as usable objects outside their global namespace, solutions using temporary or "unnamed" packages are awkward and feel hacky.

In this article, we'll focus on the last two issues: a single, global namespace for package names and nicknames, and the fact that packages are tied to this namespace.

3 EXISTING EXTENSIONS OF THE PACKAGE SYSTEM

Naming issues such as potential clashes need not necessarily be solved with technique alone. Solutions based on social conventions are often employed successfully. For example, the C language does not have any provision for namespacing. Developers simply prefix the names of functions and variables to avoid clashes.

Also, the package system itself natively provides tools for dealing with naming conflicts, such as the aforementioned nicknames and the `RENAME-PACKAGE` function.

Still, a number of extensions to the package system have been proposed and implemented over the years. We'll now review a couple of those.

3.1 Hierarchical Packages in Allegro Common Lisp

Allegro Common Lisp augments packages with some hierarchical structure [10]. Taking inspiration from languages such as Java and C#, it proposes a naming convention according to which package names are a sequence of names separated by dots.

The hierarchical structure in Allegro is just a naming convention; it does not mandate correspondence to a directory and file structure, as in Java. However, the convention is understood and enforced by a few functions that also allow users to shorten package names when they have some substructure in common, in a way that resembles file system paths. For example, `(find-package :.test)` would return the package `it.alessiostalla.app.test` if the current package were `it.alessiostalla.app`.

An informal survey by the author revealed that hierarchical packages appear to be seldom used, if at all, outside the internals of Allegro Common Lisp itself. Interestingly, the same facility exists as an open-source library by Pascal Bourguignon (https://gitlab.com/com-informatimago/com-informatimago/blob/master/common-lisp/lisp/relative-package.lisp).

3.2 Package-local Nicknames in SBCL and Other Lisps

Other Common Lisp implementations, in particular at least SBCL [11] and ABCL, allow to define package-local nicknames. That is, a package, say P, can specify local nicknames for other packages. When P is the current package, and only then, those nicknames can be used to refer to the nicknamed packages. Thus, the local nicknames do not pollute the global package namespace. Therefore, users can shorten frequently-used package names without fearing collisions with other unrelated packages that happen to have the same nickname.

This apparently simple feature is nevertheless quite powerful, and indeed, from an informal survey by the author of this paper, it is actively used or at least well regarded by a number of experienced developers on the Common Lisp Professionals mailing list.

4 SYMBOLS AS NAMESPACES

Let's now analyze the more radical option of using symbols as namespaces for other symbols. How would it look like? Is it worthwhile? Is it possible to extend Common Lisp with such a feature and to deprecate packages while maintaining backwards compatibility with the Common Lisp standard and with existing Common Lisp code?

4.1 What We Want to Achieve

We want to use symbols as namespaces for other symbols, and so on recursively. In other words, we want every symbol to potentially act as a package. We want these extended symbols to be recognized by the Lisp reader and Lisp printer. We'll use the syntax `foo:bar:baz` to represent a symbol named `baz`, which is an external symbol in the symbol named `bar`, which is itself an external symbol in the symbol named `foo`. So the idea is to extend the Common Lisp syntax to allow multiple, non-consecutive package markers. According to the Common Lisp standard, the treatment of tokens with multiple package markers is undefined and implementation-dependent [7], so this extended syntax is allowed by the standard. Similarly, we'll use two consecutive colon characters to refer to internal symbols.

Symbols that are not interned in other symbols, or in other words have no parent symbol, are called *root symbols*. We want to have exactly one canonical root symbol, that we'll denote here as `#<ROOT>`, such that the syntax `:foo` represents a symbol named `foo` whose parent is the canonical root symbol. Other root symbols can be created (for example, with the standard function `MAKE-SYMBOL`), but only the canonical one gets the special syntax described here. The canonical root symbol cannot be replaced. It is printed as a an empty string, thus it cannot be read by itself.

In general, we'll need to add new operators to our Common Lisp implementation to support these new features. Rather than adding them in a new package, we have chosen to intern them in the `SYMBOL` symbol, as it will become apparent in the next section.

4.2 Backwards Compatibility

The idea is to retrofit packages as a facade over namespacing symbols, in order to maintain Common Lisp compatibility for existing programs that do not make use of, or even know about, our new symbol type. So, our extended symbols will need to retain all the package features such as exporting, USEing other namespaces, shadowing, etc. We could leverage the symbols' property lists for those features, or we could add implementation-dependent fields to the symbol type itself. We provide an operator, `(symbol:as-package symbol)`, that given a symbol will return a package object reflecting that symbol's name and contents.

Furthermore, since packages can be given nicknames, our new symbols must support the same feature if we want them to replace packages. So, it must be possible to intern the same symbol with different names (aliases) in the same or in different namespaces. This is an important difference from Common Lisp, where symbols either have exactly one home package, or they are *uninterned*. Our symbols can have multiple aliases in multiple namespaces, but they always have a name and a parent symbol (or NIL). We provide two operators:

```
(symbol:alias symbol alias &optional export)
```

to create an alias of a symbol in its parent namespace, and

```
(symbol:remove-alias symbol alias)
```

to remove an alias. To create an alias of a symbol in a different namespace, we use the standard `IMPORT` function, which of course now works on symbols as well as on packages. Note that the hierarchical nature of symbols implies that aliases are local, just like the package-local nicknames extension in SBCL and ABCL.

5 THE ROOT SYMBOL, KEYWORDS AND TOP-LEVEL PACKAGES: A PROBLEM

It is apparent that keywords and symbols in the root namespace have something in common. In part this comes from the choice of syntax: since `foo:bar` is symbol `bar` with parent `foo`, `:foo` ought to designate the symbol `foo` with parent `#<ROOT>`, just like paths in a file system. But `:foo` in Common Lisp is already the syntax for the keyword named `foo`. It's not only a matter of the choice of syntax, though. Keywords are meant to be read uniformly and unambiguously no matter what the current package happens to be. Analogously, one is supposed to be able to reach to the root namespace with the same syntax no matter what the current namespace is. Even not considering syntax, keywords and symbols in the root namespace appear to be similar beasts.

Another seemingly obvious choice is that legacy Common Lisp packages – which are inherently top-level, global names – ought to live in the root symbol, so that package `COMMON-LISP` is actually the symbol `#<ROOT>:COMMON-LISP`, which is then printed as `:COMMON-LISP`. Thus the root symbol is the global map that contains all packages, which in standard Common Lisp is an object that users of the language cannot access.

These two apparently natural choices, however, don't play well together. In fact, to preserve backwards compatibility, the Lisp reader, when searching for a package (say, `COMMON-LISP`), must either search it locally to the current namespace (we'll call this option L for Local first), or it must search it in the root first, then in the current namespace (option R for Root first).

Choice (L) implies that, for, say, `CL:LIST` to be read consistently everywhere, every package must import the `:CL` symbol. More generally, every symbol which denotes a package must be accessible (imported) in every namespace. But if the symbol, such as `:CL`, is also a keyword by design, then it is a constant and it cannot be rebound, not even locally. This is a strong limitation and a problem for backwards compatibility, especially for packages with common names like `SEQUENCE`, `SYSTEM`, `EXTENSIONS` etc. which collide with symbols in the `COMMON-LISP` package or with symbols in user code. Clearly, having packages named by keywords and requiring all namespaces to import those keywords has heavy usability implications.

Choice (R) instead implies an inconsistency. In the expressions `CL:X` and `CL`, the two character strings "CL" might be read as different symbols: the first as `#<ROOT>:CL`, the second as, for example, `CL-USER:CL`. Also, with that scheme, `:KEYWORD` would be a symbol whose namespace is itself, which is confusing, but this is probably just a minor annoyance.

6 AN IMPERFECT SOLUTION

We can then decide that the root symbol is not the keyword package after all. This, however, has other problems. In fact, there is a read inconsistency for keywords. `:foo` must be read as a keyword at least for backwards compatibility, but in the expression `:foo:bar`, `foo` is not a keyword, it is symbol `FOO` in symbol `#<ROOT>`.

Also, there is still the inconsistency of, e.g., `SEQUENCE` in the expressions `'SEQUENCE:COUNT` and `'SEQUENCE` being two different symbols if `SEQUENCE` is a top-level package, unless the symbol `SEQUENCE` is imported in the current package. This is for backwards compatibility, because if a user evaluates `(defpackage foo)`, Common Lisp mandates that `foo::x` refers to the top-level package `FOO` no matter what the current package is.

However, there isn't the additional limitation of keywords being constants, so `SEQUENCE`-the-package and `SEQUENCE`-the-CL-symbol can be arranged to be the same symbol without drawbacks, by importing `:sequence` in CL (or by importing `cl:sequence` into `#<ROOT>`). For system packages, the implementation can probably arrange things like that automatically, so for users it's transparent. For user packages, this cannot be done by the implementation, users have to write the boilerplate manually if they want to avoid the inconsistency.

Ideally, if no backwards compatibility were required, we could mandate that the names of top-level packages be always prefixed by a colon – as in `:cl:count` and `:sequence:count` – unless locally imported. However, in an existing Common Lisp system, this is not possible. Legacy compatibility could be turned on and off with flags, but this still seems a bit of a mess. Things don't click, they're too complex. The beauty of the original idea seems lost.

7 THE REAL SOLUTION

Things flow much better if we take a different route. Namely, that top-level packages (actually, their names) are not top-level symbols. Instead, let's make them live in another, non-root symbol, say `:TOP-LEVEL-PACKAGES`. The root symbol, then, continues to be the home namespace of keyword symbols, that is, the Common Lisp keyword package, and the only "special" symbol and package in the system.

So, when it encounters the expression `foo:bar`, the reader looks for `foo` in the current namespace first; if it is not present or it is not a namespace, i.e. it doesn't contain other symbols (a distinction that is necessary to avoid excessively shadowing top-level packages), then it continues its search in the symbol `:TOP-LEVEL-PACKAGES`. An inconsistency can still happen – the expressions `foo:bar` and `foo` referring to different `foo` symbols – but only if a local symbol named `foo` exists and it is not used as a namespace. In that case, it is reasonable that the same sequence of characters "foo" refers to different things according to whether it's denoting a namespace or a symbol name – after all, that's how today's Common Lisp works.

As a minor annoyance, to spell, say, the `CL:LOOP` symbol in its absolute form, one must write `:top-level-packages:cl:loop`; that is, the abstraction leaks a bit.

8 IMPACT ON COMMON LISP ASSUMPTIONS

The change we are proposing cuts deep in the fundamentals of Common Lisp and arguably of Lisp as it was first conceived. What are the consequences?

One area that should definitely be explored as further work is read-print inconsistencies, as it is apparent from the previous sections. We haven't studied the issue enough to report something meaningful here.

Another problematic impact is the interplay with the function `DELETE-PACKAGE`.

8.1 DELETE-PACKAGE

In Common Lisp, `DELETE-PACKAGE` removes a package with all its nicknames from the global namespace and renders the package object unusable in portable code. In our extended Common Lisp, `DELETE-PACKAGE` should do the same thing to any symbol.

However, with symbols being potentially imported, exported, aliased, used as namespaces in several places, to delete a symbol atomically from the system requires a lock on the whole symbol system. And, even ignoring concurrency issues, there are new possibilities for failure that are currently absent in Common Lisp. Removing an alias can uncover a conflict between two used packages, for example. `DELETE-PACKAGE` is necessary because packages do not exist outside their global map; symbols do live just as well without a parent, and they can be uninterned. Once a symbol is no longer referenced by any live object it can be garbage collected. So, if our proposal is to be adopted, `DELETE-PACKAGE` should be restricted to work only on symbols and aliases in `:TOP-LEVEL-PACKAGES`, or it should be deprecated altogether in favor of `REMOVE-ALIAS` and `UNINTERN`, or both.

9 APPLICATIONS

So far hierarchical symbols might seem just a cool feature, a bizarre experiment or a hack for the sake of hacking. In our opinion, they make packages "better citizen" in a world where everything is a first-class object and can be created, manipulated and discarded at will. They also arguably (if we don't consider the backwards-compatibility complexities) provide a better, more consistent design of symbols as "things that give names to things", all the way down. However, they have practical applications, too. Here we propose one and hint at a few others.

9.1 A File System Facility

Common Lisp's pathnames are another frequently debated feature that is known to have made most users scratch their heads in confusion. Here we propose a simple library that provides an easier API for basic pathname usage, leveraging hierarchical symbols. This example will show how hierarchical symbols are a versatile feature that allows to represent all sort of hierarchical names in the language and will showcase some of the functions supporting our new symbols.

The key idea of this small library is to represent pathnames as symbols. One can mount a physical pathname, with its implementation-dependent syntax, to a given symbol. Then one can construct related pathnames by interning symbols in it and so on recursively, without setting up complex translations, manipulating strings or using the awkward Lisp pathnames API. When done with it, one can also unmount a symbol, i.e. remove all filesystem-related information from it.

So by evaluating `(mount 'foo (user-homedir-pathname))` one can represent paths such as `'foo::Downloads::virus.exe` (having readtable-case set to `:PRESERVE` helps as file systems can be case sensitive). The pathname of a given symbol can be obtained with the `pathname` operation:

```
(pathname 'foo::Downloads::virus.exe)
=> #P"/Users/alessio/Downloads/virus.exe"
```

Symbols-as-pathnames can be tested for existence, opened for reading or writing, and operated upon in all the ways supported by the native Common Lisp pathname facility.

9.2 FFI and Interoperation

There are other areas where having composite, hierarchical names can be beneficial. One is, of course, interoperation with languages that themselves have, or simulate, such names. For example, a Java FFI could allow the following:

```
(jffi:import 'java:lang:String) ;optionally :as 'java-string
(jffi:new 'String "a string") ;New object creation
(String::valueOf 42) ;Static method call
(String::toCharArray **) ;Instance method call
```

9.3 Addressing Other Kinds of Paths

Generally, every time we might want to map paths or hierarchies to symbolic data, hierarchical symbols can offer an advantage. For example:

- JSON or XML paths (XPath)
- mapping objects to database systems (e.g., schema:table:column)
- representing URL's and network path
- invoking remote functions, services, procedures

10 IMPLEMENTATION

An implementation of the above concepts and a few support functions has been realized on Armed Bear Common Lisp (ABCL). ABCL is a Common Lisp running on the JVM, written in a combination of Java and Lisp and with a significant amount of code inherited from CMUCL/SBCL.

The result is a working ABCL that has the symbol data type described earlier and fails the same ANSI tests it failed before the changes. Most modifications involved only 4 files: Symbol.java (the symbol type), Package.java (the package type), Primitives.java (primitive functions) and Stream.java (where most of the reader is defined).

As an additional consequence, ABCL's serialization of symbols in FASL files, which was brittle, broke irreparably and was rewritten to be more solid (basically printing symbols with *print-readably* bound to T, which causes them to be printed in their absolute form starting from the root, e.g. :TOP-LEVEL-PACKAGES::COMMON-LISP::T). However, this causes a certain increase in FASL size and a deterioration of load times, particularly at startup or when loading big systems, which are already a pain point in ABCL.

The implementation can be found at https://github.com/alessiostalla/abcl on the branch hierarchical-symbols.

11 FURTHER WORK

The work here is just a foundation. The implications of hierarchical symbols in Common Lisp should be analyzed further. In particular, that there are no read-print inconsistencies in corner cases.

A low-hanging fruit is to enhance our work by providing a few missing usability features. For example, since symbols can be aliased, have an :import-from :as option in defpackage/define-namespace.

A particular area of interest is the porting of the feature to other Lisp implementations. Implementing it on ABCL has been relatively easy, but it might be just a fortunate case. Also, investigating whether it is possible to implement this proposal in pure Common Lisp, with no modifications to the implementation, is a worthwhile goal.

ACKNOWLEDGMENTS

Thanks to the ABCL developers and the users of the Common Lisp Professionals mailing list for their insight and experience.

REFERENCES

[1] Zach Beane. Quicklisp beta. URL https://www.quicklisp.org/beta/.
[2] Inc. Free Software Foundation. Gnu emacs lisp reference manual. URL https://www.gnu.org/software/emacs/manual/html_node/elisp/Creating-Symbols.html#Creating-Symbols.
[3] Richard P. Gabriel. The structure of a programming language revolution. URL https://www.dreamsongs.com/Files/Incommensurability.pdf.
[4] Ron Garrett. Lexicons: First-class global lexical environments for common lisp. URL http://www.flownet.com/ron/lisp/lexicons.pdf.
[5] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.
[6] John McCarthy. *LISP 1.5 Programmer's Manual.* The MIT Press, 1962. ISBN 0262130114.
[7] Kathy; et al. Pitman, Kent; Chapman. The common lisp hyperspec - section 2.3.5 valid patterns for tokens, . URL http://www.lispworks.com/documentation/HyperSpec/Body/02_ce.htm.
[8] Kent Pitman. The revised maclisp manual (the pitmanual), . URL http://www.maclisp.info/pitmanual/symbol.html#10.9.1.
[9] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.).* Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.
[10] Unknown. The allegro common lisp documentation - packages, . URL http://franz.com/support/documentation/current/doc/packages.htm.
[11] Unknown. Sbcl 1.4 user manual, . URL http://www.sbcl.org/manual/#Package_002dLocal-Nicknames.
[12] Daniel Weinreb. *Lisp Machine Manual.* Massachusetts Institute of Technology, Cambridge, MA, USA, 1981. ISBN B0006Y4UVA.